Daixtrose User Manual

Markus Werle

March 20, 2003

Contents

1 Installation Instructions

1.1 System Requirements

Daixtrose is (or at least should be) platform independent. It's only a C++ header library, nothing more. If You want to use the GNU autotools based installation procedure which compiles the testsuite, any unix system or a similar environment like cygwin is nice to have, but this library is intended to be platform independent.

1.2 Software Requirements

Daixtrose depends on the following software

- A modern high performance ANSI compliant C++ compiler.
  This library exploits most of the bright and dark corners of C++ templates. At present only Intel's C++ version 7.0 with option -ansi is known to compile this code. GNU gcc-3.4 will be able to compile this, since it only suffers from a parser bug in using declarations, which is promised to be fixed soon.
- the `boost` library (see `http://www.boost.org`)
- the `loki` library (see `http://sourceforge.net/projects/loki-lib`)

This library is completely header-based and a simple `#include "SomeHeader.h"` will do. The most important one is `daixtrose/Daixt.h` which includes all of the core engine. But of course You may `#include` the files in a more refined way.

Daixtrose is shipped with a fixed-sized matrix and vector package. `#include "tiny/TinyMatAndVec.h"` in order to use it. **Daixtrose** also comes with a (yet incomplete) linear algebra package (sparse matrix + vector). In order to use these one needs to `#include "linalg/Linalg.h"`

1.3 Autotools Based Installation Procedure

To ease the compilation of the testsuite and in order to support possible future diversification of the library, daixtrose is shipped with a GNU autotool-based configure shell script.

Perform the following steps (probably with some modifications), if You like to get the examples built (it is recommened to create a new directory *outside* of this directory, to keep the original source tree clean):

```
$ cd ..
$ mkdir build-daixtrose
$ cd build-daixtrose
$ ../daixtrose-0.0.1/configure \
CXX=icpc CPPFLAGS="-g -wd76 -ansi -O3 -Ob2" \
--prefix=$HOME/daixtrose-0.0.1 \
--with-loki=/path/to/loki/prefix \
--with-boost=/path/to/boost/prefix

$ make
$ make check
$ make install
```

At present the command `make` will do nothing.

The command `make check` is optional and may be omitted. It evokes compilation of all the examples in the directory `demos` and can be regarded as a compiler test.

The command `make install` will install the headers under `prefix/include`.

For further control over the installation procedure consider the output of `../daixtrose-0.0.1/configure --help`.

1.4 Update of the Library

This lib undergoes permanent evolution. So it is a good idea to consider regular updates. Visit the homepage of **Daixtrose** at `http://daixtrose.sf.net` for download instructions.

A more frequent update is possible through checkouts of the cvs repository. The following commands are sufficient to check out the latest version:

```
$ cvs -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/daixtrose login
$ cvs -z3 -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/daixtrose \
  co daixtrose
```

You may also visit the cvs web-interface at `http://sourceforge.net/cvs/?group_id=75079`

2 Code Organization

The code of **Daixtrose** is organized as follows:

All sourcecode is located in the directory `src`.

- directory `daixtrose` constains the core Expression Template library
- directory `tiny` contains the fixed sized matrix and vector library built on top of the **Daixtrose** core library
- directory `linalg` contains a linear algebra library mainly consisting of an STL-based sparse matrix implementation with compatible vector classes.
- directory `demos` contains smaller examples which give an idea about the features offered by the already mentioned parts of the library.

3 Quick Tour

You want to know what **Daixtrose** is good for? This is an attempt to explain it within 10 minutes. Built on top of the core **Daixtrose** library are two matrix and vector library snippets, one for applications where the sizes of matrices or vectors are known at compile time and one for large scale applications with sparse storage of the matrix and a more flexible approach.

Both primarily were intended as demo code to show the power of **Daixtrose**, but they already do a decent job in the authors doctoral thesis project, a time-implicit finite element solver. Therefore you can profit from Daixtrose even if you do not want to build your own Expression Template library on top of **Daixtrose**, but rather are only in need of a fairly good matrix and vector library.

All code examples presented here can be found in the directory `src/demos/quicktour/` and will get compiled during a `make check` as described in the installation instructions (section1).

3.1 Fixed-Sized Matrix and Vector

`TinyMatrix` and `TinyVector` are inspired by Todd Veldhuizen's `blitz++` library. They are one-based (first index is 1 not 0) and can be used like this:

```
#include <iostream>
#include "tiny/TinyMatAndVec.h"

int main()
{
  // all features are hidden in sub-namespaces and have to be made
  // visible when needed, here we want to use the output utilities and
  // we want to work with matrix-vector expressions
  using namespace Daixt::OutputUtilities;
  using namespace Daixt::DefaultOps;

  // a shorthand saves us from a lot of typing
  using namespace TinyMatAndVec;
  typedef TinyVector<double, 3> TinyVec;
  typedef TinyQuadraticMatrix<double, 3> TinyMat;

  TinyMat M1; // no data initialization (for efficiency reasons)
  TinyMat M2 = 3.0; // all entries are initialized with the value

  M1 = 2.0; // set all entries at once

  M1(2, 1) = 15.99; // The index operator

  TinyMat M3 = M2 + Transpose(M1);

  // Some of the things one expects from a tiny matrix lib
  M3 *= 0.5;
  M3 -= 1.0;
```

```
  // note that the next operation does not produce any temporaries and
  // therefore the rhs might be completely optimized away
  TinyMat M4 = M1 * M3;

  // pretty print output
  std::cerr << M1 << M2 << M3 << M4 << '\n';

  TinyVec V1 = 5.0;
  V1(2) = 0.0;

  // no temporaries - and a looong compile time ;-)
  TinyVec V2 = M1 * M2 * (M3 - M4) * V1;

  // delimiters required
  std::cerr << (V2 == M1 * M2 * (M3 - M4) * V1);
}
```

3.2 Linalg: Sparse Matrix and Vector

The sparse matrix implementation presented here assumes a row-oriented usage like e.g. in matrix-vector multiplications or additions resp. subtractions. The matrix data is stored in a `std::vector` of user-definable `RowStorages` that must behave like a `std::map<size_t, T>`, which is the default. Each line in the matrix is represented by a `RowStorage`, where the keys store the colum number. In addition to the common matrix interface (index operator, etc.) the user additionally has read-only access to each line or column, which allows individual manipulations if needed. Like for `TinyMatrix` the sparse matrix and the compatible vector are one-based.

Due to the sparse representation of the matrix it is not completely possible to avoid all temporaries during assignment of matrix expressions, but the run-time and memory overhead is negligible for matrices with more than 100 lines. The assignment loop creates new rows and inserts them at the correct places of the asignee.

Self assignment and assignment from expressions which contain the assigned Matrix or Vector will be detected at runtime and will safely produce a temporary before final assignment.

To allow complete, fine grained control over memory management, the allocators for the `RowStorage` can be optionally chosen to be different from `std::allocator`

Here is some code which demonstrates the features which are available. (Mentioning the allocators in the typedefs is not needed, but for demonstration purposes the defaults are shown in the following example).

```
#include "linalg/Linalg.h"

#include <map>
#include <cstddef>
#include <iostream>
```

```
int main()
{
  // Shortcuts. Mentioning the allocators is not needed, here the defaults are shown
  typedef Linalg::Matrix<double,
                         std::map<std::size_t, double>,
                         std::allocator<std::map<std::size_t, double> > > Matrix;

  typedef Linalg::Vector<double,
                         std::allocator<double> > Vector;
  // use operaor+,-,*,/ etc
  using namespace Daixt::DefaultOps;
  // allows multiplication of non-disambigauted scalars
  using namespace Daixt::Convenience;
  // output routines
  using namespace Daixt::OutputUtilities;


  std::size_t n = 3; // Only for demontration purposes: this number is a little
                     // bit small, linalg was designed for larger scales

  Matrix M1(n, n);

  // all possible ways to fill in some matrix data:
  for (size_t i = 1; i != n+1; ++i) M1(i, i) = 42.0;

  M1(1, 2) = 21.0;
  M1.SetEntriesInRowTo(2, 3.14);
  M1.SetEntriesInColTo(3, 42.5);


  Matrix M2 = M1;

  Matrix M3(n, n);
  M3 = M1;

  M1.ReplaceRow(2, M1(3));

  M2 += M3;
  M2.swap(M3);

  Matrix M4 = M2 + Lump(Transpose(M1));
  M2 = 0.0;

  Matrix M5(n, n);
  M5 = M2 - M3 + M1;

  Vector V1(n);
  std::fill(V1.begin(), V1.end(), 1.0);

  Vector V2 = (2.0 * M2 - Lump(M1)) * V1;

  std::cerr << M1 << M2 << M3 << M4 << M5 << V1 << V2 << std::endl;
}
```

3.3 A Sparse Matrix of Fixed-Sized Matrices

Of course you can have a sparse matrix of fixed-sized matrices. That is easy enough to achieve:

```
typedef TinyMat::TinyQuadraticMatrix<double, 3> BlockMatrix;
typedef Linalg::Matrix<BlockMatrix> Matrix;

typedef TinyVec::TinyVector<double, 3> BlockVector;
typedef Linalg::Vector<BlockVector> Vector;
```

You can find complete examples in `src/demos/linalg/TestBlockedMatAndVec.C` `src/demos/linalg/TestPrintingOfBlockedMatrix.C`

3.4 Using the Core Engine (Building Expression Templates at Home)

3.4.1 Delaying Evaluation

Daixtrose was designed to provide a convenient way to apply Expression Template techniques to any kind of C++ classes without the need to modify any line of code of these classes. As an example consider you have written a very elaborated high performance `class X`. The only things missing are the operators and other code which would allow objects of type `X` to be used within expressions. No problem for us, we have **Daixtrose**:

```
#include "daixtrose/Daixt.h"

class X {};


int main()
{
  X x1, x2, x3;

  using namespace Daixt::DefaultOps;

  (
   (x1 + x2)
   *
   RationalPow<4, 3>(x3) / Pow<2>(x2)
   )
  /
  ((-x1) * x3);
}
```

Congratulations! You have combined the power of `class X` with the invaluable features of **Daixtrose**. This code already compiles. Of course there is some lack of functionality, but we fix that in a minute.

What is important here is that with the help of `namespace Daixt::DefaultOps` any expression gets mapped unto an Expression Template tree. E.g. the expression above results in contructor calls for `Expr<T>`, `BinOp<...>` and `UnOp<...>` and yields an object of type[1] (big breath)

```
Expr<
     BinOp<
           BinOp<
                 BinOp<
                       BinOp<X, X, BinaryPlus>,
                       UnOp<X, RationalPower<4, 3> >,
                       BinaryMultiply
                       >,
                 UnOp<
                      X,
                      RationalPower<2, 1>
                      >,
                 BinaryDivide
                 >,
           BinOp<
                 UnOp<X, UnaryMinus>,
                 X,
                 BinaryMultiply
                 >,
           BinaryDivide
           >
     >
```

This is what **Daixtrose** offers. Users (or library authors) do not have to retake the tedious task of reinventing yet another Expression Template machinery but can rely on the **Daixtrose** Expression Template core engine.

Expression Templates avoid temporaries through delayed evaluation. The first step of a delayed evaluation, namely the delaying, has already been taken in the code above. Now it is up to the user to perform the evaluation.

3.4.2 Expression Evaluation

Evaluation can be achieved with a set of template functions (or a functor with an overload set of template member functions alias `operator()`. These functions can rely on the public interfaces of the **Daixtrose** classes, especially on the member functions

```
Expr<T>::content()
BinOp<...>::lhs()
BinOp<...>::rhs()
UnOp<...>::arg()
```

To illustrate this, let us extend the example above:

[1] This expression tree has been simplified. All class names reside inside namespaces, which were omitted for the sake of readability

```
#include "daixtrose/Daixt.h"
#include <iostream>
#include <string>

class X
{
  std::string Name_;
public:
  X(const std::string& Name) : Name_(Name) {}
  std::string Name() const { return Name_; }
};

std::ostream& operator<<(std::ostream& os, const X& x)
{
  os << x.Name();
  return os;
}


template <class T>
void Evaluate(const Daixt::Expr<T>& arg)
{
  Evaluate(arg.content());
}

template <class LHS, class RHS, class OP>
void Evaluate(const Daixt::BinOp<LHS, RHS, OP>& arg)
{
  Evaluate(arg.lhs());
  std::cerr << OP::Symbol();
  Evaluate(arg.rhs());
}

template <class ARG, class OP>
void Evaluate(const Daixt::UnOp<ARG, OP>& arg)
{
  std::cerr << OP::Symbol();
  Evaluate(arg.arg());
}

void Evaluate(const X& x)
{
  std::cerr << x;
}

int main()
{
  X x1("x1"), x2("x2"), x3("x3");

  using namespace Daixt::DefaultOps;

  Evaluate((x1 + x2) / ((-x1) * x3));
}
```

3.4.3 Assignment from an Expression

In a similar manner assignment can be achieved by reusing existent public interfaces. Consider a case where `class X` has a public member function that allows resetting some internal value. Here we show an example where no temporary object of type `X` is created, but only temporray objects of the contained data type are created. Applying this technique to classes which contain large data sets like e.g. sparse matrices is the key to runtime efficiency.

```
#include "daixtrose/Daixt.h"
#include <iostream>
#include <string>

class X
{
  std::string Name_;
public:
  X(const std::string& Name) : Name_(Name) {}
  std::string Name() const { return Name_; }
  void ResetName(const std::string& NewName)
  {
    Name_ = NewName;
  }
};

std::ostream& operator<<(std::ostream& os, const X& x)
{
  os << x.Name();
  return os;
}


template <class T>
std::string Evaluate(const Daixt::Expr<T>& arg)
{
  return Evaluate(arg.content());
}

template <class LHS, class RHS, class OP>
std::string Evaluate(const Daixt::BinOp<LHS, RHS, OP>& arg)
{
  return
    Evaluate(arg.lhs())
    + " " + OP::Symbol() + " "
    + Evaluate(arg.rhs());
}

template <class ARG, class OP>
std::string Evaluate(const Daixt::UnOp<ARG, OP>& arg)
{
  return OP::Symbol() + Evaluate(arg.arg());
}
```

```
std::string Evaluate(const X& x)
{
  return x.Name();
}

// operator= must be a member function,
// but maybe we cannot or do not want to modify X,
// so we use operator%= here
template<class T>
X& operator%=(X& x, const Daixt::Expr<T>& E)
{
  x.ResetName(Evaluate(E));
  return x;
}

int main()
{
  X x1("x1"), x2("x2"), x3("x3");

  using namespace Daixt::DefaultOps;

  X x4("x4");

  std::cerr << x4 << " = ";
  x4 %= ((x1 + x2) / ((-x1) * x3));
  std::cerr << x4;
}
```

Note that in the example shown here assignment from an expression can be performed without modifying `class X`, but of course adding an appropriate `operator=` to `class X` is the way of choice in cases where `class X` was designed to rely on **Daixtrose**:

```
class X
{
  std::string Name_;

public:
  X(const std::string& Name) : Name_(Name) {}

  template<class T>
  X& operator=(const Daixt::Expr<T>& E)
  {
    Name_ = Evaluate(E));
    return *this;
  }

  std::string Name() const { return Name_; }
};
```

In order to ease the output of expressions, `namespace Daixt::OutputUtilities` contains everything needed to output any expression. This feature works for any leaf `class T` that has a

`std::ostream& operator<<(std::ostream& os, const T& t)` defined. See how this feature was used in the sparse matrix example subsection3.2 above.

3.4.4 Disambiguation

Sometimes it is a good idea to let the compiler emit an error if incompatible types are combined in an exprression. Perhaps one does not want a `Vector` added to a `TesorRankOne` to pass the compiler ending in long debugging sessions.

Daixtrose takes care of preventing the combination of incompatible types into one expression and provides fine-grained control over its behaviour. The disambiguation mechanism compares types. Per default it takes the type of the leaves of the expression tree (`class X` in our example) as disambigution type.

There are two other possibilities for providing a disambiguation type: classes can contain a typedef like in

```
struct X_Disambiguation {};

class X {
  typedef X_Disambiguation Disambiguation;
};
```

or one can specialize `Daixt::Disambiguator` which again allows plugging class X into the Expression Template engine without modification:

```
class X {};

struct X_Disambiguation {};

namespace Daixt
{
template <>
struct Disambiguator<X>
{
  static const bool is_specialized = true;
  typedef X_Disambiguation Disambiguation;
};
}
```

This techique allows one to combine classes which differ in type, but where it makes perfect sense to combine them in expressions without error:

```
#include "daixtrose/Daixt.h"

namespace MyOwn
{
template <class T> class Matrix {};
```

```
struct MatrixExpression {};
}

namespace Daixt
{
template <class T>
struct Disambiguator<MyOwn::Matrix<T> >
{
  static const bool is_specialized = true;
  typedef MyOwn::MatrixExpression Disambiguation;
};
}

int main()
{
  MyOwn::Matrix<double> M1;
  MyOwn::Matrix<int> M2;

  using namespace Daixt::DefaultOps;
  M1 + M2;
}
```

Performing an operation on some object may result in another type. Consider the multiplication of a matrix with a vector. The result is a vector expression. Daixtrose allows to communicate this fact via specializations of BinOpResultDisambiguator and UnOpResultDisambiguator:

```
#include "daixtrose/Daixt.h"

namespace MyOwn
{

struct MatrixExpression {};
struct VectorExpression {};

template <class T> class Matrix
{
public:
  typedef MatrixExpression Disambiguation;
};

template <class T> class Vector
{
public:
  typedef VectorExpression Disambiguation;
};

// define a new Daixtrose-compatible unary functions
DAIXT_DEFINE_UNOP_AND_FUNCTION(RowSum, RowSumOfMatrix, DoNothing)

} // namespace MyOwn


namespace Daixt
```

```
{
// Matrix * Vector = Vector
template <>
struct BinOpResultDisambiguator<MyOwn::MatrixExpression,
                                MyOwn::VectorExpression,
                                Daixt::DefaultOps::BinaryMultiply>
{
  typedef MyOwn::VectorExpression Disambiguation;
};

template <>
struct UnOpResultDisambiguator<MyOwn::MatrixExpression,
                               MyOwn::RowSumOfMatrix>
{
  typedef MyOwn::VectorExpression Disambiguation;
};

} // namespace Daixt

int main()
{
  MyOwn::Matrix<double> M;
  MyOwn::Vector<double> V1;
  MyOwn::Vector<int> V2;

  using namespace Daixt::DefaultOps;
  V1 + M * V2 - MyOwn::RowSum(M);
}
```

3.4.5 Value Semantics vs. Reference Semantics

Daixtrose default behaviour is to use value semantics while building the object that represents the expression tree. This means that all leaves (operands) get copied once. Users can modify this behaviour by specializing template `class CRefOrVal`:

Adding the following code to the example above prevents matrices and vectors to get copied.

```
namespace Daixt
{

template <class T>
struct CRefOrVal<MyOwn::Matrix<T> >
{
  typedef Daixt::ConstRef<MyOwn::Matrix<T> > Type;
};

template <class T>
struct CRefOrVal<MyOwn::Vector<T> >
{
  typedef Daixt::ConstRef<MyOwn::Vector<T> > Type;
};

}
```

Daixtrose now uses const references to the leaf objects. Of course the user then is responsible for avoiding dangling references by ensuring the lifetime of all leaves is longer than the lifetime of the expression.

3.4.6 Adding Polymorphic Behaviour to Expression Templates

It's a kind of magic, but it works well. `Daixt::Expr<T>` inherits from `FeaturesOfExpressions` using the Couriously Recursive Pattern (due to Geoffrey Furnish). Users can specialize this class and by this add publicly accessible member functions on the fly. This feature is build on top of a disambiguation changing mechanism which may help around some brain-damage in other situations, too.

But see yourself: a piece of code says more than further explanations:

```
#include "daixtrose/Daixt.h"

#include <iostream>


////////////////////////////////////////////////////////////////////////////
// A _very_ simple example for how to use Daixt.

struct ValueGetter;

class Variable
{
  double Value_;
public:
  Variable() : Value_(0.0) {}
  Variable(double Value) : Value_(Value) {}

  template <class T> Variable(const Daixt::Expr<T>& E);

  inline double GetValue() const { return Value_; }
  inline void SetValue(double Value) { Value_ = Value; }
};

////////////////////////////////////////////////////////////////////////////
// all overloads of a feature are gathered together in one place, the functor
// ValueGetter. You could have the same effect with free functions, too
struct ValueGetter {
  inline double operator()(const Variable& v) const
  {
    return v.GetValue();
  }

  template<class T>
  double operator()(const Daixt::Expr<T>& E) const
  {
    return (*this)(E.content());
  }
```

```
  ////////////////////////////////////////////////////////////////////////
  // always rely on the default behaviour

  template <class ARG, class OP>
  double
  operator()(const Daixt::UnOp<ARG, OP>& UO) const
  {
    return OP::Apply((*this)(UO.arg()),
                     Daixt::Hint<double>());
  }

  template <class LHS, class RHS, class OP>
  double
  operator()(const Daixt::BinOp<LHS, RHS, OP>& BO) const
  {
    return OP::Apply((*this)(BO.lhs()),
                     (*this)(BO.rhs()),
                     Daixt::Hint<double>());
  }
};


// mourning about the C++ rules ...
template <class T>  Variable::Variable(const Daixt::Expr<T>& E)
  :
  Value_(ValueGetter()(E)) {} // delegate to the functor

////////////////////////////////////////////////////////////////////////
// Introduce some features we would like to add sometimes to the expression

struct SomeFeature
{
  virtual double GetValue() const = 0;
  virtual SomeFeature* DeepCopy() const = 0;
  virtual ~SomeFeature() {}
};



namespace Daixt
{
template <class T>
class FeaturesOfExpression<SomeFeature, T>
  : public SomeFeature
{
public:
  double GetValue() const
  {
    return ValueGetter()(static_cast<const T&>(*this));
  }

  FeaturesOfExpression<SomeFeature, T>* DeepCopy() const
  {
```

```
    return new T(static_cast<const T&>(*this));
  }

  virtual ~FeaturesOfExpression() {};
};

} // namespace Daixt

////////////////////////////////////////////////////////////////////////////

int main()
{
  using namespace Daixt::DefaultOps;

  Variable a = 5.0, b = 3.0, c = 2.0;
  Variable d = a + b + c / (a - b);

  std::cerr << ValueGetter()(d) << std::endl;

  std::cerr << Daixt::ChangeDisambiguation<SomeFeature>
    (a + b + c / (a - b)).GetValue() << '\n';

  SomeFeature* sf =
    Daixt::ChangeDisambiguation<SomeFeature>(a + b + c / (a - b)).DeepCopy();

  std::cerr << sf->GetValue() << std::endl;
}
```

3.4.7 Compile Time Differentiation

If You ever want to let the compiler analytically differentiate arbitrary expressions, **Daixtrose** is the library of choice. This is not exactly what the compiler was invented for, but this feature is extremely useful e.g. for creating generic versions of exact Newton solvers.

There is one restriction: compile time differentiation only works for variables that differ in type. Of course one wants all variables to behave similar, so the nearest solution is to use a template class that makes the variables distinguishable *at compile time* through the template argument.

Some possible candidates are

```
template <std::size_t Number> class Variable;
```

or

```
template <const char Name[256]> class Variable;
```

The first version is used in `src/demos/simplify/TestSimplify.C`, the second version in the following demo example. Differentiation was implemented such that the product and sum rules are formally applied without any simplification. This means that zero terms are not automatically removed.

Example: `Diff(a + b, b)` yields `0 + 1` or - in C++ terms - an object of type `Expr<BinOp<...> >` instead of an `Expr<IsOne<..> >`.

For production code the compile time zero-branch deletion is crucial for the runtime performance. Therefore expression simplification is implemented separately and can be applied to the result of a differentiation. Example: `Simplify(Diff(a + b, b))` yields the desired result `1`.

In order to speed up compilation for cases where these features are not needed differentiation resides in `Daixt::Differentiation` and expression manipulation in `Daixt::ExprManip`.

The following example shows both features in action.

```
#include "daixtrose/Daixt.h"

template <const char Name[256]> class Variable {};

template <const char Name[256]>
inline std::ostream& operator<<(std::ostream& os, const Variable<Name>& v)
{
  return os << Name;
}

struct VariableDisambiguation {};
namespace Daixt
{
template <const char Name[256]>
struct Disambiguator<Variable<Name> >
{
  static const bool is_specialized = true;
  typedef VariableDisambiguation Disambiguation;
};
}


template <class T, class D>
void TestDiff(const T& t, const D& d)
{
  using namespace Daixt::OutputUtilities;
  using namespace Daixt::ExprManip;
  using namespace Daixt::Differentiation;

  std::cerr << "Diff(" << t << ", " << d << ")\n   = "
            <<  Diff(t, d) << "\n   = " << Simplify(Diff(t, d)) << std::endl;
}


//////////////////////////////////////////////////////////////////////
#define DECLARE_VARIABLE(VARNAME) \
extern const char NameOf##VARNAME[256] = #VARNAME ; \
Variable<NameOf##VARNAME> VARNAME


DECLARE_VARIABLE(a);
```

```
DECLARE_VARIABLE(b);
DECLARE_VARIABLE(c);
DECLARE_VARIABLE(d);


int main()
{
  using namespace Daixt::DefaultOps;

  TestDiff(RationalPow<3, 2>(d), d);
  TestDiff(a + b / (c - d), d);
  TestDiff(a/d, d);
}
```

3.5 Have a lot of fun

This is the end of the Quick Tour. Further Information can be gathered by looking at the files in directories `demos`, `tiny` and `linalg`.

Please send me some feedback via the mail addresses at `http://daixtrose.sourceforge.net/contact.html`

Enjoy!

4 Programmer's Reference

This chapter has not been written yet. I did not find the time to write a more detailed description of what **Daixtrose** can be used for and how it can be extended.