Archive of Numerical Software
vol. 5, no. 1, 2017, pages 169–192
DOI: 10.11588/ans.2017.1.27477

FunG - Automatic differentiation for invariant-based modeling

Lars Lubkoll

Received: January 31st, 2016; **final revision:** July 9th, 2016; **published:** March 6th, 2017.

Abstract: This document describes a C++ -library for the generation of invariant-based models, including first three derivatives. Using expression templates, features of C++11/14, forward automatic differentiation and modern SFINAE-techniques admits a highly efficient implementation with a simple and intuitive interface.

1 Introduction

Efficient and robust mathematical algorithms for the solution of real-world and scientific problems often require the computation of derivatives of complex functions. Since the manual computation of derivatives is time-consuming and error-prone, this problem is typically solved by software. Different strategies have been developed to solve this task. The most popular approach, finite differences, is known to cause significant problems due to truncation errors and the difficulty to control their accuracy for highly nonlinear problems.

For this reason several other approaches have been investigated and implemented. Besides finite differences, the most popular strategies for the computation of derivatives are symbolic and automatic differentiation (AD). The former computes the derivative of a function, which then can be evaluated, whereas the latter directly computes the evaluation of the derivative.

In C++ the standard implementation technique for both symbolic and automatic differentiation is to provide overloads for arithmetic operators and elementary functions in `cmath`, resp. in `math.h` such that their direct evaluation can be replaced by an object that carries all information necessary for the computation of derivatives. One distinguishes forward- or reverse-mode schemes. Forward-mode schemes implement the same procedure that one typically uses for the manual computation of derivatives. Starting from the whole function, differentiation rules are applied recursively until we reach the point where only simple expressions, for which the derivatives are known, must be processed. Reverse-mode schemes use a more involved strategy for the evaluation of derivatives and are not of interest here[1]. The interested reader is referred to eg. Griewank and Walther [2008].

An incomplete list of forward-mode schemes includes ADOLC Walther and Griewank [2012], though with main focus on reverse-mode schemes, Sacado Heroux et al. [2005], which is part of the Trilinos project, CoDiPack Sagebaum et al., a particularly efficient implementation of the scientific computing group of Technische Universität Kaiserslautern, ADEPT Hogan [2014],

[1] For the use-cases that are of interest here, reverse-mode schemes are significantly slower than forward schemes.

FADBAD++ Bendtsen and Stauning [1996] and CppAD Bell. Other approaches use symbolic differentiation to compute the derivative of a function. This strategy is followed in SEMT SEM, Gil and Gutterman [1998] and in Kourounis et al. In particular the latter seems to be interesting. Unfortunately the article was never published and the used code is not publicly available[2]. For further references the interested reader is referred to Kourounis et al. and the website `www.autodiff.org`.

Despite the large number of available AD-libraries FunG[3] provides some unique features. First it is not tied to a specific type of variable. Instead it provides a small and flexible interface that admits to work with general vector space elements. In particular FunG admits the computation of directional derivatives with respect to matrices and provides all functionality for invariant-based models, i.e. models that describe physical processes in terms of matrix invariants. This type of model is the basis of state of the art models for hyperelastic materials, cf. Lubkoll [2015] and the references therein. For these materials the mechanical properties are given in terms of a strain density function $W(F)$ that depends on the deformation gradient $F \in \mathbb{R}^{3,3}$. As an example we consider a model for the description of muscle tissue from Martins et al. [1998], given through

$$\begin{aligned} W(F) &= c\left[\exp(b(\bar{\iota}_1(C)-3))-1\right] + d\left[\exp(a(\bar{\iota}_6(C,M)-1)^2)-1\right], \\ \bar{\iota}_1(C) &= \operatorname{tr}(C)\det(C)^{-1/3}, \qquad C = F^TF, \\ \bar{\iota}_6(C,M) &= \operatorname{tr}(CM^2)\det(C)^{-1/3}, \end{aligned}$$

where $\bar{\iota}_1$ is called the first modified invariant and $\bar{\iota}_6$ the third modified mixed invariant. The latter is used to describe the anisotropic influence of the muscle fibers. The need to assemble $\int_\Omega W^{(n)}(F)\,dx$, $n = 1,\dots,3$ was the motivation for the development of FunG.

Another advantage is the runtime performance of the generated models, which is demonstrated in Sec. 5. This has several reasons. The treatment of multiple variables is different from other implementations and admits to avoid redundant computations. Furthermore, due to a compiler-friendly code design, the generation of efficient code is strongly simplified. Careful profiling and suitable optimizations yield an implementation that contains only small runtime overhead.

Eventually, all other investigated AD-implementations require the introduction of additional types. Mostly these are containers for scalar unknowns. Only FADBAD++ does not need these wrappers, though it requires to use a particular container for the implementation of the function for which we want to compute the unknown. In this respect FunG requires less setup code for using it. For the above introduced model of muscle tissue we can simply write:

C++ code

```
using namespace FunG;
// for given type Matrix
// initialize the structural tensor M
auto M = LinearAlgebra::unitMatrix<Matrix>();
// initialize the deformation gradient F
auto F = LinearAlgebra::unitMatrix<Matrix>();
// generate relationship between strain and deformation gradient
auto C = LinearAlgebra::strainTensor(F);
// generate the relationship between strain and energy density
auto W = c*( exp( b*( mi1(C) - 3 ) ) - 1 )
          + d*( exp( a*pow<2>( mi6(C,M) - 1 ) ) - 1 );
```

Here `mi1` and `mi6` denote the first modified invariant $\bar{\iota}_1(F)$, resp. the third modified mixed invariant $\bar{\iota}_6(F,M)$ and the function `strainTensor(F)` generates the strain tensor $C = F^TF$.

[2] A preprint of the paper is available at `www.researchgate.net/profile/Michael_Saunders2/publications`.

[3] The name FunG is both an abbreviation for Function Generation and a tribute to Yuan-Cheng Fung one of the founders of modern biomechanics.

While FunG has several unique features it also lacks some functionality provided by other AD-implementations. In particular it currently does not provide reverse-mode automatic differentiation. Due to its generic approach it also lacks full jacobian computation, which would require the introduction of new customized data structures, i.e. for higher order tensors. This strongly opposes one of the principal ideas behind FunG, namely providing an abstraction layer that still allows each user to use his own, possibly optimized, data structures[4].

To keep the presentation concise we will mainly restrict the discussion to scalar variables. Extensions for matrices are discussed where necessary. The implementation strictly follows the underlying mathematical structure. This is explained in Sec. 2. Then provided mathematical functions, such as common trigonometric functions and matrix invariants, are shortly introduced in Sec. 3. In Sec. 4 we will discuss different optimization strategies that are implemented in FunG, using two invariant-based models as examples. Then, in Sec. 5, the performance of FunG is compared with several automatic differentiation (AD) libraries. In Sec. 6 examples that illustrate several features of FunG are given. This paper ends with a short summary of the attained results (Sec. 7) and instructions for downloading and installing the library (Sec. 8).

2 Concepts

We begin by introducing the main concepts and techniques that are necessary for an efficient and generic implementation. For this first the interfaces for functions and variables are introduced. Then we discuss the heart of FunG, the implementations of elementary differentiation rules.

2.1 Functions in FunG

In FunG every nullary callable is treated as a constant function. Non-constant functions additionally provide a function `update` to change the point of evaluation $x \in X$, where X denotes some vector space. Optionally up to the first three directional derivatives may be specified. The interface for functions is summarized in the following list:

- Evaluate $f(x)$, where x has been assigned before:

C++ code

```
1 template <class Arg>
2 const auto& operator()() const;
```

- Set the point of evaluation x for general arguments (for non-constant functions):

C++ code

```
1 template <class Arg>
2 void update(const Arg& x);
```

- Evaluate the first directional derivative $f'(x)\delta x$ (for non-constant functions):

C++ code

```
1 template <class Arg>
2 auto d1(const Arg& dx) const;
```

- Evaluate the second directional derivative $f''(x)(\delta x, \delta y)$ (for non-linear functions):

C++ code

```
1 template <class ArgX, class ArgY>
2 auto d2(const ArgX& dx, const ArgY& dy) const;
```

[4] While not providing this functionality in a generic setting it is straightforward to write the corresponding assembly-loop using FunG.

- Evaluate the third directional derivative $f'''(x)(\delta x, \delta y, \delta z)$ (for non-linear, non-quadratic functions):

C++ code

```
template <class ArgX, class ArgY, class ArgZ>
auto d3(const ArgX& dx, const ArgY& dy, const ArgZ& dz) const;
```

Derivatives that are not implemented are interpreted as functions that always return zero. This design decision has significant advantages during code generation, see Sec. 4.2. However, it is not very convenient to use a function that may or may not provide certain member functions. Thus it is advisable to always use a *finalized version* of the generated function, i.e. instead of the generated function `f`, better use the result of `finalize(f)`. The call to `finalize` adds all undefined derivatives and, depending on the generated function, may perform some simplifications of the interface. This is explained in see Sec. 6. [5]

2.2 Function arguments in FunG

In the last section an interface for functions on vector spaces was presented. Therefore function arguments should at least satisfy the requirements of a vector space element, they should be scalable and sumable. For non-scalar variables additional operations should be supported, such as matrix-matrix- or matrix-vector-multiplication. As example consider the matrix-valued example in the introduction, where scalability, sumability and matrix-matrix-multiplication is required for the function arguments.

In the following subsections the required interfaces for the classical function arguments scalars, vectors and matrices are summarized. Other vector space elements are possible, but must be provided by the user together with corresponding functions.

2.2.1 General interface. The following requirements hold for all variables that are to be used with FunG:

- one of the following operations must be valid for variables `v,w` of type `V`:

C++ code

```
v += w;
V x = v + w;
```

If only the first alternative is available, then FunG will generate an implementation of the free summation operator.

- one of the following operations must be valid for `a` of arithmetic type `A` [6]:

C++ code

```
v *= a;
V w = a*v;
```

If only the first alternative is available, then FunG will generate an implementation of the free multiplication operator.

[5] Currently no derivatives or order higher than the third are implemented. This is due to two reasons. First, for solving optimization problems typically no more than the first two or three derivatives are required. Secondly, derivatives of arbitrary order require to additionally pass the order as template argument. Together with the increasing number of template and function arguments this leads to signatures that are sufficiently hard to read to justify the introduction of a proxy object to increase readability. This in turn leads to a less intuitive syntax.

[6] Here "arithmetic" is used in the sense that `std::is_arithmetic<A>::value == true`.

These operations are not only available for built-in arithmetic types, but at least one of the alternatives is implemented in most C++ matrix libraries, such as Eigen (Guennebaud and Jacob [2010]), Dune-ISTL (Blatt and Bastian [2007]) or Armadillo (Sanderson [2010]).

2.2.2 Extended interface. For matrices and vectors additional functionality is required. Since vectors are treated analogously, the discussion is restricted to matrices. For these the additional requirements are:

- access to entries,
- access to the number of rows and
- access to the number of columns.

For all cases there exist free template functions that provide access to the desired quantity via traits classes. For accessing matrix entries there is:

C++ code

```
template <class Matrix, class Index,
          class = std::enable_if_t<std::is_integral<Index>::value> >
decltype(auto) at( Matrix&& A, Index i, Index j )
{
  return AccessEntryOfMatrix<Matrix>::apply( A, i, j );
}
```

Currently, FunG supports the most popular ways of accessing matrix entries, via:

`A[i][j]` or `A(i,j)`.

Using the SFINAE[7]-technique of Sec. 2.4 it is easy to choose the correct implementation at compile time [8].

Regarding the access of the number of rows, resp. columns, matrices of constant size and of dynamic size are distinguished. For constant size matrices access to the number of rows and columns at compile time enables loop-unfolding optimizations for the C++-compiler. To exploit this fact, slightly different interfaces are provided for matrices of constant and dynamic size.

2.2.2.1 Matrices of constant size. For constant size matrices the following free template functions admit access to the number of rows resp. columns:

C++ code

```
template <class Matrix>
constexpr auto rows()
{
  return NumberOfRows<Matrix>::value;
}

template <class Matrix>
constexpr auto cols()
{
  return NumberOfColumns<Matrix>::value;
}
```

Matrices of constant size often take the numbers of rows and columns as template parameters. If template matrix types can be specified with one of the patterns

[7] Substitution Failure Is Not An Error.
[8] In case that both ways of accessing entries are supported, square brackets are used.

C++ code

```
using M1 = MyMatrix<nRows,nCols>
// or
using M2 = MyMatrix<Scalar,nRows,nCols>
```

then FunG will automatically detect the number of rows and columns. Else you need to provide a specialization of the template classes `NumberOfRows` and `NumberOfColumns`. To illustrate usage we consider the implementation of the scalar product $(A, B) := \sum_{i,j} A_{ij}B_{ij}$:

C++ code

```
template <class Matrix>
auto scalarProduct(const Matrix& A, const Matrix& B)
{
  using Index = decltype( rows<Matrix>() );

  auto result = decltype( at(A,0,0) )(0);
  for( Index i = 0 ; i < rows<Matrix>() ; ++i )
    for( Index j = 0 ; j < cols<Matrix>() ; ++j )
      result += at(A,i,j) * at(B,i,j);

  return result;
}
```

2.2.2.2 Matrices of dynamic size. For matrices of dynamic size the following free template functions admit access to the number of rows resp. columns:

C++ code

```
template <class Matrix>
auto rows(const Matrix& A)
{
  return DynamicNumberOfRows<Matrix>::apply( A );
}

template <class Matrix>
auto cols(const Matrix& A)
{
  return DynamicNumberOfColumns<Matrix>::apply( A );
}
```

Currently, there is support for

- access via public member variables `n_rows` and `n_cols` (as used in Armadillo),
- access via public member functions `rows()` and `cols()`.

For other signatures suitable specializations of the template classes `DynamicNumberOfRows` and `DynamicNumberOfColumns` must be provided. Again, we illustrate usage with the implementation of the scalar product $(A, B) := \sum_{i,j} A_{ij}B_{ij}$:

C++ code

```
template <class Matrix>
auto scalarProduct(const Matrix& A, const Matrix& B)
{
  assert( rows(A) == rows(B) && cols(A) == cols(B) );
  using Index = decltype( rows(A) );

  auto result = decltype( at(A,0,0) )(0);
  for( Index i = 0 ; i < rows(A) ; ++i )
    for( Index j = 0 ; j < cols(B) ; ++j )
      result += at(A,i,j) * at(B,i,j);

```



```
12   return result;
13 }
```

2.2.3 Usage of non-built-in arithmetic types. FunG admits the usage of user-defined arithmetic types. For this a suitable specialization of the template class `IsArithmetic` must be provided:

C++ code

```
1 template <>
2 struct IsArithmetic< UserDefinedScalarType >
3   : std::true_type
4 {};
```

2.2.4 Working with expression templates in matrix libraries. In some matrix-libraries, such as Eigen (cf. Guennebaud and Jacob [2010]), mathematical operations are implemented with expression templates that delay the actual computation until the next assignment. This may interfere with generic implementations that use automatic type deduction. To avoid these problems in FunG, a suitable specialization of the template class `Decay` must be provided. This is illustrated for the Eigen-library, where expression-templates provide the underlying matrix- or vector-type with the nested type `PlainObject`.

C++ code

```
1  // Default case (identity).
2  template < class F, class = void >
3  struct Decay
4  {
5    using type = F;
6  };
7
8  // Underlying type for expression templates of the Eigen library.
9  template < class F >
10 struct Decay< F, void_t< Checks::TryNestedType_PlainObject<F> > >
11 {
12   using type = typename F::PlainObject;
13 };
```

2.2.5 Multiple variables To work with multiple variables FunG provides a special template class to associate an id with a variable:

C++ code

```
1 template <class Type, int id>
2 class Variable;
```

This class is a function in the sense of 2.1. If derivatives are computed with respect to `id`, then this class behaves like a linear function, else like a constant function. This association of ids with variables at compile time has two advantages. First, it admits the generation of highly efficient code. This is explained in sec 4.2. Secondly, this approach does not require any additional storage for `id`.

We can use automatic type deduction to simplify the creation of a new variable:

C++ code

```
1 // create variable of type Variable<std::decay_t<decltype(x)>,0>
2 auto v0 = variable<0>(x)
```

For the template argument `Type` any type that satisfies the interface for variables is admissible, see 2.2. For problems that only depend on one variable the template class `Variable` is not required.

2.3 Mathematical operations

Having defined requirements on functions and their arguments we now turn to the question how to build up complex functions from elementary ones. Exemplarily, we consider

$$h(x) = \exp(\sin(x)) + \exp(x)\log(x) = (f_0 \circ f_1)(x) + f_0(x)f_2(x), \tag{1}$$

with $f_0(x) = \exp(x)$, $f_1(x) = \sin(x)$ and $f_2(x) = \log(x)$. The representation of h as combination of more elementary functions can be represented in a tree structure. This is illustrated in Fig. 1:

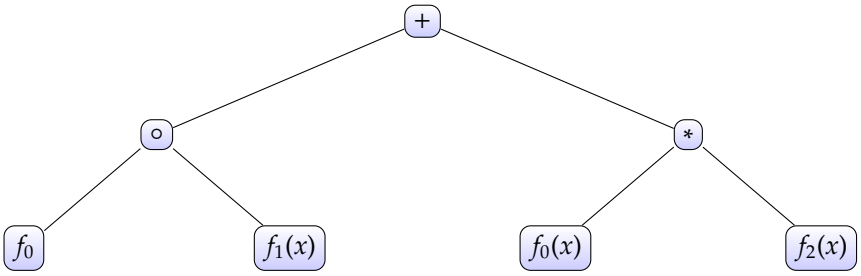

Figure 1: Tree structure associated with $h(x) = \exp(\sin(x)) + \exp(x)\log(x) = (f_0 \circ f_1)(x) + f_0(x)f_2(x)$.

The leafs are functions and all other nodes represent a mathematical operation. Then, the differentiation process can be described by a small set of operations that generate a new tree. These operations are equivalents of mathematical differentiation rules:

- the *sum rule*,
$$(f(x) + g(x))' = f'(x) + g'(x)$$
replaces the child nodes with its derivatives:

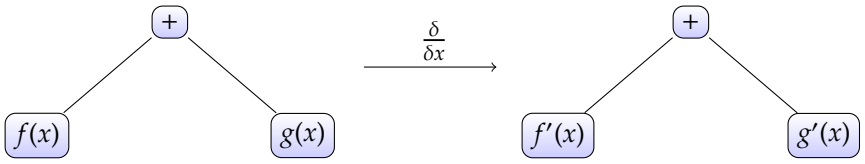

Figure 2: Tree transform associated with the sum rule.

- the *product rule*,
$$(f(x)g(x))' = f'(x)g(x) + f(x)g'(x)$$
generates a more complex tree:

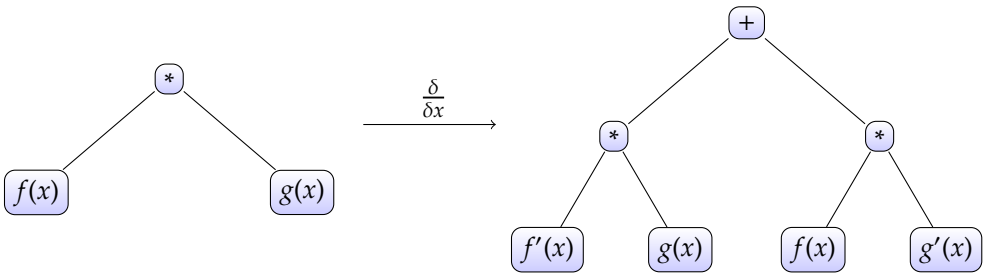

Figure 3: Tree transform associated with the product rule.

- and the *chain rule*,

$$f(g(x))' = f'(g(x))g'(x)$$

yields:

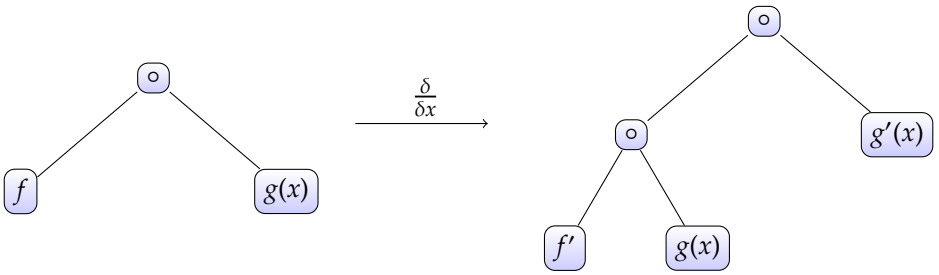

Figure 4: Tree transform associated with the chain rule.

With the given transformation rules it is easy to get the derivative of an arbitrary complex function by traversing its tree representation from top to bottom and repeatedly applying the differentiation rules. For the derivative of the function h, defined at the start of this subsection, we get:

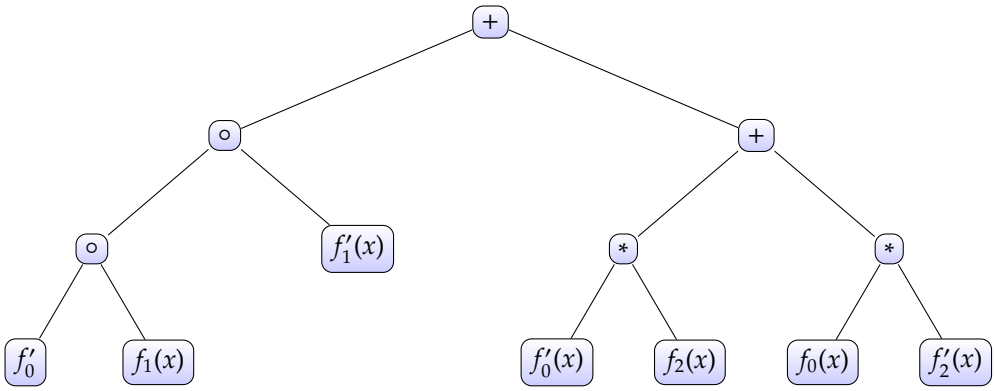

Figure 5: Example tree structure corresponding to $h'(x)$.

2.3.0.1 Expression templates. We can easily evaluate the derivative $h'(x)$ by traversing the generated tree. However, setting up such a tree structure and traversing it adds significant overhead that will slow down computations significantly. The classical technique for avoiding this overhead is based on expression-templates, cf. Veldhuizen [1995], and used in most AD-implementations, cf. eg. Hogan [2014], Walther and Griewank [2012], Sagebaum et al., Bendtsen and Stauning [1996], Heroux et al. [2005].

Expression-templates provide an abstraction layer that separates the interface for, i.e. mathematical, operations from its implementation. With this abstraction the evaluation of the tree structures can be performed at compile time.

In the context of FunG, simplified versions of the mathematical operators take the forms:

C++ code

```
1 template <class F, class G,
2           /* static concept checking */>
3 auto operator+(const F& f, const G& g) {
4   return MathematicalOperations::Sum<F,G>(f,g);
5 }
```

C++ code

```
template <class F, class G,
          /* static concept checking */>
auto operator*(const F& f, const G& g) {
  return MathematicalOperations::Product<F,G>(f,g);
}
```

The template classes `Sum` and `Product` hide the actual implementation of the mathematical operations and provide the corresponding differentiation rules. Additionally there exist implementations of `operator-` based on `operator+`.

For the chain rule an overload of the function call operator is required. This overload must be added to each function that is used within FunG (at least if the chain rule is used). This can be done by using the CRTP[9]-based class `Chainer`:

C++ code

```
template <class Function>
struct Chainer
{
  // provide function call operator
  decltype(auto) operator()() const noexcept
  {
    return static_cast<const Function*>(this)->d0();
  }

  // support chain rule
  template <class OtherFunction,
            /* static concept checks */>
  auto operator()(const OtherFunction& g)
  {
    return MathematicalOperations::Chain<Function,OtherFunction>
        (*static_cast<const Function*>(this),g);
  }

  ...
};
```

To avoid explicitly importing the overloads of `Chainer` to the derived class this class should no more define the function call operator. Instead it should define a function `d0` with same signature as the function call operator would have.

Assume that we already have defined a class `Cos` satisfying the function interface but not yet the chain rule. Then the easiest way to add support for the chain rule is to change it from

C++ code

```
class Cos {
  ...
  double operator() const noexcept;
  ...
};
```

to

C++ code

```
#include ''fung/util/chainer.hh''

class Cos : public Chainer<Cos> {
  ...
  double d0() const noexcept;
  ...
};
```

[9] Curiously Recurring Template Pattern.

As example we consider the function h given at the beginning of this section. In FunG we can write

C++ code

```
auto x = variable<0>(1.); // point of evaluation
auto fun = finalize( exp(sin(x)) + exp(x)*log(x) );
```

to generate an object of type

C++ code

```
Sum<
    Chain<Exp,Sin>,
    Product<Exp,Log>
    >
```

This class contains all relevant information to generate the derivatives of `fun`. Thus the tree structures described in the beginning of this section are generated and resolved at compile time. Since all type-information is provided to the compiler most function calls will be eliminated when resolving these trees. This approach allows to avoid almost all implementation-related runtime overhead.

2.4 SFINAE and void_t

In FunG SFINAE-techniques based on `void_t`, cf. Brown [2014], are heavily used. `void_t` is a template alias that is equivalent to `void` for all *valid* template arguments[10]:

C++ code

```
template <class...>
using void_t = void;
```

This alias matches any well-formed type into the predicable type `void`. In the context of FunG, this allows to verify interfaces at compile-time, provide a flexible interface to matrix and vector implementations as well as to remove redundant code paths from the evaluation of derivatives.

This is illustrated for the case of the first derivative in FunG, i.e. the case that we want to know if objects of type `T` provide a member function `d1`, taking one argument of type `Arg`. We can try to access this member function in an unevaluated context via

C++ code

```
template <class T, class Arg>
using TryMemFn_d1 =
  decltype( std::declval<T>().d1(std::declval<Arg>()) );
```

This alias can be used to define a meta-function as follows:

C++ code

```
// Default case, no suitable member function d1.
template <class T, class Arg, class=void>
struct HasMemFn_d1
  : std::false_type
{};

// Specialization for the case that a suitable member function d1 exists.
template <class T, class Arg>
struct HasMemFn_d1< T , Arg , void_t< TryMemFn_d1<T,Arg> > >
  : std::true_type
{};
```

[10] This implementation of `void_t` does not directly work with gcc-x, x<5. For these compiler versions another level of indirection must be added.

This meta-function can be used to choose different code paths at compile time. Alternatively one can use the same technique as in the specialization of `HasMemFn_d1` to directly use SFINAE for the choice of implementations.

2.5 Error handling

Debugging errors in strongly templated C++ code is often not easy due to the hard-to-read compiler output. Two ways of providing meaningful error messages are provided. Using the technique described in the last section, we can check satisfaction of the required interfaces and consistency of the defined function at compile time. Using `static_assert` meaningful compiler error messages are provided.

Sometimes things may go wrong because of the concatenation $f \circ g$ of two functions, where $\text{ran}(g) \not\subseteq \text{dom}(f)$. If f is implemented with the help of functions from `cmath`, then one can inspect the corresponding error flag. Alternatively one may define the macro `FUNG_ENABLE_EXCEPTIONS`. If defined, it enables an `OutOfDomainException` that is thrown as soon as a function argument leaves the admissible domain.

3 Provided functions

Mny functions that are contained in `cmath`, as well as matrix invariants and an implementation of the strain tensor are provided in FunG.

3.1 Functions from `cmath`

The functions in `cmath` are contained in namespace `std`. Though, in the C++11-standard it is allowed that the implementation just forwards to the `C` math library in `math.h`, wherein the functions and macros are part of the global namespace. This practice is followed in most `stl`-implementations. For this reason it is not possible to provide the same signature for built-in arithmetic types.

A common strategy to solve this problem in AD-implementations is the introduction of a special variable, such as `adouble` in ADOL-C. In FunG the template class `Variable` can be used for this purpose. For problems that only depend on one variable, it is also admissible to directly use custom types[11]. To avoid ambiguity for functions that are contained in `cmath` the first letter of the function name must be changed to uppercase:

C++ code

```
1 auto f = Sin(1.)                // uppercase to avoid ambiguity
2 auto g = sin( variable<0>(1.) ) // no ambiguity
```

An overview on the currently available scalar functions is given in Table 1.

3.2 Invariants

Since physical models should be independent of the position of the observer they are formulated in terms of invariants, such as $\det(A)$, $\text{tr}(A)$ or $\text{cof}(A)$. In FunG, optimized implementations of the determinant[12], the trace and the cofactors are provided for both, matrices of constant and dynamic size. Based on these, implementations of different invariants are provided. These are summarized in table 2.

The first and third invariant, the trace and determinant of a matrix $A \in \mathbb{R}^{n,n}$, can also be accessed via `trace(A)` resp. `det(A)`. Another important quantity in invariant-based modeling is the strain

[11] In this case the post-processing through `finalize` yields a simpler interface, cf. Sec. 6.
[12] The determinant is implemented for matrices $A \in \mathbb{R}^{n,n}$, $n = 2, 3$.

Table 1: Available scalar functions in FunG.

Formula	Code	Code (built-in arithmetic)
$x^{n/m}$	`pow<n,m>(x)`	`Pow<n,m>(x)`
$\sqrt{x}$	`sqrt(x)`	`Sqrt(x)`
$\sqrt[3]{x}$	`cbrt(x)`	`Cbrt(x)`
$\sqrt[3]{x^2}$	`cbrt2(x)`	`Cbrt2(x)`
2^x	`exp2(x)`	`Exp2(x)`
$\exp(x)$	`exp(x)`	`Exp(x)`
$\log(x)$	`log10(x)`	`Log10(x)`
$\log_2(x)$	`log2(x)`	`Log2(x)`
$\ln(x)$	`ln(x)`	`LN(x)`
$\sin(x)$	`sin(x)`	`Sin(x)`
$\cos(x)$	`cos(x)`	`Cos(x)`
$\tan(x)$	`tan(x)`	`Tan(x)`
$\arcsin(x)$	`asin(x)`	`ASin(x)`
$\arccos(x)$	`acos(x)`	`ACos(x)`

Table 2: Available invariants in FunG.

Invariant	Formula	Code
$\iota_1(A)$	$= \mathrm{tr}(A)$	`i1(A)`
$\iota_2(A)$	$= \mathrm{tr}(\mathrm{cof}(A))$	`i2(A)`
$\iota_3(A)$	$= \det(A)$	`i3(A)`
$\iota_4(A,M)$	$= \mathrm{tr}(AM)$	`i4(A,M)`
$\iota_5(A,M)$	$= \mathrm{tr}(AM^2)$	`i5(A,M)`
$\iota_6(A,M)$	$= \mathrm{tr}(A^2M)$	`i6(A,M)`
$\bar{\iota}_1(A)$	$= \mathrm{tr}(A)\det(A)^{-1/3}$	`mi1(A)`
$\bar{\iota}_2(A)$	$= \mathrm{tr}(\mathrm{cof}(A))\det(A)^{-2/3}$	`mi2(A)`
$\bar{\iota}_4(A,M)$	$= \mathrm{tr}(M,A)\det(A)^{-1/3}$	`mi4(A,M)`
$\bar{\iota}_5(A,M)$	$= \mathrm{tr}(AM^2)\det(A)^{-1/3}$	`mi5(A,M)`
$\bar{\iota}_6(A,M)$	$= \mathrm{tr}(A^2M)\det(A)^{-2/3}$	`mi6(A,M)`

tensor $C = F^T F$ for $F \in \mathbb{R}^{n,n}$. An efficient implementation, that exploits symmetry, can be accessed via:

C++ code

```
1 auto C = strainTensor(F);
```

4 Performance optimization

FunG was developed with focus on integrands that are evaluated in finite-element assembly procedures. Thus performance is important. In this section the three most relevant optimization strategies are shortly presented. Two examples from biomechanics are used to illustrate their effects. The first is a compressible version of the muscle model which has been shown in the

introduction:

$$\begin{aligned} W(F) &= c\left[\exp(b(\bar{\iota}_1(C)-3))-1\right] + d\left[\exp(a(\bar{\iota}_6(C,M)-1)^2)-1\right] + \\ &\quad e\det(C) + \frac{f}{2}\log(\det(C)), \\ \bar{\iota}_1 &= \operatorname{tr}(C)\det(C)^{-1/3}, \qquad C = F^TF, \\ \bar{\iota}_6 &= \operatorname{tr}(CM^2)\det(C)^{-1/3}, \end{aligned}$$

where a, b, c, d, e and f are material parameters. In FunG this can be implemented as follows:

C++ code

```
// for given MatrixType
// initialize the structural tensor M
auto M = LinearAlgebra::unitMatrix<Matrix>();
// initialize the deformation gradient F
auto F = LinearAlgebra::unitMatrix<Matrix>();

auto model = c*( exp( b*( mi1(F) - 3 ) ) - 1 )
           + d*( exp( a*pow<2>( mi6(F,M) - 1 ) ) - 1 )
           + e*det(F) + f/2*log(det(F));

// generate strain tensor
auto C = strainTensor(F);
// generate material model W(C)
auto W = model( C );
```

In the above model `mi1` and `mi6` denote the first modified invariant $\bar{\iota}_1(F)$, resp. the third modified mixed invariant $\bar{\iota}_6(F,M)$ and the function `strainTensor(F)` generates the strain tensor $C = F^TF$.

The second is a compressible model for adipose tissue from Sommer et al. [2013]:

$$\begin{aligned} W(F) &= c_{\text{Cells}}(\iota_1 - 3) + \frac{k_1}{k_2}\exp(k_2(\kappa\iota_1 + (1-3\kappa)*\iota_4)^2 - 1) + \\ &\quad e\det(C) + \frac{f}{2}\log(\det(C)), \\ \iota_1 &= \operatorname{tr}(C), \qquad C = F^TF, \\ \iota_4 &= \operatorname{tr}(CM), \end{aligned}$$

where c, k_1, k_2, κ are material parameters. Its implementation in FunG is given through

C++ code

```
// for given MatrixType
// initialize the structural tensor M
auto M = LinearAlgebra::unitMatrix<Matrix>();
// initialize the deformation gradient F
auto F = LinearAlgebra::unitMatrix<Matrix>();

auto model = c*( i1(F) - 3 )
           + d*( exp( a*pow<2>( mi6(F,M) - 1 ) ) - 1 )
           + e*det(F) + f/2*log(det(F));

// generate strain tensor
auto C = strainTensor(F);
// generate material model W(C)
auto W = model( C );
```

In the following all computations are performed on an ASUS UX32V, 4xi7-3517 1.90 GHz, 10 GiB DDR3-RAM 1600 MHz under Kubuntu 16.04. The code was compiled with gcc 5.3.1 and additional compiler flag `-std=c++1y`. Constant-size matrices of Eigen, version 3.2.0, see Guennebaud

and Jacob [2010] are used. For each described optimization technique the average computation time for the evaluation of single member functions will be given, where the average is computed over 10^7 evaluations at varying points of evaluation. No numbers are given for the evaluation of the function value. This is due to the fact that the related computations are fully performed in the `update`-function, see Sec. 4.1. Thus the function call operator only has to access the cached value. Recall that the performance of FunG is, to large parts, a consequence of compiler-friendly code design. Thus all measurements are strongly dependent on the used compiler.

4.1 Caching

The most efficient strategy to reduce the computation time in automatic differentiation is caching. This is mainly due to two reasons. First, caching intermediate results that are expensive to compute pays off. This does not only include caching of the results of iterative computations such as `sqrt(x)`, `sin(x)` or `exp(x)`, but also the direct evaluation of every function in its `update`-function. This avoids repeated evaluation in expressions such as

$$(f(x) * g(x) * h(x))' = f'(x)g(x)h(x) + f(x)g'(x)h(x) + f(x)g(x)h'(x),$$

where each function value is used twice. Secondly this admits to reduce the number, and often also the size, of temporary variables. Both effects help avoiding expensive cache misses. The influence on performance is illustrated in table 3.

	adipose tissue				muscle tissue			
caching	update	d1	d2	d3	update	d1	d2	d3
with	260 ns	44 ns	159 ns	490 ns	473 ns	62 ns	305 ns	1752 ns
without	270 ns	190 ns	714 ns	3090 ns	473 ns	353 ns	1673 ns	11907 ns
ratio	0.96	0.23	0.22	0.16	01.00	0.18	0.18	0.15

Table 3: Average computation time with and without caching.

4.2 Elimination of zeros

One of the disadvantages of automatic differentiation is the fact that for higher-order, possibly mixed, derivatives significant computation time can be spend with the computation of zeros. In most cases the compiler is not able to remove these redundant operations. Thus it pays off to implement a mechanism that eliminate these branches.

The essential idea is to not implement functions that always return zero (such as the second derivative of a linear function). Using the technique described in Sec. 2.4 it is straightforward to provide suitable optimizations for the mathematical operations described in Sec. 2.3 [13].

The impact on performance is illustrated in table 4. For the `update`- and the `d1`-function the observed differences are negligible. This is to be expected since no operation can be eliminated from the first and only few from the latter. For higher derivatives the elimination of zeros becomes more and more relevant.

[13] This design decision also enables efficient treatment of multiple variables and mixed derivatives.

	adipose tissue				muscle tissue			
zeros	update	d1	d2	d3	update	d1	d2	d3
with	260 ns	44 ns	159 ns	490 ns	473 ns	62 ns	305 ns	1752 ns
without	267 ns	42 ns	192 ns	835 ns	477 ns	60 ns	362 ns	2333 ns
ratio	0.97	1.05	0.83	0.59	0.99	1.03	0.84	0.75

Table 4: Average computation times with and without elimination of zeros.

4.3 Compiler flags

FunG largely consists of template classes, functions and aliases. Thus one of the prerequisites for the generation of efficient code is that the compiler must inline most of the function calls. In particular significantly larger amounts of code than in the "average" C++-code must be inlined. In invariant-based models, where also loops over matrix entries must be unfolded, significant performance increases can often be observed after adjusting the following compiler parameters:

- `early-inlining-insns=5000`
- `max-inline-insns-auto=3000`
- `inline-unit-growth=100`

For more details, see Stallman and the GCC Developer Community [2015], Alexandrescu [2014]. The influence of these adjustments is illustrated in table 5.

	adipose tissue				muscle tissue			
flags	update	d1	d2	d3	update	d1	d2	d3
with	210 ns	3 ns	75 ns	166 ns	400 ns	3 ns	246 ns	1369 ns
without	260 ns	44 ns	159 ns	490 ns	473 ns	62 ns	305 ns	1752 ns
ratio	0.81	0.07	0.47	0.34	0.85	0.05	0.81	0.78

Table 5: Average computation times with and without additional compiler flags.

Note that there is no golden rule for choosing compiler parameters. Its effects strongly differ between compilers and compiler versions. Moreover, when compiled together with an application, where FunG typically only is a small part, other parts of the code may influence the way the compiler optimizes the code. The latter effect can be avoided by compiling function definitions separately.

5 Comparison with AD-libraries

To give a first idea of the performance of FunG it is compared with several AD-libraries. For this, we consider the forward mode AD-schemes of

- ADOLC, version 2.6.1, see Walther and Griewank [2012],
- Adept, version 1.1, see Hogan [2014],
- CoDiPack, version 1.2.1, see Sagebaum et al.,

- CppAD, github, master branch from 01/23/2016, see Bell,
- FADBAD++, version 2.1, see Bendtsen and Stauning [1996] and
- Sacado, version 12.4.2 of Trilinos, see Heroux et al. [2005].

Again the computations are performed on an ASUS UX32V, 4xi7-3517 1.90 GHz, 10 GiB DDR3-RAM 1600 MHz under Kubuntu 16.04. The code was compiled with gcc 5.3.1 and additional compiler flag `-std=c++1y`.

Since not all investigated libraries support higher derivatives only the first derivative is considered. Moreover, the following examples are restricted to scalar variables. The reason is that matrix-valued variables are not directly supported by the other libraries [14].

In tables 6-8 computation times for different functions are compared with the best manual implementation that the author was able to provide for both, the function value and the first derivative. Ratios are given with respect to this manual implementation. Since not all libraries support the separate evaluation of function value and derivative the average evaluation time of both quantities is measured over 10^7 evaluations with different function arguments.

- The first example is
$$f_0(x) = x^{3/2} + \sin(\sqrt{x}).$$
This function can also be written as $f_0 = g \circ h$, where $g(x) = x^3 + \sin(x)$ and $h(x) = \sqrt{x}$. The latter implementation only requires one evaluation of $\sqrt{x}$. In table 6 the corresponding average computation time is given under *FunG (opt)*. From the same table we see that a straight-forward implementation of f_0 in FunG is about 10% slower, since $\sqrt{x}$ must be evaluated twice.

Manual	FunG (opt)	FunG	CoDiPack	Sacado	FADBAD	Adept	ADOLC	CppAD
60 ns	60 ns	65 ns	65 ns	68 ns	222 ns	304 ns	401 ns	926 ns
1.00	1.00	1.08	1.08	1.13	3.70	5.07	6.68	15.4

Table 6: Comparison of the average computation time for the evaluation of function value and derivative for $f_0(x) = x^{3/2} + \sin(\sqrt{x})$. Ratios are given with respect to the manual implementation.

As illustrated in table 6, FunG achieves the same performance as the optimized manual implementation. Without the above introduced reformulation of f_0 FunG provides the same performance as CoDiPack. Sacado is slighly slower in this example. Despite the fact that f_0 is a very simple function the remaining libraries, FADBAD++, Adept, ADOLC and CppAD are already significantly slower. These differences will be even more pronounced in the following examples.

- The next example is
$$f_1(x) = 1 + x(1 + x(1 + x(1 + x)))$$
$$\left(= \sum_{i=0}^{4} x^i \right)$$
This example only consists of simple and cheap operations. In particular nothing has to be computed iteratively. As illustrated in table 7, both FunG and CoDiPack achieve the same performance as the optimized manual implementation. All other implementations are significantly slower.

[14] In case that vector-valued variables are supported one could work with vectorized matrices.

Manual	FunG	CoDiPack	Sacado	FADBAD	Adept	ADOLC	CppAD
1.7 ns	1.7 ns	1.7 ns	4.0 ns	13.9 ns	45.8 ns	336 ns	558 ns
1.00	1.00	1.00	2.35	8.18	26.9	198	328

Table 7: Comparison of the average computation time for the evaluation of function value and derivative for $f_1(x) = 1 + x(1 + x(1 + x(1 + x)))$. Ratios are given with respect to the manual implementation.

- The last example is given through

$$f_2(x, y, z) = xyz.$$

This demonstrates the effect of multiple variables[15].

Manual	FunG	CoDiPack	FADBAD	Sacado	Adept	ADOLC	CppAD
1.7 ns	1.7 ns	2.0 ns	8.1 ns	40 ns	48 ns	877 ns	1781 ns
1.00	1.00	1.18	4.76	23.5	28.2	515	1047

Table 8: Comparison of the average computation time for the evaluation of function value and derivative for $f_2(x, y, z) = xyz$. Ratios are given with respect to the manual implementation.

As illustrated in table 8 FunG achieves the same performance as the optimized manual implementation. Only the forward-scheme of CoDiPack, which is 18% slower, provides roughly comparable performance. The large differences with respect to the other examined libraries is partly a consequence of the elimination strategy of Sec. 4.2.

6 Examples

We provide some examples that illustrate different features and how to use FunG. In Sec. 6.1 a model for adipose tissue is described. It illustrates the generation of invariant-based models without usage of the template class `Variable`. In Sec. 6.2 a model with different variables of different types is generated. Eventually, a model of nonlinear heat transfer is described. Such models depend on the state and the gradient of the heat distribution. How this is realized in FunG is demonstrated Sec. 6.3.

6.1 A model for adipose tissue

We begin with the model for adipose tissue of Sommer et al. [2013], which was already used in Sec. 4. It is assumed that the number of rows/columns of the used matrices can be deduced at compile time. Here $c_{Cells}, k_1, k_2, \kappa$ denote material parameters, F is the deformation gradient and M is a structural tensor describing the fiber directions of the interlobular septa [16]. The functions `i1` and `i4` implement the first invariant $\iota_1(F) = \mathrm{tr}(F)$, resp. the first mixed invariant $\iota_4(F, M) = \mathrm{tr}(FM)$.

[15] The compared libraries provide different ways of computing derivatives in vector mode. Adept computes the adjoint with respect to the Euclidean scalar product, CppAD computes the gradient and all others directly compute the directional derivatives. Since all three approaches require the computation of the same values, the author considers these slightly differing approaches worthy to compare.

[16] In adipose (fat) tissue anisotropic properties are mainly attributed to the interlobular septa, a network of long fibrous collagen bundles, see. Sommer et al. [2013].

C++ code

```
template <class Matrix>
auto adiposeTissueModel(double cCells, double k1, double k2,
                        double kappa, const Matrix& M, const Matrix& F,
                        int n = dim<Matrix>()) {
  using namespace FunG;
  using namespace FunG::LinearAlgebra;

  auto aniso = kappa*i1(F) + ( 1 - 3*kappa ) * i4(F,M) - 1;
  auto materialLaw = cCells*( i1(F) - n ) + (k1/k2) * ( exp(k2*pow<2>(aniso)) - 1 );

  return finalize( materialLaw( strainTensor(F) ) );
}

auto f = adiposeTissueModel(cCells,k1,k2,kappa,M,F);
```

Observe that no specific variable is associated with the unknown F. In this case the call to `finalize` yields a simplified interface. For some deformation F, the point of evaluation can be changed via:

C++ code

```
f.update(F);
```

Access to function value and derivatives, with perturbations $dF0, dF1, dF2$, is given through:

C++ code

```
auto value           = f();
auto firstDerivative  = f.d1(dF0);
auto secondDerivative = f.d2(dF0,dF1);
auto thirdDerivative  = f.d3(dF0,dF1,dF2);
```

Avoidance of the template class `Variable` enables another function that may be convenient. Namely a call to `finalize` adds the additional member function:

C++ code

```
template < class Arg >
auto operator()( Arg&& x ) {
  update( std::forward<Arg>( x ) );
  return operator()();
}
```

Thus it is also possible to write

C++ code

```
auto value = f(F);
```

instead of

C++ code

```
f.update(F);
auto value = f();
```

6.2 An example with two variables

In this example we consider a function that takes a scalar and a matrix-valued argument:

$$f(x, M) = \sqrt{x} * \mathrm{tr}(M).$$

To generate f we use the following code:

C++ code

```
#include ''fung.hh''

template <class Matrix>
auto myFunction(double initialX, const Matrix& initialM) {
  using namespace FunG;
  using FunG::LinearAlgebra::trace;

  auto x = variable<0>(initialX);
  auto M = variable<1>(initialM);

  return finalize( sqrt(x)*trace(M) );
}
...
f = myFunction(1.,M0);
```

Then, for scalar variables x, $dx1$, $dx2$ and matrix-valued variables M, dM we can change the point of evaluation via:

C++ code

```
f.template update<0>(x);
f.template update<1>(M);
```

Access to the function value and its derivatives is given through:

C++ code

```
double value       = f();
double df_dx       = f.template d1<0>(dx1);
double df_dM       = f.template d1<1>(dM);
double ddf_dMdx    = f.template d2<1,0>(fM,dx1);
double dddf_dxdxdM = f.template d3<0,0,1>(dx1,dx2,dM);
```

6.3 A model for nonlinear heat transfer

Nonlinear partial differential equations require another feature, the possibility to use different types associated to the same variable, such as representations of the value and gradient of a variable. A popular example for these problems are models of nonlinear heat transfer such as the following:

$$A(u, \nabla u) = (1 + |u|^2)\nabla u$$

When computing $A'(u, \nabla u)\delta u$, then both u and ∇u must be taken into account. For this we need to assign the same variable id to both value and gradient:

C++ code

```
#include ''fung.hh''

template <class Scalar, class Vector>
auto heatModel(const Scalar& u0, const Vector& du0) {
  using namespace Fung;

  auto u = variable<0>(u0);   // state variable
  auto du = variable<0>(du0); // gradient variable, both with id 0

  return finalize( ( 1 + pow<2>(u) )*du );
}

auto f = heatModel(u0,du0);
```

To work with this kind of model arguments must be packed into tuples. Then, for a variable u with gradient du we can change the point of evaluation via:

C++ code

```
1 f.template update<0>( std::make_tuple(u,du) );
```

Similarly, for a perturbation v with gradient dv, we get:

C++ code

```
1 double value = f();
2 double df_dx = f.template d1<0>( std::make_tuple(v,dv) );
3 ...
```

Note that different types for v and dv are required. Thus, this does not directly work for higher order ODE problems[17].

7 Conclusion

In this paper a library for the generation of invariant-based models and complex mathematical functions was presented. Exploiting features of `C++11/14` a highly generic and still simple implementation is provided, admitting the computation of derivatives with respect general vector space elements, such as scalar, vector- or matrix-valued variables.

As building blocks for more complex functions FunG contains several mathematical functions from `cmath` as well as a large set of principal, mixed and modified matrix invariants. Moreover an arbitrary number – within the limits of the system architecture – of unknowns of different types can be used.

With the help of automatic type deduction complicated template-techniques are not exposed to users. This enables FunG to provide a small, intuitive and easily extensible interface. The optimization strategies of Sec. 4 as well as strict application of modern `C++`-techniques and SOLID principles, cf. Martin [2009], admit the generation of highly efficient code. This was demonstrated by the examples in Sec. 5.

[17] For the next version, there is an extension planned that admits to directly use gradient variables of the same type as the state.

8 Download and installation

Both, the current stable version (1.3.2) and the master branch can be accessed from

http://lubkoll.github.io/FunG or https://github.com/lubkoll/FunG.

Aiming at high quality, re-usable code, FunG was developed using continuous integration[18] and extensive unit testing with the Google C++ test framework, cf. Sen [2010][19].

FunG is licensed under the GNU General Public License v3.0 (GPL v3). For compilation a C++-compiler with support for the following features of C++14 are required:

- `decltype(auto)` and
- the type traits aliases with trailing "`_t`".

For installation go to the root directory of FunG and issue the following commands[20]:

- `mkdir build`
- `cd build`
- `cmake ..`
- `make install`

To compile and run the unit tests issue the following commands in the build directory:

- `make`
- `ctest (--verbose)`

[18] https://travis-ci.org/lubkoll/FunG
[19] For test coverage see https://coveralls.io/github/lubkoll/FunG.
[20] Since FunG is header-only the `make install` command only copies the header files to the installation directory.

References

SEMT. https://github.com/st-gille/semt.

A. Alexandrescu. Optimization Tips - Mo' Hustle Mo' Problems. https://www.youtube.com/watch?v=Qq_WaiwzOtI (CppCon 2014), 2014.

B. M. Bell. CppAD - A Package for Differentiation of C++ Algorithms. http://www.coin-or.org/CppAD.

C. Bendtsen and O. Stauning. FADBAD, a flexible C++ package for automatic differentiation. Technical Report 1996-x5-94, Technical University of Denmark, 1996.

M. Blatt and P. Bastian. The Iterative Solver Template Library. *Appl. Par. Comp.*, 4699:666–675, 2007.

W. Brown. A SFINAE-Friendly std::common_type. Technical report, ISO/IEC JTC1/SC22/WG21, 2014.

J. Gil and Z. Gutterman. Compile time symbolic derivation with C++ templates. In *COOTS'98 Proceedings of the 4th conference on USENIX Conference on Object-Oriented Technologies and Systems - Volume 4*, 1998.

A. Griewank and A. Walther. *Evaluating Derivatives, Principles and Techniques of Algorithmic Differentiation*. SIAM, 2008.

G. Guennebaud and B. Jacob. Eigen v3. http://eigen.tuxfamily.org, 2010.

M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley. An overview of the Trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, 2005.

R. J. Hogan. Fast reverse-mode automatic differentiation using expression templates in C++. *ACM Trans. Math. Softw.*, 40(26):1–16, 2014.

D. Kourounis, L. N. Gergidis, and M. A. Saunders. Compile-Time Symbolic Differentiation Using C++ Expression Templates.

L. Lubkoll. *An Optimal Control Approach to Implant Shape Design: Modeling, Analysis and Numerics.* PhD thesis, University of Bayreuth, 2015.

R. C. Martin. *Clean Code. A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2009.

J. A. C. Martins, E. B. Pires, R. Salvado, and P. B. Dinis. A numerical model of passive and active behavior of skeletal muscles. *Comp. Meth. Appl. Mech. Eng.*, 151:419–433, 1998.

M. Sagebaum, T. Albring, and N. Gauger. CoDiPack - Code Differentiation Package. http://www.scicomp.uni-kl.de/software/codi/.

C. Sanderson. Armadillo: An Open Source C++ Linear Algebra Library for Fast Prototyping and Computational Intensive Experiments. Technical report, NICTA, 2010.

A. Sen. A quick introduction to the Google C++ Testing Framework. Technical report, IBM developerWorks, 2010.

G. Sommer, M. Eder, L. Kovacs, H. Pathak, L. Bonitz, C. Mueller, P. Regitnig, and G. A. Holzapfel. Multiaxial mechanical properties and constitutive modeling of human adipose tissue: A basis for preoperative simulations in plastic and reconstructive surgery. *Acta Biomater.*, 9:9036–9048, 2013.

R. Stallman and the GCC Developer Community. *Using the GNU Compiler Collection*. GNU Press (Free Software Foundation), 2015.

T. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, 1995.

A. Walther and A. Griewank. Getting started with ADOL-C. *Comb. Sci. Comp.*, pages 181–202, 2012.

