Extending TMB and ADMB functionality jointly

Kasper Kristensen & Anders Nielsen

September 30, 2016

Contents

1 Introduction

The ADMB-foundation is expanding its purpose to keep AD-tools available to build complex non-linear models (not only AD Model Builder). Template Model Builder (TMB) is a new and promising tool, which has been demonstrated to produce highly efficient code for random effects models (Kristensen et. al. 2015). A different — but equally important — aspect of any AD aided model building tool is how feasible it is to extend it. This report will investigate a few relevant extensions to both tools and evaluate how feasible these extensions are.

A list of extension were initially formulated to illustrate the type of extensions considered. The list was:

- Implement lacking special functions pbeta, qbeta, besselK ... and derivatives.
- Implement automatic generator of adaptive p-special functions (e.g. pbeta) by means of Gauss-Konrad quadrature (like **R**'s integrate). The user should implement the density, then TMB/ADMB should autogenerate adaptive adjoint code based on the assumption that any quadrature is something like sum(w*f(x)).
- Implement automatic generation of adaptive q-special functions (e.g qbeta). Again the user specifies f(x,theta), then ADMB/TMB solves f(x,theta)=0 via a Newton method, and the adjoint code is autogenerated either by implicit function theorem, or via the method in (B.M Bell and J.V. Burke 2008).
- Implement exact (AD) outer hessian.
- Implement Dave's suggestion w.r.t. importance sampling. As an alternative to N importance samples the suggestion is to repeat the entire optimization M times and calculate average. It is not know how well it works, so part of the study is to investigate this.
- Implement forward sub-sweep. The idea is to mark the nodes in the computational graph, which does not have to be updated when the function is evaluated (e.g. for the inner problem where the parameters are kept fixed). For some models this is estimated to give a factor of 5 speedup.
- Expand the template distributions (in ADMB) to be similar to the TMB density namespace.

This project investigates how feasible these extensions are in either tool, and illustrates it by actually implementing two or three of the extensions to compare code the complexity.

2 Differences between ADMB and TMB

In order to understand the following paragraphs one must be aware of some fundamental design differences between ADMB and TMB. The first major difference is that TMB uses a *fixed computational graph* for AD whereas ADMB allows the computational graph to change between function evaluations i.e. uses a *dynamical computational graph*. There are PROs and CONs of both approaches:

- The ADMB approach allows the user to include if/else statements in the objective function even if the branching is parameter dependent. This is not allowed in TMB. In this respect ADMB is thus more flexible than TMB.
- The TMB approach avoids a substantial amount of dynamical allocation and repeated calculations by using a fixed graph. This is probably one of the main causes of the observed speed difference between TMB and ADMB.
- The TMB approach is predictable in terms of memory usage from an early stage of program execution (until the computational graph has been allocated). This is not the case for ADMB.

From a user perspective it is important to be aware of these difference when deciding whether its worth porting an existing model from ADMB to TMB. Much existing code rely on parameter dependent branching (e.g. code using 'mfexp' in ADMB). When developing extensions to the tools its not always possible to carry over an ADMB solution to TMB. This is for instance the case for adaptive algorithms that need to run a variable number of iterations until a convergence criterion is met.

3 Adding simple functions

3.1 Adding a simple function to ADMB

Here we will quickly run through the steps of adding a simple function to ADMB.

If we find ourselves lagging a simple function in ADMB it can be declared as part of the template file where the model is defined. This gives access to the function within the that model, but (naturally) not for other models in separate template files. As an example consider a logistic regression model. To implement this we need the inverse logit function, which is defined as $f(x) = \frac{1}{1+\exp(-x)}$. This function is currently not build into ADMB. Declaring this function within the model template can be done as:

```
GLOBALS_SECTION
  #include <df1b2fun.h>
  dvariable invlogit(dvariable x){
    return 1/(1+exp(-x));
  }

DATA_SECTION
  init_int nobs
  init_vector x(1,nobs);
  init_vector y(1,nobs);

PARAMETER_SECTION
  init_number alpha
  init_number beta
  vector p(1,nobs)
  objective_function_value nll

PROCEDURE_SECTION
  nll=0;
  for(int i=1; i<=nobs; ++i){
    p(i) = invlogit(alpha+beta*x(i));
    nll+= -log(p(i))*y(i)-log(1-p(i))*(1-y(i));
  }
```

admbex/lreg.tpl

Setting up a self contained example like this, where the function is added to the template, or a in a separate file included in the template is a useful way to test the function. To include such a simple function into ADMB requires the following steps.

1. Add the source code file to the ADMB source code in the sub-folder `src/linad99` in this case we added a file called `invlogit.cpp`:

```
/*
 * $Id$
 *
 * Authors: Anders Nielsen <anders@nielsensweb.org>
 */
/**
 * \file
 * Inverse logit function.
 */
#include <fvar.hpp>

/**
 * Inverse logit function
 */
dvariable invlogit(dvariable x){
  return 1/(1+exp(-x));
}
```

admb/src/linad99/invlogit.cpp

2. The function header should be added to the file `src/linad99/fvar.hpp` in this case we added the line:

```
dvariable invlogit(dvariable x);
```

3. Finally we need to add the resulting object file to the list in the file `src/linad99/objects.lst`. Here we added the line:

```
invlogit.obj\
```

near similar functions.

After this we can recompile ADMB, and the function is now part of the recompiled version, and the above example works without including the `GLOBALS_SECTION`.

The above steps are sufficient to include a function to be used with purely fixed effects models in ADMB. If a function must be used with random effects also, then the steps must be repeated in the sub-folder `src/df1b2-separable` Briefly:

1. The source file must be added, but the function must be defined using `df1b2variables` instead of `dvariables`.
2. The function header must be added to the file `src/df1b2-separable/df1b2fun.h`.
3. The name of the resulting object file must be added to the list in the file `src/df1b2-separable/objects.lst`.

```
invlogit.obj\
```

3.2 Adding a simple function to TMB

A similar function has been added to the TMB distribution (thanks to Mollie Brooks) by adding the lines

```
template<class Type>
Type invlogit(Type x){
  return Type(1.0) / (Type(1.0) + exp(-x));
}
VECTORIZE1_t(invlogit)
```

to the file `TMB/inst/include/convenience.hpp`. The typename 'Type' refers to any scalar type and makes the function work for both double and nested AD double types. This means that the function works to any order as is. The final line extends the function to also work for vectors.
After the change the TMB package is re-installed by

```
cd adcomp
make install
```

4 Implementing special functions - current approaches

Special functions are crucial in statistics. Implementation of such functions can be a challenge on its own and even more so when combined with automatic differentiation. There are two fundamentally different ways to add a special function to an AD framework.

1 Differentiate the numerical approximation of the special function using AD or hand-coded adjoint code. This approach is generally adopted by ADMB.

2 Derive a differentiation rule for the special function in terms of existing functions in the AD framework *or* the function itself. Now the special function can be added as an 'atomic' function. This is the approach currently used in TMB.

Both approaches have strenghts and weaknesses.

Method 1 There is no guarantee that the derivative of a function approximation is a good representation of the function's derivative. As an example of what could go wrong, consider the following numerical scheme: For a fine grid of x-values x_i the function values $y_i = f(x_i)$ are pre-computed and a quadratic spline is fitted through the points (x_i, y_i). While this approach works excellent for the function value it is very bad for the function derivatives. The 3rd order derivative is everywhere zero.
The numerical scheme must fulfill convergence properties not only for the function value but also for the derivatives. Good numerical schemes are often adaptive which can lead to unpredictable memory usage if applying reverse mode AD directly on the scheme (this is only relevant for ADMB as TMB is unable to use adaptive algorithms as previously mentioned).

Method 2 The main problem with this method is that sometimes there is no simple expression for the derivative of a the given special function. The derivatives might introduce new special functions for which no known numerical evaluation scheme exists. In those cases one would have to develop ways to evaluate the new special functions accurately and efficiently which can be very challenging.

In the following sections we will demonstrate how the two methods can be used in ADMB and TMB to implement some selected special functions.

4.1 TMB pbeta implementation - a first attempt

Example 1: pbeta The (incomplete) CDF of the beta distribution is not part of TMB. It is given by:

$$f(y,\alpha,\beta) = \int_0^y x^{\alpha-1}(1-x)^{\beta-1}\,dx$$

The partial derivative wrt. y is

$$\partial_y f(y,\alpha,\beta) = y^{\alpha-1}(1-y)^{\beta-1}$$

We see that this derivative is expressed in terms of standard functions. However, derivatives wrt. the parameters introduce new special functions:

$$\partial_\alpha f(y,\alpha,\beta) = \int_0^y x^{\alpha-1}(\log x)(1-x)^{\beta-1}\,dx$$

The family of incomplete beta integrals is obviously not closed under differentiation. In order to obtain a closed family we define a generalized function that includes all higher order partial derivatives:

$$f(y,\alpha,\beta,m,n) := \int_0^y x^{\alpha-1}(\log(x))^m(1-x)^{\beta-1}(\log(1-x))^n\,dx \qquad (1)$$

This "symbol" is closed under differentiation:

$$\partial_y f(y,\alpha,\beta,m,n) = x^{\alpha-1}(\log(x))^m(1-x)^{\beta-1}(\log(1-x))^n$$

$$\partial_\alpha f(y,\alpha,\beta,m,n) = f(y,\alpha,\beta,m+1,n)$$

$$\partial_\beta f(y,\alpha,\beta,m,n) = f(y,\alpha,\beta,m,n+1)$$

The function (1) can thus be added to the derivatives table as a native symbol. The special case $m = n = 0$ gives the incomplete beta CDF.

```
namespace atomic {

  void integrand_D_incpl_beta(double *x, int nx, void *ex){
    double* parms = (double*)ex;
    double  alpha = parms[0];
    double  beta  = parms[1];
    int     m     = parms[2];
    int     n     = parms[3];
    for(int i=0; i<nx; i++){
      x[i] =
     pow(x[i], alpha - 1.0)      *
     pow(log(x[i]), m)           *
     pow(1.0 - x[i], beta - 1.0) *
     pow(log(1.0 - x[i]), n);
    }
  }
  /* n'th order derivative of (scaled) incomplete gamma wrt. shape
     parameter */
  double D_incpl_beta(double x,
                      double alpha, double beta,
                      double m,     double n){
    /* Control tolerance of R's integrate */
    double epsabs = 1e-10, epsrel = 1e-10, abserr=1e4;
    int neval = 1e4, ier = 0, limit = 1e2, last = 0;
    int lenw = 4 * limit;
    /* Workspace */
```

```
    int*   iwork = (int*)    malloc(limit * sizeof(int));
    double* work = (double*) malloc(lenw  * sizeof(double));
    /* Parameters */
    double ex[4];
    ex[0] = alpha;
    ex[1] = beta;
    ex[2] = m;
    ex[3] = n;
    double result = 0;
    double a = 0.0; /* Lower integration bound */
    double b = x;   /* Upper integration bound */
    Rmath::Rdqags(integrand_D_incpl_beta, ex, &a, &b,
            &epsabs, &epsrel, &result, &abserr,
            &neval, &ier, &limit, &lenw, &last,
            iwork, work);
    if(ier != 0){
#ifndef _OPENMP
      warning("incpl_beta (def) integrate unreliable: "
           "x=%f alpha=%f beta=%f n=%f m=%f ier=%i",
           x,   alpha,   beta,   n,   m,   ier);
#endif
    }
    free(iwork);
    free(work);
    return result;
  }

  /* Register native symbol with derivatives  */
  TMB_ATOMIC_VECTOR_FUNCTION(
                    // ATOMIC_NAME
                    D_incpl_beta
                    ,
                    // OUTPUT_DIM
                    1
                    ,
                    // ATOMIC_DOUBLE
                    ty[0] = D_incpl_beta(tx[0],
                                tx[1],
                                tx[2],
                                tx[3],
                                tx[4]);
                    ,
                    // ATOMIC_REVERSE
                    Type x =     tx[0];
                    Type alpha = tx[1];
                    Type beta  = tx[2];
                    Type m     = tx[3];
                    Type n     = tx[4];
                    Type one   = 1.0;
                    px[0] =
                    pow(x, alpha - one) *
                    pow(log(x), m) *
                    pow(one - x, beta - one) *
                    pow(log(one - x), n) *
                    py[0];                  // Partial wrt. x
                    CppAD::vector<Type> tx_(tx);
                    tx_[3] = tx_[3] + one;  // Partial wrt. alpha
                    px[1]  = D_incpl_beta(tx_)[0] * py[0];
                    tx_[3] = tx_[3] - one;
                    tx_[4] = tx_[4] + one;  // Partial wrt. beta
                    px[2]  = D_incpl_beta(tx_)[0] * py[0];
                    px[3]  = 0;             // m is a constant
                    px[4]  = 0;             // n is a constant
                    )

}


/** Beta CDF function

    Same as pbeta from R.
    \note Non-centrality parameter (ncp) not implemented.
*/
template<class Type>
Type pbeta(Type q, Type shape1, Type shape2){
  CppAD::vector<Type> tx(5);
  tx[0] = q;
  tx[1] = shape1;
  tx[2] = shape2;
  tx[3] = 0;
  tx[4] = 0;
  Type ans = atomic::D_incpl_beta(tx)[0];
  Type logBeta =
    lgamma(shape1) +
```

```
    lgamma(shape2) -
    lgamma(shape1 + shape2);
  return ans / exp(logBeta);
}
```

tmb/atomic_beta.hpp

When testing the accuracy of our new special function we found that it had poor accuracy for shape parameters near the boundary. Futhermore, the performance was not acceptable. In particular the cheap gradident principle did not hold. We therefore discarded this implementation.
The example demonstrates that there is no easy/automatic way to implementing special functions. Numerical integration should only be used as a last resort.

4.2 Implementing pbeta in ADMB

Implementing `pbeta` was simple, because a function `betai` was already to be found in the source code of ADMB. The function `betai` is the incomplete beta function, which is the same thing as pbeta.

To make the function more accessible and easier to find we introduced an alias `pbeta` same arguments as the similar function in TMB and in **R**. This was as simple as:

```
/** beta distribution function for variable objects (alias of ibeta function with same
     arguments order as in {\bf R}).
    \param x \f$x\f$
    \param a \f$a\f$
    \param b \f$b\f$
    \param maxit Maximum number of iterations for the continued fraction approximation in
         betacf.
    \return Incomplete beta function \f$I_x(a,b)\f$

    \n\n The implementation of this algorithm was inspired by
    "Numerical Recipes in C", 2nd edition,
    Press, Teukolsky, Vetterling, Flannery, chapter 2
*/
dvariable pbeta(const dvariable x, const dvariable a, const dvariable b, int maxit){
  return betai(a, b, x, maxit);
}
```

The implemented `betai` functions only supported scalar arguments for x, a, and b. To make the `pbeta` function more easily useful it should support vector arguments also. So just counting the number of possible combinations within the constant types (not keeping track of derivatives) this would lead to 8 combinations (S,S,S), (V,S,S), (S,V,S), (S,S,V), (V,V,S), (V,S,V), (S,V,V), (V,V,V), all of which would have to be written with appropriate loops for the vector arguments. Similarly we would need 8 combinations for the number type used for purely fixed effects models (`dvariable`), and 8 combinations for the number type used for random effects models (`df1b2variable`). It is a mess to write all of these functions for the `pbeta` function, but consider doing this for all the functions in ADMB. Frequent users of AD Model

builder know that sometimes a specific needed combination of arguments is not implemented and that they simply need to write the loop.

Template Model Builder has a semi-automated way to generate the vectorized versions. It uses C++ macros, such that the C++ preprocessor expands the different vectorized versions from the scalar version. This macro has been modified to work with ADMB and included in the source tree. The code is:

```
// Copyright (C) 2013-2015 Kasper Kristensen
// Modified for ADMB by Anders Nielsen
// License: GPL-2

/**  \file
     \brief Macros to do vectorization.
     */

#include <df1b2fun.h>

template<class VT>
struct getScalarType{
};

template<>
struct getScalarType<dvector>{
  typedef double scalar;
};

template<>
struct getScalarType<dvar_vector>{
  typedef dvariable scalar;
};

template<>
struct getScalarType<df1b2vector>{
  typedef df1b2variable scalar;
};

//typename getVectorType<dvariable>::vector

// Function body type declarations
// V=vector, T=scalar, I=integer, N=none
//#define declareV(arg) const dvar_vector &arg
#define declareV(arg) const VectorType &arg
#define declareT(arg) typename getScalarType<VectorType>::scalar arg
#define declareI(arg) int arg
#define declareN(arg)
// How to extract elementwise subset of the four types
#define elementV(arg,i) (typename getScalarType<VectorType>::scalar) arg[arg.indexmin()+i]
#define elementT(arg,i) arg
#define elementI(arg,i) arg
#define elementN(arg,i)
// How to place comma in front of the types
#define commaV ,
#define commaT ,
#define commaI ,
#define commaN
// Update output vector size
#define outputsizeV(n,arg) n = ((arg.indexmax()-arg.indexmin()+1)>n ? (arg.indexmax()-arg.
     indexmin()+1) : n)
#define outputsizeT(n,arg)
#define outputsizeI(n,arg)
#define outputsizeN(n,arg)
/** \brief General vectorize macro up to 6 arguments

    Applied type abbreviations: V=vector, T=scalar, I=integer, N=none.
    The longest vector input determines the length of the output.
    Arguments are not re-cycled; unequal vector lengths should result
    in a crash.
*/
#define GVECTORIZE(FUN,Type1,Type2,Type3,Type4,Type5,Type6)      \
template<class VectorType> \
VectorType FUN( declare##Type1(arg1) comma##Type2           \
            declare##Type2(arg2) comma##Type3               \
            declare##Type3(arg3) comma##Type4               \
            declare##Type4(arg4) comma##Type5               \
            declare##Type5(arg5) comma##Type6               \
            declare##Type6(arg6) )                  \
{                                           \
```

```
  int n = 0;                                          \
  outputsize##Type1(n,arg1);                               \
  outputsize##Type2(n,arg2);                               \
  outputsize##Type3(n,arg3);                               \
  outputsize##Type4(n,arg4);                               \
  outputsize##Type5(n,arg5);                               \
  outputsize##Type6(n,arg6);                               \
  VectorType res(1,n);                                \
  for(int i=0;i<n;i++) res[i+1] = FUN( element##Type1(arg1,i) comma##Type2 \
                         element##Type2(arg2,i) comma##Type3 \
                         element##Type3(arg3,i) comma##Type4 \
                         element##Type4(arg4,i) comma##Type5 \
                         element##Type5(arg5,i) comma##Type6 \
                         element##Type6(arg6,i) );          \
  return res;                                         \
}

/** \brief Vectorize 1-argument functions. */
#define VECTORIZE1_t(FUN)               \
  GVECTORIZE(FUN,V,N,N,N,N,N)

/** \brief Vectorize 2-argument functions.

    For two-arguments functions (Type, Type),
    vectorize both arguments.
*/
#define VECTORIZE2_tt(FUN)              \
  GVECTORIZE(FUN,V,T,N,N,N,N)           \
  GVECTORIZE(FUN,T,V,N,N,N,N)           \
  GVECTORIZE(FUN,V,V,N,N,N,N)

/** \brief Vectorize 3-argument functions.

    For three-arguments functions (Type, Type, int),
    vectorize first two arguments.
*/
#define VECTORIZE3_tti(FUN)             \
  GVECTORIZE(FUN,V,T,I,N,N,N)           \
  GVECTORIZE(FUN,T,V,I,N,N,N)           \
  GVECTORIZE(FUN,V,V,I,N,N,N)

/** \brief Vectorize 3-argument functions.

    For three-arguments functions (Type, Type, Type),
    vectorize all three arguments.
*/
#define VECTORIZE3_ttt(FUN)             \
  GVECTORIZE(FUN,V,T,T,N,N,N)           \
  GVECTORIZE(FUN,T,V,T,N,N,N)           \
  GVECTORIZE(FUN,T,T,V,N,N,N)           \
  GVECTORIZE(FUN,V,V,T,N,N,N)           \
  GVECTORIZE(FUN,T,V,V,N,N,N)           \
  GVECTORIZE(FUN,V,T,V,N,N,N)           \
  GVECTORIZE(FUN,V,V,V,N,N,N)

/** \brief Vectorize 4-argument functions.

    For Four-arguments functions (Type, Type, Type, int),
    vectorize first three arguments.
*/
#define VECTORIZE4_ttti(FUN)            \
  GVECTORIZE(FUN,V,T,T,I,N,N)           \
  GVECTORIZE(FUN,T,V,T,I,N,N)           \
  GVECTORIZE(FUN,T,T,V,I,N,N)           \
  GVECTORIZE(FUN,V,V,T,I,N,N)           \
  GVECTORIZE(FUN,T,V,V,I,N,N)           \
  GVECTORIZE(FUN,V,T,V,I,N,N)           \
  GVECTORIZE(FUN,V,V,V,I,N,N)

/** \brief Vectorize 6-argument functions.

    For Six-arguments functions (Type, Type, Type, Type, Type, int),
    vectorize first three arguments.
*/
#define VECTORIZE6_tttttti(FUN)             \
  GVECTORIZE(FUN,V,T,T,T,T,I)           \
  GVECTORIZE(FUN,T,V,T,T,T,I)           \
  GVECTORIZE(FUN,T,T,V,T,T,I)           \
  GVECTORIZE(FUN,V,V,T,T,T,I)           \
  GVECTORIZE(FUN,T,V,V,T,T,I)           \
  GVECTORIZE(FUN,V,T,V,T,T,I)           \
  GVECTORIZE(FUN,V,V,V,T,T,I)
```

```
// functions vectorized
VECTORIZE3_ttt(pbeta);
```

admb/src/tools99/Vectorize.hpp

The line at the end stating that:

```
VECTORIZE3_ttt(pbeta);
```

Is the specific code needed to expand the pbeta function to support all 24 combinations outlined above.

These macros should be useful for anyone adding new functions to ADMB, but also for single users adding their own functions within their templates. As an example consider this model formulation:

```
GLOBALS_SECTION
  #include <fvar.hpp>
  dvariable logBH(dvariable ssb, dvariable loga, dvariable logb){
    return loga+log(ssb)-log(1+exp(logb)*ssb);
  }
  #include <Vectorize.hpp>
  VECTORIZE3_ttt(logBH);

DATA_SECTION
  init_int n
  init_vector ssb(1,n)
  init_vector logR(1,n)
PARAMETER_SECTION
  init_number loga;
  init_number logb;
  init_number logSigma;
  sdreport_number sigmaSq;
  vector pred(1,n);
  objective_function_value nll;
PROCEDURE_SECTION
  sigmaSq=exp(2.0*logSigma);
  pred=logBH((dvar_vector)ssb,loga,logb);
  nll=0.5*(n*log(2*M_PI*sigmaSq)+sum(square(logR-pred))/sigmaSq);
```

admbex/bh.tpl

Here a user defined function is vectorized via the macro and the loops can be avoided in the user template.

4.3 Implementing qbeta in ADMB

The existing ADMB function `inv_cumd_beta_stable` is intended to be the quantile function for the beta distribution, so it was expected to be simple to create an alias (`qbeta`) for that function. Unfortunately two problems with the implementation were discovered.

Accuracy Running a few examples gave concern about the accuracy of `inv_cumd_beta_stable` function, so it was decided to compare it to the build-in **R** function `qbeta`. A grid of reasonable values for the three inputs

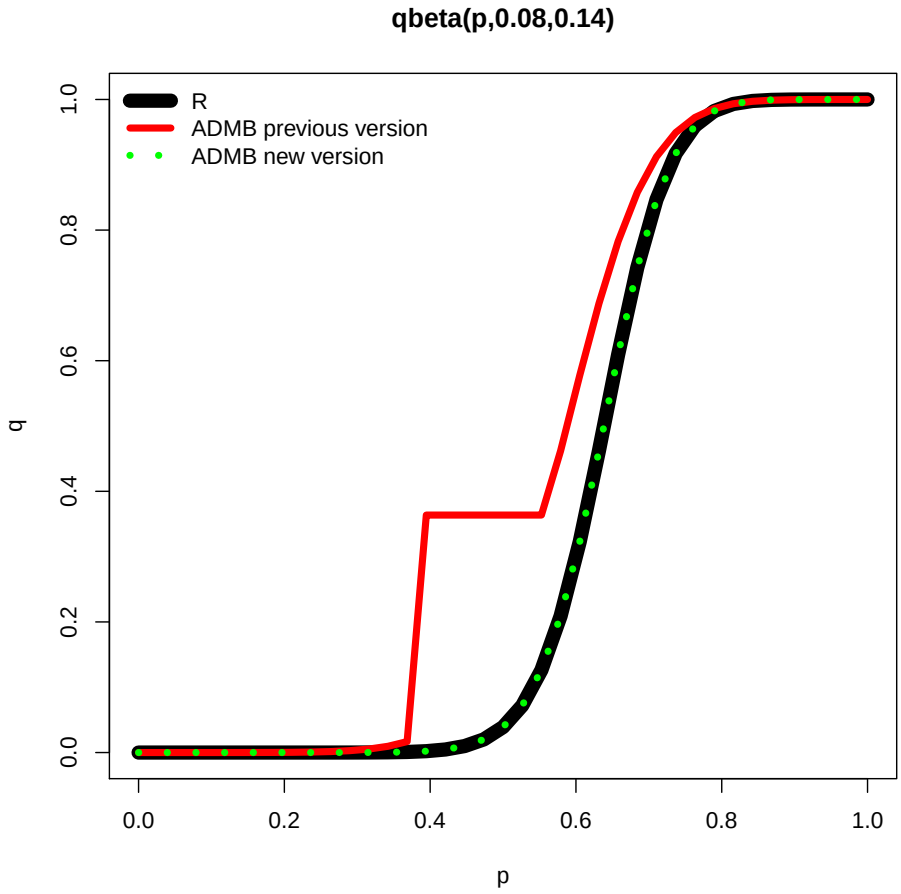

Figure 1: Three different implementations of the quantile function for the beta distribution with $\alpha = 0.08$ and $\beta = 0.14$.

p, α, and β was constructed For p values in $\left[10^{-6}; 1-10^{-6}\right]$ and for the two shape parameters values in $[0.03; 500]$ were tested (so no extremes).

The errors were higher than expected. For instance the input arguments $p = 0.473684$, $\alpha = 0.08$, and $\beta = 0.14$ gave a quantile of 0.36364. The correct answer according to **R** is 0.020135, so not a small difference. A plot of the two functions illustrates the problem (Fig. 1). The existing ADMB function was not correct, as the flat part in the middle does not correspond to a beta distribution. There is no reason to doubt the accuracy of the implementation in **R**.

Over the grid points the range of errors in the existing ADMB function (compared to the **R** function) spanned from -0.84 to 0.68 with a root mean square error (RMSE) of 0.06.

The code for the existing function was inspected and experimented with, but the cause of the error was not identified. It was noted that the numerical function solver worked on a transformed scale, which seemed unnecessary. The code was rewritten to work on the untransformed scale and the accuracy problem disappeared (green dots on Fig. 1).

Over the grid points the range of errors in the new ADMB function (compared to the **R** function) spanned from -1e-9 to +1e-9 with a RMSE of 1.6e-11.

The code for both the new and the old version of `inv_cumd_beta_stable` are not included in this report, but can be found in the file:

```
admb/src/linad99/ccumdbetainv.cpp
```

In addition an alias `qbeta` is introduced, to allow the function to be called with same arguments as the similar function in **R** .

Derivatives The derivatives were first checked by running a small models using the `qbeta` function and validating that the finite difference approximations were matching the automated first derivatives. In models with random effects the function value itself depends on the second derivatives, so when validating the first derivatives of such a model it is validating up to the third derivatives (although in an indirect way).

It was discovered that it was difficult to match the final decimals of the derivatives, so a closer inspection was necessary. All the derivatives (up to third order) is coded up in the file

```
admb/src/df1b2-separable/df1b2invcumdbeta.cpp
```

This file was modified to print all the derivatives 1., 2., and 3. order, and these were compared to finite difference approximations (from the `numDeriv` package in **R**). Exactly one of the third derivatives was different (about a factor of two in our example). The error was identified and corrected as indicated here:

```
    double F_yzz=-(F_xxx * square(*z.get_u_z())* *z.get_u_y()
                +2.0*F_xxz * *z.get_u_z() * *z.get_u_y()
                +2.0*F_xyz* *z.get_u_z()
                +F_xxy*square(*z.get_u_z())
                +2.0*F_xx* *z.get_u_z() * *z.get_u_yz()
                +F_xx* *z.get_u_y() * *z.get_u_zz()
                +2.0*F_xz* *z.get_u_yz()
                // +F_xy * *z.get_u_zz()* *z.get_u_y()  //** original
                +F_xy * *z.get_u_zz()                  //** corrected to
                +F_xzz * *z.get_u_y()
                +F_x * *z.get_u_yzz());
```

After correcting this all derivatives exactly matched the finite difference versions.

It was difficult to compare, validate and correct these derivatives, so it is of interest to develop a simpler way to obtain efficient and correct derivatives of such special functions. Further a build in tool for directly validating the higher order derivatives (like for first order) would be very useful.

4.4 Checking higher order derivatives in ADMB

In the previous example (the `qbeta` function) the higher order derivatives were evaluated by going into the code for the function and adding statements to print out the derivatives, recompile AD Model Builder and run a simple example. This procedure is not generally possible, as most functions in ADMB does not have the access to the higher order derivatives within.

To develop correct code it is important to be able to validate — especially the difficult derivative calculations. ADMB has an easy interface to comparing the first order derivatives to finite difference approximation. For random effects models the likelihood to be optimized depends on the second derivatives, so validating the first derivatives of that is indirectly validating the third order derivatives of the contributing functions.

If it is discovered that the first order derivatives of a random effects model are wrong it is currently difficult to trace which of the second or third order derivatives are causing it.

A current useful kludge to get the second derivatives has been added to ADMB (in the files `src/df1b2-separable/df1b2lap.cpp` and `src/df1b2-separable/df1b2lp1.cpp`).

The second derivatives of the joint negative log likelihood $\ell(x,\theta,u)$ w.r.t. the random effects u are used in the Laplace approximation and that is only place they are evaluated in AD Model Builder. The added code prints out the second derivatives at that point and makes sure the point of evaluation is exactly as specified.

In order to get exactly the derivatives wanted, e.g. for a specific function, the joint negative log likelihood specified must be equal to the function plus a penalty which guards the Laplace approximation against failing for numerical reasons. This penalty can in principle be anything large enough, but since it need to be subtracted from the reported second derivatives it should be chosen moderately to avoid floating point imprecisions creeping in.

Example second derivatives of pbeta: To evaluate the second order derivatives of the `pbeta(q,a,b)` function at the point (0.1, 0.2, 0.3) the following code can now be used:

```
GLOBALS_SECTION
  extern double CHECK_HESSIAN_PENALTY;

DATA_SECTION

PARAMETER_SECTION
  init_number dummy
  random_effects_vector u(1,3);
  objective_function_value nll

PROCEDURE_SECTION
  CHECK_HESSIAN_PENALTY=10;
  ad_comm::print_hess_and_exit_flag=true;
  dvariable q=0.1+u(1), a=0.2+u(2), b=0.3+u(3);
  dvariable fun=pbeta(q,a,b);
  nll=fun+(CHECK_HESSIAN_PENALTY)*sum(square(u));
```

admbex/dd2.tpl

The code must be compiled, like any ADMB program, and then run with the command line flags `-imaxfn 0 -noinit`, as in:

```
./dd2 -imaxfn 0 -noinit
```

The output produced will contain the second derivatives, here:

```
 ------------------------------------------------------------------
Hess:
 -6.33117760106491 0.8967385470621572 1.256776070934514
 0.8967385470621574 7.830304687659307 -0.4582248852821856
 1.256776070934514 -0.4582248852821875 -2.058709916695477
------------------------------------------------------------------
```

These second derivatives can then be validated against analytically derived, or finite difference approximations. Currently the **R**-package `numDeriv` can be helpful to get second order finite difference approximations, but long term such approximations should also be part of ADMB.

4.5 Implementing Bessel functions in ADMB

ADMB did not have a build-in Bessel functions. Bessel functions are solutions to Bessel's differential equation, but for statistical applications they mainly appear as part of the normalizing constant in probability density functions (e.g. von Mises and Skellam distributions).

The code for the different Bessel functions were copied from the code in Numerical Recipes and modified to be used with the different AD Model Builder types for constant, first order, and third order AD. Finally aliases were added to allow the functions to be called with the same arguments, in the same order as in **R** .

```
admb/src/linad99/cbessel.cpp
admb/src/linad99/vbessel.cpp
admb/src/df1b2-separable/df1b2bessel.cpp
```

The implementation supplies the functions `besselI`(x,ν), `besselK`(x,ν), `besselJ`(x,ν), and `besselY`(x,ν), where the argument $x >= 0$ and the order ν is an integer.

Testing the precision of the Bessel functions

The implemented functions were compared to the corresponding functions in **R** in on a grid of 10000 points (x,ν). Values of x between 0.0001 and 25 and values of ν in {1,2,...,10} were tested.

For the four Bessel functions the maximum absolute relative error was 2.1e-5, and the average absolute relative difference was 1.9e-8.

Example using Bessel function

The von Mises distribution is a continuous probability distribution on an interval (e.g. on an interval from 0 to 2π). The distribution is circular, such that the density has the same value in both ends of the interval. The distribution is also known as the circular normal distribution, and could be suitable to describe observed angles or observed times within a year. The probability density function is given by:

$$f(x,\mu,\kappa) = \frac{e^{\kappa\cos(x-\mu)}}{2\pi I_0(\kappa)}$$

Here the function $I_0(\kappa)$ in the denominator is the modified Bessel function of order 0.

Assuming that $x_1, x_2, \ldots, x_N$ is observed from a von Mises distribution the model is can be implemented in ADMB as:

```
DATA_SECTION
  init_int N
  init_vector X(1,N)
PARAMETER_SECTION
  init_bounded_number mu(0,2*M_PI)
  init_number logKappa
  sdreport_number kappa
  objective_function_value nll;
PROCEDURE_SECTION
  kappa=exp(logKappa);
  // Von Mises -logL:
  nll = -kappa*sum(cos(X-mu))+N*log(2*M_PI)+N*log(besselI(kappa,0));
```

admbex/bess.tpl

5 A unified approach to special functions: tiny_ad

Hand coding higher order derivatives has been shown to be a challenging task that can easily go wrong. In this section we propose an automatic

solution to the problem that works in TMB as well as in ADMB. In general special functions have a low number of input parameters - one to three is typical. For such small inputs we can obtain a good AD efficiency using naive forward mode automatic differentiation as opposed to reverse mode AD. The forward differentiated functions can be plugged into the reverse mode AD framework of TMB and ADMB. Forward mode AD has the following advantages:

- No need to build a computational graph so adaptive algorithms are easily handled.
- Low memory usage.
- Easy to get higher order derivatives using C++ template techniques.

5.1 Implementation of tiny_ad

A small forward mode library 'tiny_ad' was implemented (less than 300 lines of code). Here is a demonstration of how it works:

```
template<class Type, class Vector>
struct ad {
  // Data
  Type   value;
  Vector deriv;
  // Constructors
  ad(){}
  ad(Type v, Vector d) { value = v; deriv = d; }
  ad(double v)         { value = v; deriv.setZero(); }
  // Derivative rule of addition operator
  ad operator+ (const ad &other) const {
    return ad(value + other.value,
              deriv + other.deriv);
  }
  // Derivative rule of multiplication operator
  ad operator* (const ad &other) const {
    return ad(value * other.value,
              value * other.deriv +
              deriv * other.value);
  }
  // Remaining derivative rules ...
};
```

This is almost a classic way to implement tapeless forward mode AD by operator overloading. The type 'ad' contains a scalar (the value) and a vector of derivatives. The scalar type as well as the vector type are template arguments to the class.
By inserting an 'ad' type as scalar type we automatically get 2nd (and higher) order derivatives. Having a templated scalar type thus give us higher order derivatives 'for free'.
The ability to switch between different vector classes serves two purposes. First, ADMB and TMB do not share the same underlying vector class. It would be natural to use Eigen's vector class in TMB but inconvenient to require the Eigen library included in ADMB. Secondly, it is useful to be able to

switch between vector classes in order to investigate the vector class impact on AD performance.

5.2 The three vector classes

We considered the following vector class options that can be switched between using a preprocessor flag:

valarray A fixed size specialization of the *dynamic sized* vector container std::valarray was implemented. The idea is that this is already available for TMB and ADMB wihout the need to include additional code. This class is enabled in tiny_ad by adding '-DTINY_AD_USE_STD_VALARRAY' to the preprocessor.

Eigen The Eigen library provides fixed size containers. It is able to optimize vector expressions at compile time which should give faster execution. This class is enabled in tiny_ad by adding the preprocessor flag '-DTINY_AD_USE_EIGEN_VEC'.

tiny_vec To avoid the Eigen dependency we implemented a small *fixed size* vector class from scratch. It comes close to Eigen wrt. performance although tiny_vec does not use expression templates for compile time optimization. This class is intended to be used as default with tiny_ad. Currently enabled with the preprocessor flag '-DTINY_AD_USE_TINY_VEC'.

offset (bytes)	ad<valarray>	ad<EigenVector>	ad<tiny_vec>
0	Value	Value	Value
8	Length	Empty	1st derivative
16	Pointer	1st derivative	2nd derivative
24		2nd derivative	
32			

Table 1: Example of memory layout of the ad type in the case where the derivatives vector has length equal to 2 for the three different vector classes (columns).

Unfortunately valarrays do not perform well inside our AD class. The reason can be seen from Table 1 second column. A valarray consists of a pointer to a separate pool of data and a vector length. The size of an AD structure with this valarray vector is only 24 bytes *independent of the number of parameters*. However, each new AD struct instance invokes the dynamic memory allocator introducing way too much overhead. The allocated derivatives data might end up far apart from the value. This is unfortunate as we know

that the AD algorithm always needs derivatives and value simultanously.
The Eigen class outperformed valarrays with a factor 20-40 in some early benchmarks. The memory layout is shown in the third column. Notably, Eigen guaranties that data is 16 bytes aligned in order to utilize SIMD instructions. Gcc requires same alignment for the value member. Consequently there is unused space between derivatives and value.
Final column shows the memory layout for the tiny_vec class. This is the most direct layout of the three.

5.3 Theory of tiny_ad

Let x be a variable that depends on some parameters θ. A tiny_ad variable $\tilde{x}$ packs the value and a number of derivatives into a single structure. For simplicity assume that there is only one parameter i.e. the vector length in the tiny_ad class is one. The structure then contains:

$$\tilde{x} = \left(x, \frac{\partial}{\partial \theta_1} x \right)$$

Denote by `vec<n, type>` our vector structure of length n with elements of type 'type'. Then $\tilde{x}$ corresponds to the structure

```
ad<double, vec<1,double> >
```

Now let's see what happens when this structure is used as scalar type in a new tiny_ad struct. Assume that we want derivatives wrt. a new parameter θ_2. Packing the previous value $\tilde{x}$ and its derivative into a single structure yields

$$\tilde{\tilde{x}} = \left(\tilde{x}, \frac{\partial}{\partial \theta_2} \tilde{x} \right) = \left(x, \frac{\partial}{\partial \theta_1} x, \frac{\partial}{\partial \theta_2} x, \frac{\partial^2}{\partial \theta_1 \partial \theta_2} x \right)$$

In template syntax this nested structure is given by

```
ad<ad<double,vec<1,double>>,vec<1,ad<double,vec<1,double>>>>
```

So, by running an algorithm on this type we are able to track any given 2nd order cross partial derivative. When multiple cross-partials are sought we can take advantage of the vector class inside the ad struct. It allow us to have multiple derivatives associated with a single value. We could start by setting up a type to track two derivatives:

$$\tilde{x} = \left(x, \frac{\partial}{\partial \theta_1} x, \frac{\partial}{\partial \theta_2} x \right)$$

the corresponding type that tracks these derivatives is

```
ad<double, vec<2,double> >
```

By nesting this type into one of the same kind we get

$$\begin{aligned}\tilde{\tilde{x}} &= \left(x, \frac{\partial}{\partial\theta_1}x, \frac{\partial}{\partial\theta_2}x, \frac{\partial}{\partial\theta_1}\left(x, \frac{\partial}{\partial\theta_1}x, \frac{\partial}{\partial\theta_2}x\right), \frac{\partial}{\partial\theta_2}\left(x, \frac{\partial}{\partial\theta_1}x, \frac{\partial}{\partial\theta_2}x\right)\right) \\ &= \left(x, \frac{\partial}{\partial\theta_1}x, \frac{\partial}{\partial\theta_2}x, \frac{\partial}{\partial\theta_1}x, \frac{\partial^2}{\partial\theta_1^2}x, \frac{\partial^2}{\partial\theta_1\partial\theta_2}x, \frac{\partial}{\partial\theta_2}x, \frac{\partial^2}{\partial\theta_1\partial\theta_2}x, \frac{\partial^2}{\partial\theta_2^2}x\right)\end{aligned}$$

There is a substantial amount of bookkeeping involved working with these deeply nested structures. The same derivatives occur at multiple locations in the structure thus one must be careful when seeding the object (setting derivatives to one before running an algoritm). The same holds for getting the relevant information out of the object once a calculation is completed. To make things easier for the user we have introduced a class `variable<n,m>` that carries out all bookkeeping for the case where all $n$the order derivatives wrt. m variables are calculated.

Example How to get 3rd order derivatives wrt. 2 variables using tiny_ad.

```
#include "tiny_ad.hpp"
int main() {
  typedef tiny_ad::variable<3, 2> Float; // Track 3rd order derivs wrt. 2 parameters
  Float a (1.23, 0);                     // Let a=1.23 have parameter index 0
  Float b (2.34, 1);                     // Let b=2.34 have parameter index 1
  Float y = a * sin(a + b);              // Run the algorithm
  std::cout << y.getDeriv() << "\n";     // Get all 3rd order derivatives
}
```

The code produces the output

```
[ 2.36511 1.94969 1.94969 1.53427 1.94969 1.53427 1.53427 1.11884 ]
```

which should be regarded as a 2-by-2-by-2 array. There is no ambiguity in the order of the array due to symmetry of the higher order derivatives. We could dig a bit deeper into the underlying structure by replacing the final print statement with

```
  std::cout << y << "\n";                // Print underlying structure
```

Then the following output is produced

```
{ value={ value={ value=-0.51097 deriv=[ -1.53427 -1.11884 ]} deriv=[
{ value=-1.53427 deriv=[ -1.30829 -0.398659 ]} { value=-1.11884
deriv=[ -0.398659 0.51097 ]} ]} deriv=[ { value={ value=-1.53427
deriv=[ -1.30829 -0.398659 ]} deriv=[ { value=-1.30829 deriv=[ 2.36511
1.94969 ]} { value=-0.398659 deriv=[ 1.94969 1.53427 ]} ]} { value={
value=-1.11884 deriv=[ -0.398659 0.51097 ]} deriv=[ { value=-0.398659
deriv=[ 1.94969 1.53427 ]} { value=0.51097 deriv=[ 1.53427 1.11884 ]}
]} ]}
```

This is an example of the deeply nested structure containing all possible derivatives from order 0 to order 3 (with some duplicates).

5.4 Expected performance of tiny_ad

As previously noted we can represent a value and n derivatives

$$\tilde{x} = \left(x, \frac{\partial}{\partial \theta_1} x, ..., \frac{\partial}{\partial \theta_n} x \right)$$

by the structure

```
ad<double, vec<n,double> >
```

Under a few simplifying assumptions one can estimate the time it takes to calculate the gradient versus the function value for some of the basic arithmetic operators. For instance the addition $\tilde{x} + \tilde{y}$ requires $n+1$ elementary additions. The gradient/function work ratio is thus $n+1$. Likewise the $\tilde{x} * \tilde{y}$ operation requires $1+2n$ elementary multiplications and n elementary additions. Assuming equal work of multiply and add we get a work ratio of $3n+1$. Assume further that a builtin special function such as *exp*, *log*, *sin* or *cos* requires at least 10 times the work of an elementary multiplication. As an example the calculation $sin(\tilde{x})$ expands to the evaluation of function value $sin(x)$, derivative $cos(x)$ and n multiplications for the chain rule. The resulting gradient/function work ratio is thus bounded by $work(1 \times sin + 1 \times cos + n \times multiply)/work(1 \times sin) = 2 + \frac{n}{10}$.

Operator	$work(f)$	$work(d)$
Addition	1	$1+n$
Multiplication	1	$1+3n$
sin,*cos* etc.	10	$20+n$

Table 2: Estimated upper bound of gradient to value work ratio for different operators assuming that 1. n derivatives are calculated. 2. The work of multiply and add is the same. 3. The work of a builtin special function is at least 10 times the work of a multiplication.

The results are summarized in Table 2. More generally, the cost of the kth order derivatives are found by the formula

$$work(d^{(k)}) = \begin{pmatrix} 1+n & 0 & 0 \\ n & 1+2n & 0 \\ 0 & n & 2 \end{pmatrix}^k \begin{pmatrix} 1 \\ 1 \\ 10 \end{pmatrix}$$

for any $k \geq 0$.
Can we maintain a cheap gradient principle ($work(gradient)/work(function) <$

4) for reverse mode AD when plugging in the forward mode differentiated functions? Apparently the answer heavily depends on what operations are used by the algorithm. An algorithm dominated by multiplication seems to have problems already for more than one parameter. In contrast an algorithm where builtin special functions dominate can maintain the cheap gradient principle for more than 10 parameters.

There are two important factors that affect the forward mode approach in a positive direction: *instruction level parallelism* and *CPU register usage*. As an example consider the following recursive procedure consisting of multiply and addition:

```
template<class Float>
Float test(Float a, Float b) {
  Float x = 0;
  int n = 1e9;
  for(int i=0; i<n; i++) {
    x *= a;
    x += b;
  }
  return x;
}
```

This procedure is fairly close to a typical series expansion found in many special functions. In its direct form the algorithm is unable to take advantage of instruction level parallelism due to the recursive nature. However, when inserting an AD type `Float=tiny_ad::variable<1,3>` as template parameter a large portion of the derivatives can in principle be calculated in parallel by the vectorized CPU instructions (SIMD). Indeed, the generated assembler using `g++ -O3 -S` for the two cases `Float=double` and `Float=tiny_ad::variable<1,3>` respectively confirmed this theory. The AD code was able to fill up *all* CPU registers with derivatives while the double version only took advantage of *one* (128 bit) CPU register.

5.5 Measuring the performance of tiny_ad

Four benchmark examples were constructed to test the performance of tiny_ad. The first two 'bessel' and 'pbeta' are based on some ported special functions (see later section). The final two examples are toy problems that can be held up against the theory from the previous section:

```
template<class Float>
double multiply_add(Float x, Float a, Float b){
  Float ans = x + a + b;
  int n = 1e7;
  for(int i=0; i<n; i++) {
    ans = ans + x;
    ans = ans * x;
  }
  return ans;
}
```

Listing 1: Recursive multiply-add benchmark

```
template<class Float>
double exp_sin_cos(Float x, Float a, Float b){
  Float ans = x + a + b;
```

```
  int n = 1e6;
  for(int i=0; i<n; i++) {
    ans = exp(ans);
    ans = sin(ans);
    ans = cos(ans);
  }
  ans;
}
```

Listing 2: Recursive exp-sin-cos benchmark

A small **R**-script can be used to calculate the expected work ratios of derivatives versus function values. The machine used for the benchmark has ca. equal work for multiply and add whereas a builtin special function takes roughly 14 times longer to evaluate:

```
w0 <- c('add'=1, 'mult'=1, 'specfunc'=14)
A <- function(n) matrix(c(1+n, n, 0, 0, 1+2*n, n, 0, 0, 2),3)
work.ratio <- function(algo=c(1,1,0), n=3, order=1) {
    wd <- matrix(0, 3, order+1)
    wd[,1] <- w0
    for(i in seq_len(order)) wd[,i+1] <- A(n) %*% wd[,i]
    wd <- t(algo) %*% wd
    colnames(wd) <- 0:order
    wd / wd[1]
}
```

For the multiply-add benchmark we get:

```
> work.ratio(c(1,1,0), 3, order=3)
     0 1  2   3
[1,] 1 7 49 343
```

For the exp-sin-cos benchmark we get:

```
> work.ratio(c(0,0,1), 3, order=3)
     0        1        2        3
[1,] 1 2.214286 6.571429 30.71429
```

function	order	$elapsed^1$	$T(d)/T(f)^1$	$T(d)/T(f)^2$	$T(d)/T(f)^3$
bessel	0	0.01	1.00	1.00	1.00
bessel	1	0.02	3.20	3.80	2.30
bessel	2	0.07	14.80	24.00	17.30
bessel	3	0.46	91.40	133.50	140.80
pbeta	0	0.01	1.00	1.00	1.00
pbeta	1	0.08	12.70	14.50	5.00
pbeta	2	0.42	69.30	51.50	45.30
pbeta	3	3.13	521.50	334.00	420.60
multiply_add	0	0.02	1.00	1.00	1.00
multiply_add	1	0.13	6.90	1.50	1.40
multiply_add	2	0.81	42.50	13.50	31.40
multiply_add	3	6.66	350.70	131.30	282.10
exp_sin_cos	0	0.05	1.00	1.00	1.00
exp_sin_cos	1	0.09	1.70	1.50	1.60
exp_sin_cos	2	0.30	5.50	3.60	5.30
exp_sin_cos	3	1.35	24.50	13.60	21.80

Table 3: Gcc compiler results. Timings of four different test functions for derivative orders 0 to 3 ($elapsed$) and relative time of derivative calculations relative to function value ($T(d)/T(v)$). Superscript denotes the following three different configurations: 1. The tiny_vec vector class with compiler flag $-O2$. 2. The tiny_vec vector class with compiler flag $-O3-march=native$. 3. The Eigen vector class with compiler flag $-O3-march=native$.

The actual benchmark results, using the gcc compiler [1] , are shown in Table 3. Note how well the expected work ratios of derivative to function value match the theoretical work ratio (column $T(d)/T(f)^1$ last eight rows). The next column ($T(d)/T(f)^2$) shows the effect of enabling 128 bit SIMD instructions through a compiler flag. For the toy examples we get a 2-4 fold speedup. This is also in line with the theory from the previous section: The SIMD instructions benefit the derivatives - not the function value. We do not get any consistent imrovement using Eigen's vector class instead of our own 'tiny_vec' class ($T(d)/T(f)^3$) for the toy examples.

[1] gcc version 5.4.0

function	order	$elapsed^1$	$T(d)/T(f)^1$	$T(d)/T(f)^2$	$T(d)/T(f)^3$
bessel	0	0.00	1.00	1.00	1.00
bessel	1	0.01	2.00	1.70	1.50
bessel	2	0.02	6.00	5.70	17.30
bessel	3	0.17	41.20	42.50	146.00
pbeta	0	0.01	1.00	1.00	1.00
pbeta	1	0.02	2.40	2.80	2.70
pbeta	2	0.15	21.00	21.30	34.50
pbeta	3	0.90	128.00	194.30	376.50
multiply_add	0	0.02	1.00	1.00	1.00
multiply_add	1	0.03	1.50	1.50	1.50
multiply_add	2	0.26	13.80	15.30	21.80
multiply_add	3	1.77	93.10	92.70	214.50
exp_sin_cos	0	0.05	1.00	1.00	1.00
exp_sin_cos	1	0.08	1.50	1.50	1.60
exp_sin_cos	2	0.18	3.30	3.30	4.30
exp_sin_cos	3	0.62	11.30	11.40	15.60

Table 4: Clang compiler results. Timings of four different test functions for derivative orders 0 to 3 (*elapsed*) and relative time of derivative calculations relative to function value ($T(d)/T(v)$). Superscript denotes the following three different configurations: 1. The tiny_vec vector class with compiler flag $-O2$. 2. The tiny_vec vector class with compiler flag $-O3 - march = native$. 3. The Eigen vector class with compiler flag $-O3 - march = native$.

For comparison we tried to run the exact same benchmark using the clang compiler [2] (Table 4). Note that the meaning of compiler optimization flags are not the same on clang vs gcc. The clang compiler enables SIMD already at '-O2'. Clang does a remarkably good job in this benchmark: the gradient to function work ratio does not exceed 2.5 for any of the examples! We do not have any good explanation for this result.

6 Porting C-code to using tiny_ad

The tiny AD library has been used to differentiate complex algorithms written in C. One such example is the file 'toms708.c' (part of the **R** source code tree) containing more than 2200 lines of helper functions to deal with all corner cases of the beta CDF function! This C code is much more accurate and almost 10 times as fast as the corresponding function from numerical recipies. Porting the functions to tiny AD was for the most part a search-

[2] clang version 3.8.0

replace operation. Here is a small fragment of the code to demonstrate the process:

```
static double betaln(double a0, double b0)
{
/* ----------------------------------------------------------------------
 *     Evaluation of the logarithm of the beta function  ln(beta(a0,b0))
 * ---------------------------------------------------------------------- */

    static double e = .918938533204673;/* e == 0.5*LN(2*PI) */

    double
     a = min(a0 ,b0),
     b = max(a0, b0);

    if (a < 8.) {
     if (a < 1.) {
/* ---------------------------------------------------------------------- */
//                              A < 1
/* ---------------------------------------------------------------------- */
         if (b < 8.)
          return gamln(a) + (gamln(b) - gamln(a+b));
         else
          return gamln(a) + algdiv(a, b);
     }
     /* else */
/* ---------------------------------------------------------------------- */
//                 1 <= A < 8
/* ---------------------------------------------------------------------- */
     double w;
     if (a < 2.) {
         if (b <= 2.) {
          return gamln(a) + gamln(b) - gsumln(a, b);
         }
         /* else */

         if (b < 8.) {
          w = 0.;
          goto L40;
         }
         return gamln(a) + algdiv(a, b);
     }
     // else L30:    REDUCTION OF A WHEN B <= 1000

     if (b <= 1e3) {
         int n = (int)(a - 1.);
         w = 1.;
         for (int i = 1; i <= n; ++i) {
          a += -1.;
          double h = a / b;
          w *= h / (h + 1.);
         }
         w = log(w);

         if (b >= 8.)
          return w + gamln(a) + algdiv(a, b);

         // else
     L40:
         //    1 < A <= B < 8 :  reduction of B
         n = (int)(b - 1.);
         double z = 1.;
         for (int i = 1; i <= n; ++i) {
          b += -1.;
          z *= b / (a + b);
         }
         return w + log(z) + (gamln(a) + (gamln(b) - gsumln(a, b)));
     }
        // Stripped
} /* betaln */
```

Listing 3: Original C-code fragment from 'toms708.c'

We start by changing the C function to a C++ template function: The keyword `template<class Float>` is placed in front of the function declaration and 'double' is replaced by 'Float'. Note that it is safe to leave static constants as is. Often, these steps would suffice. However, in this case compilation of the source reveals that a few more changes are required.

1. The code contains a 'goto L40' and there's a declaration 'int n' in between the 'goto' and the tag 'L40:'. This is not allowed in C++ (but it's OK in C). The fix is simply to move the declaration to before the 'goto' - see the code below.

2. The code contains a cast from double to int '(int)(a - 1.)'. It would be unsafe to let the tiny AD library allow *implicit* cast from an AD type to integer. We must manually inform tiny_ad that the cast is intentional by changing the line to '(int)trunc(a - 1.)' - see code below.
   Note that *explicit* casts (introduced in C++11) could be implemented for the AD type. However, for portability reasons we avoid depending on this feature.

```
template<class Float> static Float betaln(Float a0, Float b0)
{
/* -----------------------------------------------------------------------
 *     Evaluation of the logarithm of the beta function  ln(beta(a0,b0))
 * ----------------------------------------------------------------------- */

    static double e = .918938533204673;/* e == 0.5*LN(2*PI) */

    Float
     a = min(a0 ,b0),
     b = max(a0, b0);

    if (a < 8.) {
     if (a < 1.) {
/* ----------------------------------------------------------------------- */
//                             A < 1
/* ----------------------------------------------------------------------- */
         if (b < 8.)
          return gamln(a) + (gamln(b) - gamln(a+b));
         else
          return gamln(a) + algdiv(a, b);
     }
     /* else */
/* ----------------------------------------------------------------------- */
//                  1 <= A < 8
/* ----------------------------------------------------------------------- */
     Float w;
     int n;
     if (a < 2.) {
         if (b <= 2.) {
          return gamln(a) + gamln(b) - gsumln(a, b);
         }
         /* else */

         if (b < 8.) {
          w = 0.;
          goto L40;
         }
         return gamln(a) + algdiv(a, b);
     }
     // else L30:    REDUCTION OF A WHEN B <= 1000

     if (b <= 1e3) {
         n = (int)trunc(a - 1.);
         w = 1.;
         for (int i = 1; i <= n; ++i) {
          a += -1.;
          Float h = a / b;
          w *= h / (h + 1.);
         }
         w = log(w);

         if (b >= 8.)
          return w + gamln(a) + algdiv(a, b);

         // else
     L40:
         //    1 < A <= B < 8 :  reduction of B
         n = (int)trunc(b - 1.);
         Float z = 1.;
```

```
        for (int i = 1; i <= n; ++i) {
         b += -1.;
         z *= b / (a + b);
        }
        return w + log(z) + (gamln(a) + (gamln(b) - gsumln(a, b)));
    }
       // Stripped
} /* betaln */
```

Listing 4: Resulting C++-code fragment from 'toms708.cpp'

6.1 Fixing wrong derivatives

Even if an algorithm compiles with AD types it is not certain that the derivatives are correct! We recommend checking the derivatives against numerical derivatives (e.g. using **R**'s numDeriv package) up to at least second order on a fine grid. If the derivatives turn out to be wrong it can be a big challenge to locate the root of the problem in a large code base. To this end a code coverage tool such as 'gcov' (part of gcc) turned out extremely valuable. It can quickly point out the relatively small subset of code lines executed for a given special case of the parameters.
In this section we summarize the code issues we ran into while checking the derivatives.

Underflow protection While porting the besselK function we encountered the following code:

```
if (nu < sqxmin_BESS_K) {
  nu = 0.;
}
```

Here `nu` is a parameter and `sqxmin_BESS_K` is equal to `1.49e-154`. The branch was entered with the frequently occuring value `nu=0`. The statement has no effect if nu is of double type. However, for an AD type the statement `nu=0` sets both the value *and* derivatives to zero! We out-commented the redundant line.

Redundant absolute value A redundant absolute value

```
if ( 0. <= nu && nu < 1. ) {
  // ...
  nu = fabs(nu);
  // ...
}
```

had no effect for the original code. However, for AD types we got a wrong derivative in the case `nu=0`. This is because tiny_ad differentiates *abs* to the *sign* function which takes the value 0 in 0. We may consider changing this derivative to its right continuous version.

Point-wise special case A frequently occuring problem causing non-differentiability is when an algorithm has special cases for certain parameter values. Such special cases almost surely doesn't hold in a neighborhood around the parameter value. For instance the following lines appeared in the besselI function:

```
if (0. <= nu && nu < 1.) {
  if (nu != 0.)
    sum *= (Rf_gamma_cody(1. + nu) * pow(*x * .5, -nu));
}
```

This is a point-wise special case for the parameter value `nu==0` stating that in this case the multiplier evaluates to one, hence can be omitted. However, the opimization corrupts the derivative in the special case. The solution is to out-comment the if-statement.

Minimum-maximum Minimum and maximum operators must be carefully thught out before using with AD. The following code is used in the pbeta function to swap a and b if b is smallest:

```
#define min(a,b) ((a < b)?a:b)
#define max(a,b) ((a > b)?a:b)
a = min(a0, b0);
b = max(a0, b0);
```

In an AD context the problem arise when $a = b$. In this case $min(a,b)$ and $max(a,b)$ both return b. Either *min* or *max* must be redefined. We made the change `#define min(a,b) ((a <= b)?a:b)`.

Series convergence One of the most tricky situations encountered is when the number of iterations in a series suffice for the function value but not for the derivatives. This problem occured in several of the pbeta series expansions. Here is one example:

```
/* ----------------------------------------------------------------------- */
/*               COMPUTE THE SERIES */
/* ----------------------------------------------------------------------- */
Float tol = eps / a,
  n = 0.,
  sum = 0., w;
Float c = 1.;
do { // sum is alternating as long as n < b (<==> 1 - b/n < 0)
  n += 1.;
  c *= (0.5 - b / n + 0.5) * x;
  w = c / (a + n);
  sum += w;
} while (n < 1e7 && fabs(w) > tol);
```

Mostly, the loop would run for many iterations - except for integer values of the parameter b. When $b = 1$ the variable c becomes zero and the loop terminates after the first iteration even if the derivatives require more iterations to converge. We introduced a new function `max_fabs` that returns the maximum absolute value of all AD struct members (in the `double` case it is just the `fabs` function). We then replaced `fabs` with `max_fabs` in all do-while loops.

6.2 List of ported algoritms

The ported algorithms are located in the tiny_ad folder. We intended to make them as selfcontained as possible. In particular there are no **R**-dependencies and all required defines are part of each of the separate routines although this involves a certain amount of repetition. It should thus be equally easy to use the routines from ADMB and TMB.

Most algorithms are ported C-code from **R** (version 3.3.0). One function (dtweedie) was obtained from the **R**-package cplm (version 0.7-4). The ported versions are kept as close to the originals as possible in order to ease maintenance. If an original version is patched there is a chance the patch may apply unchanged to the ported version. However, given the maturity of **R** base we find it unlikly that any critical changes will take place to the original C-code.

A list of ported functions is shown in Table 5. Note that both ADMB and TMB already have the gamma function. However, not in templated form required by tiny_ad. The gamma function is a common dependency of all the algorithms except `integrate`.
The Gauss kronrod integrate routine was implemented in its original one-dimensional form and an interface for multi-dimensional integration was added.

Algorithm	Origin	License	Include
gamma and log-gamma	Base R	GPL-2	`gamma/gamma.hpp`
pbeta	Base R	GPL-2	`beta/pbeta.hpp`
bessel(K,I,Y,J)	Base R	GPL-2	`bessel/bessel.hpp`
Gauss-Kronrod integrate	Base R	GPL-2	`integrate/integrate.hpp`
dtweedie	cplm	GPL-2	`tweedie/tweedie.hpp`

Table 5: Overview of ported algorithms.

6.3 Checking derivatives of ported algoritms

The ported algorithms have been included in TMB as atomic functions. Automated derivatives checks up to order 2 have been added as part of the TMB distribution (github version). The tests are located in `tmb_syntax/check_derivatives.R` and run by sourcing this **R**-script. The script sets appropriate grids for the tested functions. For the Bessel function was used:

```
grid <- expand.grid(1:100, 1:100) / 10
```

i.e. 10000 equally spaced grid cells in the range 0 to 10 for each parameter.
For the besselK function we get the output:

```
================================================================
besselK

Value relative error:
     Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
0.000e+00 0.000e+00 0.000e+00 1.175e-17 0.000e+00 7.742e-16

Worst value grid-point (numderiv/ad):
x  = 1.3 7.6
ND = 28194.91
AD = 28194.91

Gradient relative error:
     Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
4.500e-14 5.018e-12 8.342e-12 6.142e-11 1.143e-11 1.367e-08

Worst gradient grid-point (numderiv/ad):
x  = 7.2 0.1
ND = -0.0003664404 4.47796e-06
AD = -0.0003664404 4.47796e-06

Hessian relative error:
     Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
0.000e+00 1.800e-10 1.010e-09 4.357e-07 3.660e-09 2.874e-04

Worst hessian grid-point (numderiv/ad):
x  = 0.1 10
ND = 2.043369e+22 -9.930309e+20 -9.930309e+20 5.1336e+19
AD = 2.043369e+22 -9.933164e+20 -9.933164e+20 5.134229e+19
```

The output shows a summary of the relative error for the function value, gradient and Hessian. It also points out the grid point for which value, gradient and Hessian attain the largest relative error. Similar output for the pbeta function is:

```
================================================================
pbeta

Value relative error:
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
      0       0       0       0       0       0

Worst value grid-point (numderiv/ad):
x  = 0.105 0.19 0.21
ND = 0.3650588
AD = 0.3650588

Gradient relative error:
     Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
2.912e-12 1.523e-11 2.349e-11 4.029e-11 3.879e-11 9.965e-10

Worst gradient grid-point (numderiv/ad):
x  = 0.905 0.19 1.61
ND = 0.0554032 -0.01833799 0.00891067
AD = 0.0554032 -0.01833799 0.00891067

Hessian relative error:
     Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
0.000e+00 0.000e+00 1.000e-10 1.061e-05 6.000e-10 1.599e-03

Worst hessian grid-point (numderiv/ad):
```

```
x  = 0.905 0.19 0.21
ND = 5.422079 2.121844 0.09008768 2.121844 3.954561 2.003997 0.09008768 2.003997 -8.33119
AD = 5.422844 2.121957 0.0899437 2.121957 3.954561 2.003997 0.0899437 2.003997 -8.33119
```

Judging from this output both value and gradient are very accuarate. However, there seems to be some disagreement between the numeric Hessian and the AD Hessian in the point $(0.905, 0.19, 0.21)$. Is the AD derivative wrong or is the discrepancy caused by inaccuracy in the finite difference approximation ? Fortunately there is an alternative way to calculate this Hessian. Given that we trust the AD gradient we can numerically differentiate this gradient to get the Hessian:

```
> jacobian(obj$gr, c(0.905, 0.19, 0.21))
          [,1]     [,2]       [,3]
[1,] 5.4228441 2.121957  0.0899437
[2,] 2.1219567 3.954561  2.0039972
[3,] 0.0899437 2.003997 -8.3311902
```

The hessian is in perfect aggrement with the AD Hessian. The maximum relative error is $\approx 10^{-10}$. Hence in this case the numDeriv Hessian was wrong - not the AD Hessian.
The example illustrates that automatic derivatives checking could be improved by only using finite differences to first order:

1 Check 1st order AD against 1st order numerical derivative.

2 Check 2nd order AD against 1st order numerical derivative of 1st order AD.

3 Check 3rd order AD against 1st order numerical derivative of 2nd order AD.

7 Using tiny_ad in ADMB

ADMB builds the stack of operations every time a given function is evaluated, so (unlike in TMB) reverse mode automatic differentiation can handle some conditional expressions, which are common in special functions. There are however many other reasons why tiny_ad is interesting to include in ADMB.

Tiny_ad is an elegant implementation of forward mode automatic differentiation. Forward mode automatic differentiation is efficient for functions with few input arguments (e.g. less than around 5). Forward mode is further advantageous if the function consists of a high number of operations, as forward mode does not need to store all these operations (contrary to reverse mode).

Special functions has often been implemented in AD Model Builder by first obtaining the simplest possible C++ code to evaluate the function (often

from Numerical Recipes) and then either using reverse mode automatic differentiation (which may be inefficient) or implementing the derivatives by hand (which may be prone to errors).

With tiny_ad it should be possible — and even efficient — to get the derivatives without hand written derivatives, which should enable ADMB to use the best C++ code to evaluate the functions.

7.1 Including the code in ADMB.

Tiny_ad is included into ADMB as two short files, which are copied into the files:

```
admb/src/tools99/tiny_ad.hpp
admb/src/tools99/tiny_vec.hpp
```

The first file is the forward mode automatic differentiation code to any order and the second file is a convenient and efficient vector class, which is used by tiny_ad.

The functionality is made available both for developers adding code to ADMB and for users writing support functions in their model templates. To make this possible the Makefiles are modified to copy these files to the include folder, and the files `admb/src/linad99/fvar.hpp` and `admb/src/df1b2-separable/df1b2fun.h` are updated to include tiny_ad for use in both standard (fixed effects only) ADMB and with random effects.

7.2 Using from user template — fixed effects

Using tiny_ad to calculate the first order derivative of a simple one parameter function may be a bit excessive, but is the natural first example to illustrate its use. Consider the function $f(x) = \exp(-\frac{1}{2}x^2)$ as implemented in the following complete ADMB program:

```
1  GLOBALS_SECTION
2    #include <fvar.hpp>
3
4    template<class Float>
5    Float errf(Float x){
6      Float y=exp(-0.5*x*x);
7      return y;
8    }
9
10   dvariable errf(dvariable x){
11     typedef tiny_ad::variable<1, 1> Float;
12     Float x_ (value(x), 0);
13     Float ans=errf(x_);
14     dvariable y;
15     value(y)=ans.value;
16     tiny_vec<double, 1> der = ans.getDeriv();
17     AD_SET_DERIVATIVES1(y,x,der[0]); // 1 dependent variable
18     return y;
19   }
20
21 DATA_SECTION
22
```

```
23 PARAMETER_SECTION
24   init_number x;
25   !! x=2;
26   objective_function_value f;
27
28 PROCEDURE_SECTION
29   f=square(errf((dvariable)x));
```

admbex/simpletiny.tpl

A lot of things are happening here:

line 4-8 The function is defined, but as a template type, which is required for use with tiny_ad.

line 10-19 The `dvariable` type function is defined, which is the function the user will be calling (line 10-19).

line 11 Within the function the type of derivative object is defined. The first argument is the derivative order (here 1) and the second argument is the number of parameters to the function (here 1).

line 12 The value of the function argument `x` is copied into an argument `x_` of the newly declared type. The 0 indicates argument number starting from zero.

line 13 The function is evaluated with the tiny_ad typed argument `x_`. The returned object contain both the function value and the forward mode derived gradient (here of length 1).

line 14-15 The `dvariable` type object `y` to be returned by the function is declared and it is assigned the value from the function evaluation.

line 16 The gradient is extracted from the tiny_ad object and stored in a tiny_ad vector object (this is necessary even when the gradient is of length one).

line 17 The value and derivative are stored with the macro `AD_SET_DERIVATIVES1`. Conveniently these simplifying macros exists to store up to 4 dependent variables, which approximately covers the number of dependent variables where forward mode AD is efficient.

line 18 Finally the calculated `dvariable` is returned.

The remaining code is to wrap it into a minimal standard AD Model Builder program, which is used to test the derivative by running it with the `-dd 0` flag.

7.3 Using from user template — random effects

For code which is intended to be used with random effects the first, second, and third derivatives are all needed and must be specified. Tiny_ad can automatically calculate this and the relevant derivatives can be accessed via statements like these:

```
  typedef tiny_ad::variable<3, 1> Float;
  Float x_ (value(x), 0);
  Float ans=errf(x_);
  double val=ans.value.value.value;
  tiny_vec<double, 1> der1 = ans.value.value.getDeriv();
  tiny_vec<double, 1> der2 = ans.value.getDeriv();
  tiny_vec<double, 1> der3 = ans.getDeriv();
```

Notice that now the tiny_ad variable is declared to order 3. After these lines the three vectors `der1`, `der2`, and `der3` contain the first, second, and third derivatives respectively.

Storing the derivatives in ADMB is a bit more work for random effects models. The following code illustrates how it is done:

```
  df1b2variable tmp;
  value(tmp)=val;
  double * xd=x.get_u_dot();
  double * tmpd=tmp.get_u_dot();
  for (unsigned int i=0;i<df1b2variable::nvar;i++)
  {
    *tmpd++ = der1[0] * *xd++;
  }
  f1b2gradlist->write_pass1(&x,&tmp,der1[0],der2[0],der3[0]);
    // x f(x) df/dx ddf/dxx dddf/dxxx
  return tmp;
```

First the variable to be returned is declared and its value is assigned. Next the pointers to the existing gradients are extracted. The loop uses forward mode AD to assign the gradient to the function value. Finally all the computed derivatives are written, and the function result is returned.

The collected example with tiny_ad adjoint code for both fixed and random effects models is included here:

```
GLOBALS_SECTION
  #include <df1b2fun.h>

  template<class Float>
  Float errf(Float x){
    Float y=exp(-0.5*x*x);
    return y;
  }

  dvariable errf(dvariable x)
  {
    typedef tiny_ad::variable<1, 1> Float;
    Float x_ (value(x), 0);
    Float ans=errf(x_);
    dvariable y;
    value(y)=ans.value;
    tiny_vec<double, 1> der = ans.getDeriv();
    AD_SET_DERIVATIVES1(y,x,der[0]);
    return y;
  }

  df1b2variable errf(df1b2variable x)
  {
    typedef tiny_ad::variable<3, 1> Float;
```

```
    Float x_ (value(x), 0);
    Float ans=errf(x_);
    double val=ans.value.value.value;
    tiny_vec<double, 1> der1 = ans.value.value.getDeriv();
    tiny_vec<double, 1> der2 = ans.value.getDeriv();
    tiny_vec<double, 1> der3 = ans.getDeriv();

    df1b2variable tmp;
    value(tmp)=val;
    double * xd=x.get_u_dot();
    double * tmpd=tmp.get_u_dot();
    for (unsigned int i=0;i<df1b2variable::nvar;i++)
    {
      *tmpd++ = der1[0] * *xd++;
    }
    f1b2gradlist->write_pass1(&x,&tmp,der1[0],der2[0],der3[0]);
    // x f(x) df/dx ddf/dxx dddf/dxxx
    return tmp;
  }

DATA_SECTION
  !! cout<<"Testing double version "<<errf((double)2.0)<<endl;

PARAMETER_SECTION
  init_number x;
  random_effects_vector u(1,1);
  !! x=2;
  objective_function_value f;

PROCEDURE_SECTION
   f=square(errf((dvariable)x+u(1)))+square(u(1));
```

admbex/simpletinyr.tpl

This example may appear complicated, and it is certainly excessive for this simple function. The key point is however that any function (in this case on one variable) can have derivatives computed in this way. No matter how complicated the computations within the function is. The function needs to be specified only one time. This is a huge advantages compared to having to declare essentially the same code many times for different types. Also this tiny_ad approach completely avoids hand written adjoint code.

7.4 Using tiny_ad to simplfy the AD code base

As a simple example of the usefulness of tiny_ad within the core code for AD Model builder consider the `pbeta` function. The core of this function is a continued fraction function called `betacf`.

The `betacf` function does a long series of simple calculations until it converges, so it will potentially require a lot of memory to store all the operations for reverse mode AD. To avoid this the implementation in ADMB was split into 3 files: 1) to compute just the `double` value of the function `src/linad99/cbetacf.cpp` (∼60 lines), 2) to be used with first derivatives, which uses hand written adjoint code `src/linad99/vbetacf.cpp` (∼227 lines), and 3) to be used with random effects (part of file `src/df1b2-separable/df1b2bet.cpp`). These three files all contain essentially the same algorithm and code, but carefully adapted to each use.

When using tiny_ad to setup the same functionality a single file with the algorithm is added `src/linad99/betacf_val.hpp` (∼60 lines), which is sim-

ilar to the plain `double` version, except that it is defined for the template type, as in:

```
template<class Float>
Float betacf(Float a, Float b, Float x, int MAXIT){
. . .
```

The three needed versions can now be obtained by defining three small functions calling this joint piece of code. First, to get a `double` typed version is simply:

```
#include <fvar.hpp>
#include "betacf_val.hpp"
double betacf(const double a, const double b, const double x, int MAXIT){
  typedef double Float;
  return betacf<Float>(a,b,x,MAXIT);
}
```

Second, to get a version including 1. derivatives a version supporting `dvariable` is needed:

```
#include <fvar.hpp>
#include "betacf_val.hpp"

dvariable betacf(const dvariable& a, const dvariable& b, const dvariable& x, int MAXIT)
{
  typedef tiny_ad::variable<1, 3> Float;
  Float a_ (value(a), 0);
  Float b_ (value(b), 1);
  Float x_ (value(x), 2);
  Float ans = betacf<Float>(a_, b_, x_, MAXIT);
  tiny_vec<double, 3> der = ans.getDeriv();

  dvariable hh;
  value(hh) = ans.value;
  AD_SET_DERIVATIVES3(hh,a,der[0],b,der[1],x,der[2]);
  return hh;
}
```

Notice that compared to the example of a simple one parameter shown above only a couple of things need to be changed. The tiny_ad variable need to be declared of dimension 3, a corresponding `Float` of each parameter must be declared and numbered 0,1, and 2. Finally the macro `AD_SET_DERIVATIVES3` must be used to update the derivatives. The important thing to notice is that it was not necessary to repeat the code of the actual function `betacf`.

To get a version with up to 3. order derivatives, which is needed for models including random effects, a version of type `df1b2variable` must be defined:

```
#include <df1b2fun.h>
#include "../linad99/betacf_val.hpp"

df1b2variable betacf(const df1b2variable& a,const df1b2variable& b, const df1b2variable& x,
     int MAXIT)
{
  typedef tiny_ad::variable<3, 3> Float;
  Float a_ (value(a), 0);
  Float b_ (value(b), 1);
  Float x_ (value(x), 2);
  Float ans = betacf<Float>(a_, b_, x_, MAXIT);
  double val=ans.value.value.value;
  tiny_vec<double, 3> der1 = ans.value.value.getDeriv();
  tiny_vec<double, 9> der2 = ans.value.getDeriv();
```

```
  tiny_vec<double, 27> der3 = ans.getDeriv();

  df1b2variable tmp;
  value(tmp)=val;
  double * xd=a.get_u_dot();
  double * yd=b.get_u_dot();
  double * zd=x.get_u_dot();
  double * tmpd=tmp.get_u_dot();
  for (unsigned int i=0;i<df1b2variable::nvar;i++)
  {
    *tmpd++ = der1[0] * *xd++ + der1[1] * *yd++ + der1[2] * *zd++;
  }
  if (!df1b2_gradlist::no_derivatives)
  {
    f1b2gradlist->write_pass1(&a,&b,&x,&tmp,
     der1[0],der1[1],der1[2],
     der2[0],der2[1],der2[2],der2[4],der2[5],der2[8],
     der3[0],der3[1],der3[2],der3[4],der3[5],der3[8],der3[13],der3[14],der3[17],der3[26]);
  }
  return tmp;
}
```

Also for the 3rd derivatives version a few things are different compared to the simple one parameter version above. Again tiny_ad variable need to be declared of dimension 3, and corresponding `Float` of each parameter must be declared and numbered 0,1, and 2. Next the 1st derivative is now of length 3, the 2nd derivative is of length 9, and the 3rd derivative is of length 27.

The forward loop to assign the gradient to the function value is also different now. The existing gradients of all three parameters must be extracted, multiplied with the corresponding 1st order derivatives, and added to the functions gradient.

Finally all the gradient information is written to the stack, but in ADMB only the unique derivatives should be written, which are the ones written here.

This may look complicated, but it is less so that the code originally in ADMB. It is only a fraction of the original amount of code. The code for the algorithm is only there ones. No manual adjoint code is required. This recipe for adding 1, 2, 3 parameter functions is wrapped in macros (see section 7.6) which makes it really easy to add 'computationally heavy' functions of few parameters.

7.5 Efficiency of tiny_ad in ADMB

To compare the effiency of tiny_ad to hand written adjoint code, the following ADMB code was used:

```
1  DATA_SECTION
2    vector x(1,20000)
3    !! x.fill_seqadd(0,.00005);
4    vector y(1,20000)
5    !! y=pbeta(x,0.3,0.4);
6
7  PARAMETER_SECTION
8    init_number loga
9    init_number logb
10   sdreport_number a
11   sdreport_number b
```

```
12   objective_function_value nll
13
14 PROCEDURE_SECTION
15   nll=0;
16   a=exp(loga);
17   b=exp(logb);
18   dvar_vector pred=pbeta((dvar_vector)x,a,b);
19   nll=sum(square(y-pred));
```

admbex/pppbeta/pbeta.tpl

This simply does least squares estimation of pbeta compared to a large number (20000) of precomputed values. ADMB with hand written adjoint code used 1.7s to optimize this model, but ADMB with the new tiny_ad used only 0.8s. Both versions gave identical results. So the new version with simpler code, on hand written adjoint code and no repeated code, was more than twice as fast in a purely fixed effect model.

For a similar model including random effects, which then invokes up to third order derivatives, a similar difference is seen. The model was fitted in 8.5s with the new tiny_ad code in place, but in 18.0s with the old code base.

This example has focused on an example where the replaced function was the main part of the computations. In a real application the difference will be smaller, but the important thing is that the code base can be simplified and made more efficient at the same time by using tiny_ad.

7.6 Macro interface to tiny_ad in ADMB

The process described above for using tiny_ad within AD Model Builder for user defined functions is a bit complicated. The tiny_ad tool is the perfect tool for functions with few parameters, but requiring a long list of calculation. For such functions it can create efficient and exact derivatives — without writing long calculations to the stack. To make this feature easily accessible to users of ADMB three convenient macros were included in AD Model builder. The following code show how the simplest of the three is implemented:

```
#define TINYFUN1(FUN,par1)                                          \
                                                                    \
double FUN(double par1){                                            \
  return FUN<double>(par1);                                         \
}                                                                   \
                                                                    \
dvariable FUN(dvariable par1){                                      \
  typedef tiny_ad::variable<1,1> Float;                             \
  Float par1##_ (value(par1), 0);                                   \
  Float ans=FUN(par1##_);                                           \
  dvariable y;                                                      \
  value(y)=ans.value;                                               \
  tiny_vec<double, 1> der = ans.getDeriv();                         \
  AD_SET_DERIVATIVES1(y,par1,der[0]);                               \
  return y;                                                         \
}                                                                   \
                                                                    \
dvariable FUN(prevariable par1){                                    \
  return FUN((dvariable)par1);                                      \
}                                                                   \
                                                                    \
```

```
df1b2variable FUN(df1b2variable par1){                          \
  typedef tiny_ad::variable<3, 1> Float;                        \
  Float par1##_ (value(par1), 0);                               \
  Float ans=FUN(par1##_);                                  \
  double val=ans.value.value.value;                             \
  tiny_vec<double, 1> der1 = ans.value.value.getDeriv();          \
  tiny_vec<double, 1> der2 = ans.value.getDeriv();                \
  tiny_vec<double, 1> der3 = ans.getDeriv();                    \
                                          \
  df1b2variable tmp;                                       \
  double * xd=par1.get_u_dot();                                 \
  double * tmpd=tmp.get_u_dot();                           \
  *tmp.get_u()=val;                                   \
  for (unsigned int i=0;i<df1b2variable::nvar;i++){               \
    *tmpd++ = der1[0] * *xd++;                                  \
  }                                       \
  if (!df1b2_gradlist::no_derivatives){                         \
    f1b2gradlist->write_pass1(&par1,&tmp,der1[0],der2[0],der3[0]);    \
  }                                       \
  return tmp;                                       \
}
```

admb/src/tools99/tiny_wrap.hpp

The code shown adds a macro called `TINYFUN1`, which is to be called with a function name and the name of the one parameter the function is a function of. The macro then generates different versions of the code needed for evaluation with `double`, `dvariable`, and `df1b2variables`, and the fast derivatives are transferred correctly to ADMB's derivative structure.

Macros are added for functions of 1, 2, and 3 parameters and the macros are named `TINYFUN1`, `TINYFUN2`, and `TINYFUN3` respectively. The code for these macros are added to ADMB in the file: `admb/src/tools99/tiny_wrap.hpp`.

The first example of how to use these macros is to take a simple function and verify that the derivatives are right. Consider the following implementation of a Beverton-Holt model:

```
GLOBALS_SECTION
  #include <df1b2fun.h>
  template<class Float>
  Float logBH(Float ssb, Float loga, Float logb){
    return loga+log(ssb)-log(1+exp(logb)*ssb);
  }
  TINYFUN3(logBH,ssb,loga,logb);
  VECTORIZE3_ttt(logBH);

DATA_SECTION
  init_int n
  init_vector ssb(1,n)
  init_vector logR(1,n)
PARAMETER_SECTION
  init_number loga;
  init_number logb;
  init_number logSigma;
  sdreport_number sigmaSq;
  vector pred(1,n);
  objective_function_value nll;
PROCEDURE_SECTION
  sigmaSq=exp(2.0*logSigma);
  pred=logBH((dvar_vector)ssb,loga,logb);
  nll=0.5*(n*log(2*M_PI*sigmaSq)+sum(square(logR-pred))/sigmaSq);
```

admb/tests/tinyfun/tinyfun.tpl

The function is added as a template function of type `Float`, then the macro is called, which creates functions of all types needed in the following program. The final macro is to allow vectorized calls.

This code ran and gave exactly the same results (and derivatives) as a similar implementation where the function was implemented via `dvariables`. The run times were also the same, so even with a simple example as this no efficiency is lost by using tiny_ad.

A simple way to illustrate what happens if the complexity of the function increases is to modify the function in this example. Instead of simply computing the function value the function is changed to compute the same function value a high number of times and return the average. The modified function returns the same value with the same derivatives, but the internal computation is a much longer chain of operations. The code for the modified function is:

```
Float logBH(Float ssb, Float loga, Float logb){
  Float s=0;
  for(int i=1; i<=200000; ++i){s+=loga+log(ssb)-log(1+exp(logb)*ssb);}
  return s/200000;
}
```

If this code is used with tiny_ad in the example, then the full minimization is done in less than a minute and with almost no memory used. If the same function is declared with the `dvariable` type (not using tiny_ad) then the minimization takes 6 minutes and uses more than 3.2GB of memory. This illustrates that complex functions can be made much more efficient by tiny_ad without making the code more complicated and without hand written adjoint code.

7.7 Using R's integrate code in ADMB

The C-code for **R**'s integrate function is more than 2K lines and it would have been a big, difficult, and error-prone task to write the adjoint code up to 3rd order by hand. Having included tiny_ad in AD Model builder makes it possible to take such big chunks of code, modify it semi-automatically, and use it.

The modification of the code to use `Float` as the numeric type is described above. ADMB is extended with the following files

```
admb/src/tools99/integrate.cpp
admb/src/tools99/integrate.hpp
admb/src/tools99/integrate_wrap.cpp
```

The two first files are copied verbatim from TMB and the last file defines some macros to make it easier to use the integrate function in ADMB.

The macros can be called with a function to generate a new function with the same name prepended with `integrate`, so for instance for a function `F` a function called `integrateF` is generated. The generated function supports

all ADMB types (`double`, `dvariable`, and `df1b2variable`). The simplest of the macros are coded as:

```
#define INTEGRATE0(FUN)                                                 \
template<class Float>                                                   \
Float integrate##FUN(Float from, Float to){                             \
  FUN<Float> f;                                                         \
  Float ans = 0.0;                                                      \
  return integrate(f,from,to);                                          \
}                                                                       \
TINYFUN2(integrate##FUN,from,to)
```

Notice that the macro uses the already defined macro `TINYFUN2`, which takes care of defining all the different typed functions and writing the tiny_ad calculated derivatives into ADMB's derivative structure.

As a first example of its use consider calculating the integral:

$$\int_0^\infty \exp(-x^2)(\log(x))^2 dx$$

The follow example shows the syntax to be used in ADMB:

```
GLOBALS_SECTION
  #include<df1b2fun.h>

  template<class Type>
  struct F {
    typedef Type Scalar; // Required
    Type operator()(Type x){
      return exp(-pow(x,2))*pow(log(x),2);
    }
  };
  INTEGRATE0(F)

DATA_SECTION
  !! cout<<endl<<integrateF(0,INFINITY)<<endl;
  !! ad_exit(0);

PARAMETER_SECTION
  init_number dummy;
  objective_function_value obj

PROCEDURE_SECTION
```

admbex/integrate/intex.tpl

First the integrand must be defined in the special structure outlined, then the macro `INTEGRATE0` is called. Finally the newly defined function `integrateF(0,INFINITY)` can be called to evaluate the integral. Notice that `INFINITY` is allowed. The code runs and gives the correct result.

An integrand can depend on more variables than the one integrated over and the imported code from **R** with derivatives from tiny_ad can handle that. To allow one extra variable the macro `INTEGRATE1` is supplied and its use will be illustrated in the next example. Allowing two or more extra variables turned out to be problematic in ADMB. Writing externally calculated 3rd order derivatives back into ADMBs derivative structure is currently only supported for functions up to 3 variables. The code allowing it for exactly 3 variables is about 1000 lines (file `df32fun1.cpp`). Trying to duplicate the same setup for 4 or 5 variables would result in many times as many lines. Long term a more general solution should be developed.

The limit of only writing externally calculated 3rd order derivatives for functions of up to 3 variables sounds restrictive, and it is for the integral function, but it must be remembered all of the existing AD Model Builder has been developed within this constrain. Going through the code it appears that much less hand-written adjoint code is in the random effects part of ADMB (the part using 3rd derivatives) than in the fixed effects part.

An integral with one extra parameter and an upper and lower limit is a function of three parameters. If the lower limit is fixed e.g. to $-\infty$, then the integrand can have two additional parameters, so macros supporting that is defined. These macros are called `LTAIL0`, `LTAIL1`, and `LTAIL2`, because they calculate the lower tail of integrands with 0, 1, and 2, additional parameters respectively. The macros generate functions with the name of the integrand prepended with `ltail`.

To illustrate the syntax for additional parameters in the integrand consider observations from a Poisson process with a parametric intensity function $\lambda(t) = a(\sin(t\pi)+1)$ where the unknown parameter is a. The code for its likelihood involves an integral and can be specified as:

```
GLOBALS_SECTION
  #include<df1b2fun.h>

  template<class Type>
  struct L {
    typedef Type Scalar; // Required
    Type a;              // Parameters for integrand
    Type operator()(Type x){
      return a*(sin(x*M_PI)+1.0);
    }
  };
  INTEGRATE1(L,a);

DATA_SECTION
  init_int N;
  init_vector X(1,N);

PARAMETER_SECTION
  init_number logS
  sdreport_number S;
  objective_function_value obj

PROCEDURE_SECTION
  S=exp(logS);
  L<dvariable> l; l.a=S;
  obj=0.0;
  for(int i=1; i<=N;++i){
    obj -= log(l(X(i)));
  }
  obj+=integrateL(0.,10.,S);
```

admb/tests/poisp/poisp.tpl

Notice how the additional model parameter enters the defined integrand, and how it is assigned in the procedure section.

8 Discussion and future work

This project considered many different extensions to both TMB and ADMB. The selected extensions were implemented in both tools. It was first noticed

that the process of adding a simple custom function is a bit simpler in TMB compared to in ADMB (see section 3).

An important design difference between TMB and ADMB was identified (see section 2). ADMB recomputes the stack of operations at each function evaluation, whereas TMB uses a pre-setup computational graph. This has big consequences implementing special functions. Where ADMB can fairly easy import code from e.g. Numerical Recipes including conditional statements and loops depending on model parameters that is not possible for TMB. Hence it was easy for ADMB to get basic simple special functions working.

The problem was mostly solved in TMB by writing a neat efficient forward AD library tiny_ad (section 5). Forward mode AD does not need a computational graph to be stored, so it can follow conditional code.

In the process of writing these special functions it was identified that is very necessary to validate derivative code. This is simple for 1st derivatives in ADMB but difficult for 2nd and 3rd derivatives. A kludge to extract the 2nd derivatives from ADMB was derived (section 4.4). It is simple to validate derivatives from TMB.

The tiny_ad library turned out to be very useful for ADMB also, as it can be used to import big code chunks into ADMB (section 7), which would previously had been difficult, as it would either had been inefficient (by using AD for long calculations), or included a lot of difficult work with hand written adjoint code. In fact, all the functions ported to TMB can fairly simply be ported to ADMB. This could e.g. be relevant if one wanted to improve the precision of pbeta or the bessel functions from the OK, but not great versions from Numerical Recipes. Macros have been added to make this process simple, and care has been taken to make all ported functions standalone.

Finally it was noted that the procedure of writing externally computed derivatives back into ADMB is problematic for higher order derivatives of more than 3 model parameters. This made some extensions less flexible in the ADMB versions.

The conclusion is that TMB with the addition of tiny_ad is easier to extend than ADMB, but most of the extensions were useful to both tools. Collaborating between the two projects is the best way to validate and improving both tools.

Some final comments and ideas for future issues to improve:

- We managed to implement the suggested special functions pbeta, qbeta, besselK. The list was exapanded during the project to contain the remaining Bessel functions and the Tweedie density function.

- An automatic generator for p functions has been implemented via the the added a Gauss-Kronrod integrator. The integrator accepts `-INFINITY` as lower limit, so if the integrand is a probability density function then the integrator exactly returns the p function. Further the integrator is able to calculate other useful integrals.

- An automatic generator of q functions can in principle be constructed by combining an adaptive Gauss-Kronrod integrator with a Newton solver. The Gauss-Kronrod integrator is now added to both TMB and ADMB so it is feasible to implement an automatic q-generator also. However, in terms of numerical precision and performance it would be impossible to match specially taylored special functions in e.g. **R**. For this reason we have put this idea on hold for now.

- The proposal to implement an exact AD Hessian would require major re-strucuring for both TMB and ADMB. For a fixed effect model in ADMB an approach to get the AD hessian is derived in the section on checking derivatives. In general (for random effects models) an AD Hessian would be both slower and more memory consuming than the current approch (finite differences on the AD gradient). *If* one decides to implement an AD outer Hessian the most promising aproach might be to feed tiny_ad into the Laplace approximation and its gradient.

- The interface for adding tiny_ad derived functions has been added to ADMB so it would require only a small amount of work to add the ported dtweedie to ADMB. Similarly adding the TMB ported bessel functions to ADMB would give better presision and allow non-integer nu parameter.

- Dave's suggestion w.r.t. importance sampling was not tested, but judged to be equally easy to test in both TMB and ADMB.

- The forward sweep idea would be unfeasible to carry out in ADMB because ADMB doesn't have any examples on how to access the underlying computational graph. It would be possible - but not easy - to get it working in TMB. One could use the sparsity detection algorithm as starting point.

- Expanding the template distributions in ADMB to be similar to TMB's density namespace would not be meaningful. It would be fairly easy to add classes and method to give ADMB a similar calling syntax as in TMB, but the design of ADMB's random effects, where *all* calculations need to be structured in separable functions to preserve a sparse internal hessian (to get efficient code) would work within these encapsulated classes. E.g. a neat class to define a multivariate random

walk would result in slower code than a specifying the increments as separable functions in the template.

- Develop a general interface for writing externally calculated 2nd and 3rd order derivatives back into AD Model Builder. Currently only an interface for up to 3 model parameters exists.

- Supplement derivative checks with code coverage analysis so that we know that all branches of the special functions are tested by the derivatives checker. We might also consider adding tests up to order three.

- The simple scheme presented in this report computes all higher order derivatives without exploiting symmetry of these derivative arrays. tiny_ad is already equipped to address this issue to some extent. For instance the unique derivative indices up order 3 wrt 3 parameters is given by $\{(i,j,k) : 1 \leq i \leq j \leq k \leq 3\}$. This set of 10 derivative indices is well approximated by the cube $\{1,2\} \times \{1,3\} \times \{2,3\}$ (appropriately re-ordered) and the corners $(1,1,1)$, $(2,2,2)$ and $(3,3,3)$. All latter derivatives (cube and corners) can be directly obtained using tiny_ad and there's only one wasted index ($2 \times 2 \times 2 + 1 + 1 + 1 = 11$). Ideas along these lines might be able to speed up tiny_ad for higher order derivatives.

- It would be useful to lighten the burden of porting code to using tiny_ad - especially pointing out lines in the source code where derivatives start to get wrong. Compile time checks would be ideal but this is probably too difficult to achieve. The second best option could be a debug mode where the AD class includes finite difference checks up to any point in the program and triggers abort when finite difference starts to diverge from AD.

Acknowledgments

The work in this report is funded by JIMAR Project No. 6100327. In addition we would like to thank Johnoel Ancheta, David Fournier, and Dodie Lau for invaluable assistance with different parts of the project.

References:

B.M. Bell, J.V. Burke 2008. Algorithmic differentiation of implicit functions and optimal values. Advances in Automatic Differentiation. Editors: C. Bischof, H. Bucker, P. Hovland, U. Naumann, and J. Utke.

K. Kristensen, A. Nielsen, C.W. Berg, H. Skaug, B.M. Bell (2016). TMB: Automatic Differentiation and Laplace Approximation. Journal of Statistical Software, 70(5), 1-21.