To appear in *Optimization Methods & Software*
Vol. 00, No. 00, Month 20XX, 1–18

Efficient computation of derivatives for solving optimization problems in R and Python using SWIG-generated interfaces to ADOL-C

K. Kulshreshtha,*[a] S. H. K. Narayanan,[b] J. Bessac[b] and K. MacIntyre[c]

[a]*Universität Paderborn, Germany;* [b]*Argonne National Laboratory, Illinois, USA;* [c]*Northwestern University, Illinois, USA*

(*Received 00 Month 20XX; final version received 00 Month 20XX*)

Scripting languages are gaining acceptance because of their ease of use and value for rapid prototyping in many fields, including machine learning and statistics. In the context of algorithmic differentiation, however, the main development effort continues to be concentrated on traditional compiled languages such as Fortran and C/C++, whether source transformation tools or operator overloading tools. There is therefore a need for AD tools for computing derivatives efficiently within scripting languages. ADOL-C is an operator overloading-based C++ library that provides accurate first- and higher-order derivatives for applications in C++. SWIG is a preprocessor that uses the C/C++ header files to wrap the API of a library to be callable from scripting languages such as R and Python and several other high-level programming languages. Although every language has its caveats, the overall process of making the C/C++ API available via SWIG is the same for all scripting languages. After an initial effort required per language, because SWIG is an automated interface generator based on the library's actual header files, only minimal effort is required to maintain the scripting interface in sync with upstream developments in the original C/C++ library. In addition to achieving our original goal of creating an interface for R, we were able to generate an interface for Python that proved an order of magnitude faster the the previously implemented interface. This paper gives an overview of the interface generation process, the challenges we encountered with both scripting languages, and some numerical results to demonstrate both usefulness and efficiency.

Keywords: Algorithmic Differentiation, Autodiff, Machine Learning, Stochastic Optimization, Data Analysis, ADOL-C, Scripting, R, Python, SWIG

1. Introduction

Algorithmic differentiation (AD) [13] is a technique to compute derivatives of functions represented as evaluation procedures efficiently and accurately with round-off error within machine precision [12]. Various programming tools have been developed in the past to compute derivatives of programs written in high-level languages such as Fortran, C and C++. Such tools fall into two main categories. Some are built on a programming-language parser and compiler in order to output augmented program code that will, when executed, compute the desired derivatives along with the original output of the program. This technique is known as source transformation. Examples include Tapenade [14], ADIC2 [21], OpenAD [25] and TAF [10]. Others rely on the capability of object-oriented programming

*Corresponding author. Email: kshitij@math.upb.de

languages to overload operators and functions in order to perform additional tasks besides those that they would for builtin datatypes. Naturally, these tools are called operator-overloading tools. Prominent examples are ADOL-C [31], CppAD [5], dco/c++ [20], Adept [15] and CoDiPack [22]. The community website `http://www.autodiff.org` provides further information about tools and references on algorithmic differentiation.

High-level programming languages such as R and Python have become popular in fields such as machine learning, statistics, and data analysis. Many data analysis problems can be recast as familiar problems in nonlinear optimization. Indeed they often involve the optimization of a cost function; for instance, in the model-oriented framework of inferential statistics, parameters of the model are estimated most of the time by minimizing a cost function. Data assimilation is another example of problems involving the optimization of a cost function; see [16]. Various other fields of research do not involve this type of optimization although they still rely on the use of derivatives of objective functions. One example is sensitivity analysis [17, 23]. Too often, however, the derivatives of these functions are numerically approximated or coded by hand because a suitable AD tool is not available for the chosen programming language. The use of finite-difference approximation can result in reduced performance; hand coding on the other hand, requires time-consuming and error-prone maintenance of derivative computations. Furthermore, the burden of writing derivative code may deter users from using more complicated functions or more sophisticated solution algorithms.

R is a language and environment for statistical computing and graphics [34]. It is widely used in statistics and data mining. To obtain derivatives in R, one can use several non-native approaches, including the Template Model Builder system [39] and Ryacas [11]. However, none of these options support the differentiation of functions expressed as R programs, as would an AD tool for R. Attempts to develop such a tool include radx [32]. This tool can compute first- and second-order forward-mode derivatives of univariate functions, but it is no longer actively developed. Natively, inside R, the `numDeriv` package provides methods for calculating (usually) accurate numerical first- and second-order derivatives [35]. Accurate calculations are done by using Richardson's extrapolation, or, when applicable, a complex step derivative is available; a simple difference method is also provided. The `deriv` function from the `stats` package computes derivatives of simple expressions symbolically [33]. Since numerical finite-differences are not reliably accurate and cannot compute adjoints (see [13]), there is a need to provide derivatives within R using AD tools.

Python is a powerful object-oriented programming language that is widely used for rapid prototype development including in data science and for other *ad hoc* programming tasks. It is easily extensible by adding new modules implemented in a compiled language such as C or C++. Support for symbolic derivative computation as well as finite difference approximations in Python is provided by the `SymPy` package [38]. This package is a general symbolic manipulation tool, part of the SciPy family, which also includes NumPy and MatPlotLib [36]. Numerical approximation of derivatives is also provided by the `numdifftools` package [8]. A package called `theano` for symbolic mathematics on CPU/GPU combinations is also available [24]. Because Python is an object-oriented language, it lends itself easily to the ideas of operator overloading. Therefore, a few extensions to operator-overloading AD tools were written to provide derivatives for Python programs. Prominent among these are PyADOLC and PyCppAD [26–28], which are both wrappers around the C++ implementations of ADOL-C and CppAD using the Boost Library's `boost::python` interface. Another AD tool developed in Python using NumPy is AlgoPy [29]. A performance comparison of AlgoPy with PyADOLC, `numdifftools` and `theano` was done in [30] and the authors concluded that PyADOLC worked well for the applications under consider-

ation. Wrappers based on the Boost Library, however, require extensive manual updates whenever the interface of the underlying C++ library changes or introduces new features. Another drawback is that the Boost Library does not support similar interfaces for any other language of interest. Thus, there is, a need to provide an automated mechanism for interfacing ADOL-C to Python without requiring manual updates.

In this work we show how the AD tool ADOL-C can be interfaced with R and Python and be used to efficiently compute derivatives for optimization, statistics and machine learning problems expressed within those languages. We use the preprocessing and code generation tool SWIG [2, 3] to generate the interfaces in an automated manner. This obviates, to a large extent, the need for manual upkeep of the interfaces when updates to ADOL-C are made. We have tested the approach in R with an implementation of a simplified statistical model that produces surface wind speed prediction. We have also tested the approach with Python applications that implement machine learning algorithms using stochastic gradient descent and stochastic quasi-Newton.

The rest of the paper is organized as follows. In Section 2 we describe the preparation required for SWIG to generate interfaces for ADOL-C. In Section 3 we show how to write programs using the generated interfaces in both R and Python. In Section 4 we present the statistical and optimization applications in R and Python that use the generated interfaces. In Section 5 we summarize our conclusions and briefly describe future work in this direction.

2. SWIG Interface Generator

SWIG was identified based on our need for creating an interface between ADOL-C and the R and Python languages. Furthermore, we wanted to minimize the development time required to maintain the interface every time the underlying ADOL-C API is changed, whether new features are added to it or old ones removed or modified. SWIG is a software development tool that connects library APIs written in C and C++ with a variety of high-level programming languages. SWIG is typically used to parse C/C++ interfaces and generate the "glue code" required for the target languages to call into the C/C++ code [2–4, 37]. It can generate interfaces for many different languages including R, Python, TCL, and Octave. Important for this work, by using SWIG, an interface for ADOL-C can be generated automatically during the build process of the ADOL-C library. Once the interface generation with SWIG has been set up correctly for the intended target languages, the generated interface will automatically contain all the new features and updates from ADOL-C.

SWIG generates interfaces based on an input file (for example `mymodule.i`). This input file consists of SWIG macros. A simple module may be defined by using the input file in Figure 1(a). This will create a module with the name `mymodule` containing a wrapped interface in the scripting language of choice for the C/C++ API declared in the file that is given in the `%include` macro. In this case it is `<myheader.h>`. Actual C/C++ code is given between the macro delimiters `%{` and `%}`. This is the code required in order to compile and link the generated interface with the original C/C++ library.

Other macros of importance are `%ignore` and `%rename`. These will cause SWIG to ignore a certain C/C++ identifier name or rename it to something else for the generated interface, respectively. These features are useful if these names contain characters that are unsupported by the target language or include keywords or if wrapping these in the target language is not desirable at all. An example is the ADOL-C driver `function()` that evaluates the function value from trace (see [31]), which cannot be used in R because `function` is

```
%module mymodule
%{
#include <myheader.h>
%}

%include <myheader.h>
```

(a)

```
%module adolc
%{
#include <adolc/adolc.h>
%}
// many ignores and renames
%include "adolc_all.hpp"
      // generated by running
      // C++ preprocessor
```

(b)

Figure 1. (a) SWIG input file for an simple example module; (b) skeleton ADOL-C SWIG input file.

a keyword in the R language. Therefore, we use the macro `%rename (eval_func) function ;` when declaring the R interface. Similarly the ADOL-C library exports an API callable from Fortran, which is a copy of the C/C++ API. To export this to either Python or R is not desirable, and therefore we use macros such as `%ignore gradient_;`.

One caveat in using the `%include` macro is that unlike the C/C++ preprocessor it will read only the file named in the macro and will not recursively read any other files that are `#include`d inside this file. This feature is to prevent extraneous code from being generated that wraps any system APIs that were `#include`d in the C++ header file of a library. This poses a challenge for processing ADOL-C via SWIG, however, because the convenient header file `<adolc/adolc.h>` contains a large number of `#include` directives for subsidiary headers, as well as required system headers. Applying the C++ preprocessor directly, however, results in a file containing all the APIs from all the system headers as well as all the subsidiary headers. We do not need to wrap the system APIs for the target language, only the ADOL-C API. We therefore wrote a Python script that first excludes all the system headers from the ADOL-C headers and then runs the C++ preprocessor on it to temporarily produce a flat single header containing all ADOL-C APIs but no system APIs. This file is then `%include`d and processed with SWIG, and the generated sources are compiled.

2.1 Interface to the R language

Contrary to our expectations SWIG did not generate a working interface from the input file automatically. We encountered several difficulties when R was the chosen target language. First, several API functions in ADOL-C require in-place modification of arrays given as arguments. Generally, R programmers prefer to use the returned values from a function as the output instead of modifying the input arguments. However, in-place modification is the standard practice in C/C++ when multiple values need to be output. SWIG version 3.0.11 did not have the necessary mechanism for modifying the input arguments. We therefore needed to modify the SWIG sources themselves and introduced `%typemap(argout )` instructions as detailed in Section 11.5 of the SWIG documentation for considering 1D and 2D arrays as in-place modifiable arguments in R. At the time of this writing, these changes in the SWIG sources are not yet included in any official SWIG release or source repository but can be expected in the future.

Another difficulty is imposed by the structure of the R language. It does not allow for operator overloading in the same sense as C++ does. The C++ compiler generates additional code and lookup tables so that the correct operator or function can be chosen at runtime, based on context in any expression, that is, dynamic dispatch. In R, the programmer is responsible for checking the arguments to any overloaded function or operator and

dispatching the correct version. As a safeguard against inadvertent overloading of common mathematical operators, the SWIG-generated interface contains named functions for such operators; for example, `` `Plus` `` is generated in R for the C++ `operator +`. To utilize operator overloading correctly, we needed to modify the generated R source code to save the existing definition of `` `+` `` as shown in the first line of Figure 2(b) and dispatch it if the arguments are not the datatypes we expect for the ADOL-C library.

The generated code shown in Figure 2(a) works well if the mathematical operators take only scalar arguments. Further changes were required to handle a real application involving mathematical operators and function calls such as `sin()` and `cos()` and the list and matrix data structures in R.

The function `class` used by the generated code returns the type of the variable for scalar variables. For list and matrix variables, however, the function will return `list` and `matrix`, respectively, instead of the type of variables stored in the list or matrix. To overcome this difficulty, we implemented a function called `get_argtype()` to return the required variable type. Once the type is known, the generated code can correctly choose the C++ interface function that should be called for these objects. The interface functions, however, accept only scalar arguments. Therefore, instead of calling the interface function directly, we call a helper routine `adolc_operator_dispatch()` or `adolc_dispatch()` to dispatch the function call based on the data structure as well. For example, if `sin(arg)` is called where arg is a list, the dispatch function will iterate over elements of the list, call `sin()` for each element and add the result of each call to a new list that is then returned. A peculiarity of R prevents a similar approach for a matrix argument. R allows a matrix to be made up of only nonclass elements. Therefore, when `sin(arg)` is called where arg is a matrix, the dispatch function will iterate over elements of the matrix. call `sin()` for each element and place the result in a list containing only that result. The resulting matrix is composed of these lists containing only one element.

One drawback of this dispatch code, however, is that it makes the use of overloaded operators and functions rather slow during the creation of the trace for ADOL-C. But we were unable to find any alternative way to dispatch the functions and operators more efficiently.

2.2 Interface to Python and NumPy

Using the experience in creating an interface between R and ADOL-C, we were able to create an interface for ADOL-C and Python. SWIG has been used extensively to generate Python interfaces to C++ software, for example, in the FEniCS project [1, 18, 19]. There are differences, however, in the way Python and C++ deal with intermediate results, as well as how array data structures are handled in NumPy, the numerical mathematics module in Python. In C++, the assignment operator can be overloaded to account for the temporary intermediate `adub` objects that are allocated on the stack with short lifetimes. In Python, the assignment operator cannot be overloaded, and all objects must be allocated on the heap. This difficulty is straightforward to handle; we can simply `%ignore` the operators defined in C++ and write simple one-line wrappers that will return a heap-allocated `adub *` instead of a stack-allocated `adub` using a special typecast operator defined in ADOL-C. Python's own garbage collection mechanism deals with the resulting memory.

Arrays in Python are handled as `numpy.array` or `numpy.ndarray` objects. The NumPy authors have provided a SWIG input file `numpy.i` containing the specific typemaps for converting a C/C++ array argument given as a pointer and its size in a separate function argument. However, these work only if each such array has its own size right next to it.

```
`Plus` <- function(...) {
  argtypes <- mapply(class, list(...));
  argv <- list(...);
  argc <- length(argtypes);
# dispatch functions
  if (argc == 1) {
    if (extends(argtypes[1], '_p_badouble') &&
      length(argv[[1]]) == 1) {
      f <- Plus__SWIG_0;
    }
  } else if (argc == 2) {
    if (extends(argtypes[1], '_p_badouble') &&
      length(argv[[1]]) == 1 && extends(
     argtypes[2], '_p_badouble') && length(argv
     [[2]]) == 1) {
      f <- Plus__SWIG_1;
    } else {
     # many such cases ...
    }
  } else {
    stop("cannot find overloaded function for Plus
     with argtypes (",toString(argtypes),")");
  };
  f(...);
}
```

(a)

$\Longrightarrow$

```
`oldplus` <- `+`
`+` <- function(...) {
  argtypes <- get_argtypes(...);
  argv <- list(...);
  argc <- length(argtypes);
# dispatch functions
  if (argc == 1) {
    if (extends(argtypes[1], '_p_badouble')) {
      f <- Plus__SWIG_0;
    } else {
      f <- oldplus;
    }
  } else if (argc == 2) {
    if (extends(argtypes[1], '_p_badouble') &&
      extends(argtypes[2], '_p_badouble')) {
      f <- Plus__SWIG_1;
    } else if ( # other argument checks ) {
     # many such cases ...
    } else {
      f <- oldplus;
    }
  } else {
    f <- oldplus;
  };
  return(adolc_operator_dispatch(..., f=f));
}
```

(b)

Figure 2. (a) Generated R interface source; (b) manual modification for operator overloading.

In ADOL-C, most drivers take several array arguments with the size, either the number of dependents or number of independents, and these sizes are known from the trace. For all such functions to be able to interpret and return NumPy arrays properly, some simple wrappers are again required, with modified C++ signatures. A few such signatures are shown in Figure 3. These wrappers are written purely in C/C++, and the maintainer does not need to write any Python code or use any Python or NumPy API for C.

2.3 Maintenance effort for future API changes

The amount of effort to maintain the interfaces depends on the kind of API changes in the original C++ library. If the old API is simply removed or some simple driver functions are added in the header files of the library, these will be automatically taken care of by the SWIG modules. Similarly new classes and their member functions will be automatically wrapped in the SWIG modules, without needing any manual changes. Simple manual changes may be required, however, if there are name clashes in the target language, for example, using the `%rename` directive.

For compatibility with NumPy, overloaded operators or functions declared in C++ as **`friend`** to a certain class could require a redeclaration as a member function in the target language using the `%extend` directive, which enables one to declare additional member functions in any class, just for the current target language. These redeclarations are just one line of code for each original function or operator. If a new API function requires compatibility for arguments as NumPy arrays, a small wrapper with a NumPy-aware parameter declaration and memory allocation needs to be written in C++ by the library maintainer, similar to the ones that have already been written for the current API. The

```
int gradient(short tag, int n,
             double* x,    // size n
             double* g);   // size n
   // g preallocated
int jacobian(short tag, int m,
             int n,
             double* x,    // size n
             double** J);  // size m, n
   // J preallocated
int vec_jac(short tag, int m,
            int n,
            double* x,     // size n
            double* v,     // size m
            double* w);    // size n
   // w preallocated
```

(a)

```
void npy_gradient(short tag,
                  double* x, int n0,
                  double** g, int* n1);
  // *n1 = n, from trace
  // g allocated & returned to python
void npy_jacobian(short tag,
                  double* x, int n0,
                  double** J, int* m1,
                  int* n1);
  // *m1 = m && *n1 = n, from trace
  // J allocated & returned to python
void npy_vec_jac(short tag, int repeat,
                 double* x, int n0,
                 double* v, int m1,
                 double** w, int* n1);
  // *n1 = n, from trace
  // w allocated & returned to python
```

(b)

```
g = gradient(tag,x)
J = jacobian(tag,x)
w = jac_vec(tag,repeat,x,v)
```

(c)

Figure 3. (a) ADOL-C drivers with original signatures; (b) their NumPy array-aware wrapper signatures; (c) their usage in Python.

number of lines of code is proportional to the number of arguments in the new API function.

Major interfacing effort is needed only for inheritance of classes, when a class is designed to be a parent class in the target language interface. The code generation for R in SWIG does not currently support the key functionality called a *director class*. Therefore, the facility of external functions is currently still under development. Because we have adopted the use of custom helper routines for calling interface functions based on the type of variables and their shape, we must modify the generated R code every time the interface is regenerated. These changes are mechanical, however, and limited to small portions of the generated code. Indeed they may be done by using a small patchfile for the generated R code, with only this patchfile requiring maintenance for future API changes.

3. Using the Generated Interface

In this section we describe how to use the interfaces to ADOL-C in R and Python. Drivers for both R and Python are similar to ADOL-C drivers in C/C++. Minor changes are required to the code being differentiated, because of limitations in the interfaces, name clashes, and peculiarities in the underlying language.

3.1 Using ADOL-C-computed derivatives in R

Figure 4 shows the usage of ADOL-C in R to solve an optimization problem using the R optimization routine `optim`. This routine uses an argument `gr` to accept a function to return the gradient for the `BFGS`, `CG` and `L-BFGS-B` methods. If `gr` is `NULL`, a finite difference approximation will be used. ADOL-C is used to construct a function that will provide the gradients to `optim`. The first step is to load the ADOL-C library through the commands

```
source('init_adolc.R')
# Download the data
source('ExamplesOnData/DownloadData.r')

# Functions
n <- 12
cov_emp <- var(ws.nwp.JAN12[,1:n])
Lat <- c(Lat_grid0[1:n])
Long <- c(Long_grid0[1:n])

fr <- function(x,cov_emp,Lat,Long){
    n <- length(Lat)
    s0 <- 0
    for (k in 1:n){
      for (l in 1:n){
       s0 <- s0 + ( cov_emp[k,l] - ( (x[1] + x[2]*Lat[k] + x[3]*Long[k])* (x[1] + x[2]*Lat[l] + x
     [3]*Long[l])*exp(-x[4]*(Lat[k]-Lat[l])^2)*exp(-x[5]*(Long[k]-Long[l])^2) ) )^2   }}
    s <- s0
    s }

trace_on(1)
x <- c(adouble(0.1),adouble(0.1),adouble(0.1),adouble(0.1),adouble(0.1))
badouble_declareIndependent(x)
y <- fr(x,cov_emp=cov_emp,Lat=Lat,Long=Long)
badouble_declareDependent(y)
trace_off()

frADOLC <- function(x,cov_emp,Lat,Long) {
    xx <- x
    yy <- c(0.0)
    eval_func(1,1,5,xx,yy);
    yy     }

# Gradient of 'fr'
grrADOLC <- function(x,cov_emp,Lat,Long) {
    xx <- x
    yy <- c(0.0,0.0,0.0,0.0,0.0)
    gradient(1,5,xx,yy);
    yy     }

# General purpose optimization call using adolc gradient
res <- optim(par=c(.1,.1,.1,.1,.1), fn=frADOLC, gr=grrADOLC, cov_emp=cov_emp,Lat=Lat,
          Long=Long, method = "L-BFGS-B", control = list(type = 3, trace = 2))
```

Figure 4. Code for optimizing a function in R using ADOL-C derivatives.

written in the file `init_adolc.R` provided for this purpose. In this example, the function `fr` is the computation being differentiated. It is called by using input that is read from a data file. The differentiation process requires `fr` to be traced. Tracing is achieved by placing a call to `fr` between the usual `trace_on(1)` and `trace_off()` commands. Additionally, function calls `badouble_declareIndependent(x)` and `badouble_declareDependent(y)` are used to identify the independent and dependent variables, respectively. The function `frADOLC` is written to invoke the ADOL-C-provided routine `eval_func()` to efficiently compute the original function. The function `grrADOLC` is written to invoke the ADOL-C-provided routine `gradient()` and return the result. Then, `optim` is invoked with the argument expressions `fn=frrADOLC` and `gr=grrADOLC` to make `optim` use the ADOL-C-generated gradients.

```
import sys, os
sys.path.insert(0,
   os.path.abspath('../../ADOL-C/swig/python'
     ))
import adolc
```

(a)

```
adolc.trace_on(1)
adarray = adolc.as_adouble(array)
for item in iter(adarray):
  item.declareIndependent()

# Perform main computation that computes ay

ay.declareDependent()
adolc.trace_off()
```

(b)

Figure 5. (a) Importing the ADOL-C SWIG module in Python; (b) creating the trace of a function in Python.

```
secondSum = numpy.sum(numpy.cos(2.0*math.pi*array[:len(array)]))
secondSum += math.cos(2.0*math.pi*chromosome[c])
```

(a)

```
secondSum = numpy.sum(numpy.cos(2.0*math.pi*adarray[:len(adarray)]))
secondSum += adolc.cos(2.0*math.pi*adchromosome[c])
```

(b)

Figure 6. (a) Original Python computation. (b) Python computation with ADOL-C.

3.2 Using ADOL-C-computed derivatives in Python

The built SWIG module files `adolc.py` and `_adolc.so` must be in the user's Python system path so that the module may be imported. One may add any given path to this path using the code shown in Figure 5(a). Straightforward changes to a driver routine also are needed in order to create an ADOL-C trace of the required function, as shown in Figure 5(b). Then the driver may call any derivative computation routine, a few examples of which are shown in Figure 3(c).

A few other changes may be required in the computation routines. Consider the lines of code in Figure 6(a). Any mathematical functions called, such as `math.cos`, must be rewritten using the equivalent ADOL-C functions, in this case `adolc.cos`. However, if a function is performed on the whole array using `numpy`, such as `numpy.cos`, the code remains unchanged, because `numpy` can automatically call the overloaded functions from `adolc`. The resulting new code is shown in Figure 6(b).

4. Applications

In this section we present two applications in which the procedures described in Section 3 are applied to two testing examples. In Section 4.1 an application in R is introduced for a statistical model that was initially built for space-time prediction of surface wind speed. This application involves optimizations of a least squares cost function; several configurations of the optimization routine are compared and discussed. In Section 4.2 we introduce machine learning applications that use the stochastic gradient descent and stochastic quasi-Newton algorithms. For these applications, we compare the performance of PyADOLC derivative code and the SWIG interface to the latest ADOL-C version.

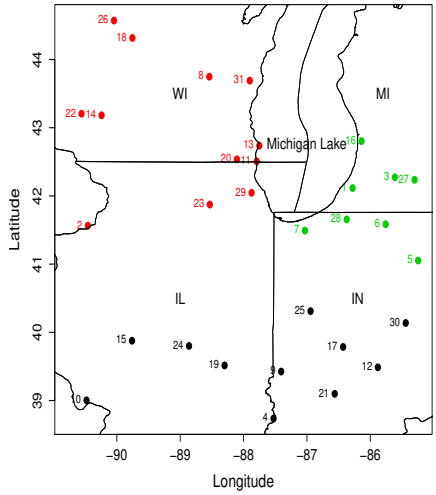


Figure 7. Map of the considered area (Midwest; visible are Lake Michigan and the states of Michigan, Illinois, Indiana, and Wisconsin): clusters are depicted with different colors: in black are the 11 stations of sub-region C_1, in red the 12 stations of C_2 and in green the 8 stations of C_3.

4.1 Space-time Gaussian process for wind speed in R

This application is motivated by a statistical model that fuses two datasets of atmospheric wind speed in order to provide a statistical prediction of wind speed in space and time [6]. The two datasets under consideration are numerical weather forecast model outputs and ground measurements of wind speed in the Great Lakes region (see Figure 7). In this context, these two datasets are assumed to be realizations of two space-time Gaussian processes. One purpose of the work in [6] is to specify the joint distribution of these two processes and then generate a prediction of wind speed from this distribution. The joint process has a Gaussian distribution whose mean and covariance are parameterized in space and time through hyperparameters. In [6], the statistical model is calibrated on the two datasets by maximum likelihood. Maximum likelihood estimation aims to maximize the probability of obtaining the studied data, when this probability results from the selected distribution model. Because of the sensitivity to initial conditions of the maximum likelihood procedure, the optimization of the likelihood is initialized with parameters obtained by a least squares estimation between empirical quantities (mean and covariance) and the proposed parametric ones.

In the present work, we consider simpler parametric shapes to focus more on the computational aspects than on the modeling ones. Furthermore, we focus on the subregion C_2 depicted in red in Figure 7 and on the month of January 2012. We simplify the framework proposed in [6] by considering a single dataset: the physical model outputs. Moreover, we fit a Gaussian process only along the spatial dimension; we then fit a space-time mean to the dataset. Gaussian processes are characterized by their first- and second-order moments; consequently we focus here on the mean and covariance structures.

The likelihood optimization requires computing the inverse of the covariance matrix, which in R can be done by using the function `solve`. This presents a problem, however, because the arguments to `solve` must be of `numeric`, `complex` or `logical` type. A check within `solve` throws an error if the arguments are of **`adouble`** type, which precludes the use of the ADOL-C interface. In the future, we will support the use of external functions to provide user-written derivative code for `solve`. For the present, in this work the proposed

models are fitted by a least squares estimation instead, which does not require the use of `solve`.

4.1.1 Spatial example

First, we focus only on modeling along spatial coordinates; the temporal dimension is not accounted for here. The parametric spatial mean and covariance are fitted on the empirical ones by an ordinary least squares estimation. The following parametric mean is proposed to be fitted on the empirical mean wind speed:

$$\mu(s) = \alpha_0 + \alpha_1 \mathrm{Lat(s)} + \alpha_2 \mathrm{Long(s)} + \alpha_3 \mathrm{Lat(s)}^2 + \alpha_4 \mathrm{Long(s)}^2, \tag{1}$$

where s denotes the spatial location and Lat and Long denote the latitude and longitude coordinates, respectively. The parameters α_0, α_1, α_2, α_3, and α_4 are estimated by least squares using `optim` and a cost function between the empirical mean and the proposed parametric one in (1); the cost function is similar to the one shown in Figure 4. For the optimization with the derivatives obtained from ADOL-C, we follow the procedure described in Section 3.1.

The parametric covariance is chosen as

$$\Sigma(s_i, s_j) = \sigma(s_i)\sigma(s_j) \exp(-\beta_3(\mathrm{Lat(s_i)} - \mathrm{Lat(s_j)})^2 - \beta_4(\mathrm{Long(s_i)} - \mathrm{Long(s_j)})^2), \tag{2}$$

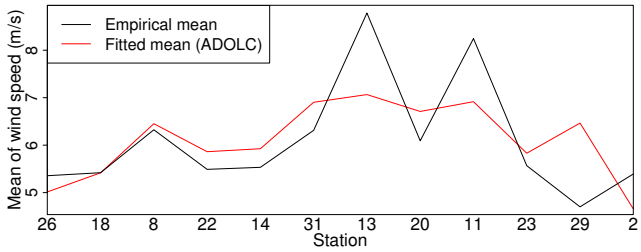
with $\sigma(s_i) = \beta_0 + \beta_1 \mathrm{Lat(s_i)} + \beta_2 \mathrm{Long(s_i)}$. Similarly β_0, β_1, β_2, β_3, and β_4 are the parameters that are estimated by a least squares procedure.

The fitted and empirical shapes are depicted in Figure 8. We expect the proposed parametric structures and the empirical one to match as much as possible in shape and intensity. We note that both the parametric mean and correlation structures capture a great part of the spatial features of the empirical mean and correlation. The fitted correlation displays higher dependencies (these are seen in [6]), but most of the spatial structure is captured.

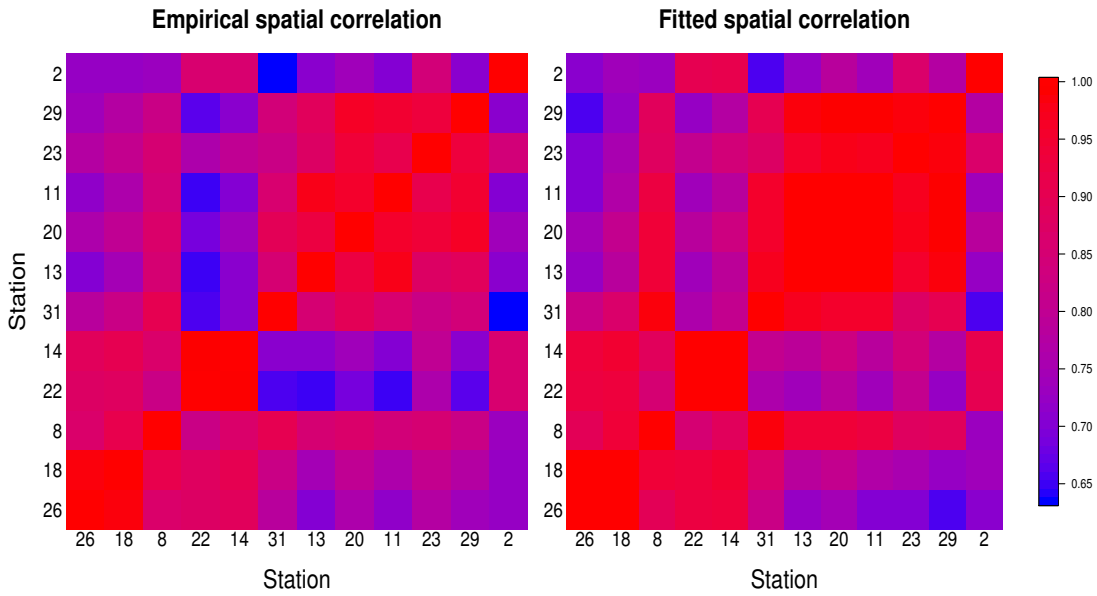
In this example, the optimization has been performed in four ways: with `optim`, which uses the internal finite differences derivatives; with `optim` when the user provides the gradient computed from `numDeriv`; with `optim` using derivatives from ADOL-C; and with `optim` using the analytical derivatives. The four methods lead to similar results in terms of quality of fitting, but the times to compute are significantly different, as seen in Table 1. In this example, we optimize along the vector $x = (\alpha_0, \alpha_1, \alpha_2, \alpha_3, \alpha_4)$ of dimension 5. In this case the size of the x does not increase; however, the size of the wind speed dataset increases in terms of spatial locations added. We observe that the optimization with ADOL-C derivatives exhibits a time increase of 425% when the amount of data doubles; however, the method with the internal finite differences derivatives shows an increase in the time of only 60%. The two other methods have a time increase of more than 300%. In absolute terms, however, ADOL-C is the fastest approach. Table 1 does not report, however, the time for tracing the cost function for ADOL-C. Table 2 reports this with increasing numbers of stations. These values are higher than any of the computation times in Table 1 because of inefficiency caused by loop iterations in the manually written dispatch functions, described in Section 2.1.

4.1.2 Space-time example

We now fit the parametric space-time mean proposed in [6, Equation (14)] by a least squares estimation using the derivatives from ADOL-C and the procedure described in



(a)



(b)

Figure 8. (a) Empirical (black) and fitted (red) mean of the wind speed at the locations of the subregion C2 estimated in January 2012; (b) empirical (left) and fitted (right) parametric spatial correlation of the wind speed.

Table 1. Time in s for the estimating the parameters of the parametric spatial mean depicted in panel (a) of Figure 8. The number of parameters to be estimated is constant and equals 5; the number of stations studied increases.

# of Stations	ADOL-C	`numDeriv`	Internal FD	Analytical
6	0.0017	0.004	0.005	0.001
8	0.0016	0.005	0.006	0.001
10	0.0018	0.006	0.007	0.001
12	0.0028	0.020	0.008	0.004

Table 2. Time for tracing the cost function used to fit the parametric spatial mean depicted in Figure 8. The number of parameters to be estimated is constant and equals 5; the number of stations studied increases.

# of Stations	6	8	10	12
Time (s)	0.028	0.034	0.040	0.049

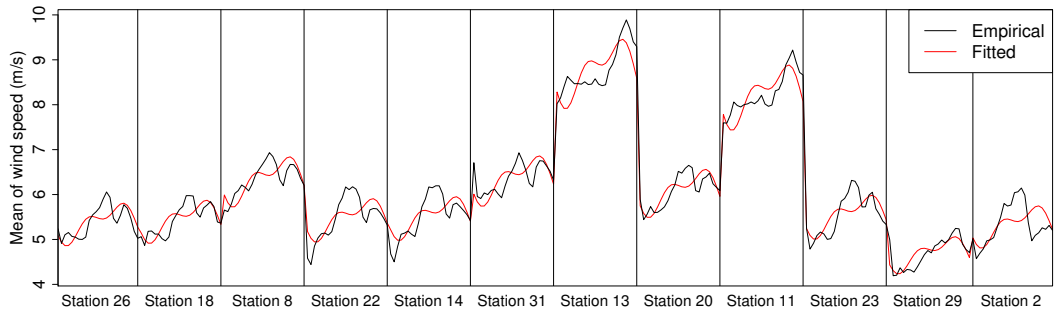


Figure 9. Empirical (black) and fitted (red) parametric mean of wind speed at each hour of a day and at each station in the subregion C2 in January. Vertical lines separate each station. Within each of these windows, each hour of the day is considered.

Table 3. Time in s for estimating the parameters of the parametric space-time mean depicted in Figure 9. The number of parameters to be estimated (left column) increases with the number of stations studied (in parentheses in the left column).

`length(x)` (# of Station)	ADOL-C	`numDeriv`	Internal FD	Analytical
13 (6)	0.0017	0.32	0.47	0.03
15 (8)	0.0018	0.38	0.58	0.03
17 (10)	0.0018	0.56	0.84	0.04
19 (12)	0.0028	0.85	1.29	0.05

Table 4. Time in s for tracing the cost function used to fit the parametric space-time mean depicted in Figure 9. The number of parameters to be estimated (top row) increases with the number of stations studied (in parentheses in the top row).

`length(x)` (# station)	13 (6)	15 (8)	17 (10)	19 (12)
Run time (s)	0.03	0.03	1.62	1.99

Section 3.1. The parametric structure is simplified here for practical purposes and does not account for the land use, namely, the parameter that describes the nature of the land for a considered grid point. We show visually in Figure 9 the quality of the fitting. We see that both curves match well, indicating that the proposed parametric mean captures most of the space-time structure of the mean of the wind speed.

Similar to the preceding spatial example, we compared the optimization process with different gradient inputs. The four optimizations led to similar results in terms of quality of the fitted mean; however, the run times were significantly different. In Table 3, we show the compute time when these least squares optimizations are performed. The proposed parametric shape has a parameter specific to the spatial location. Consequently, increasing the size of the dataset necessitates increasing the size of the vector to optimize along, contrary to Table 1. We notice here that the compute times with ADOL-C derivatives are considerably smaller than those of `numDeriv` and internal derivatives. The rate of increase in time with ADOL-C is also lower that of the other two methods. Indeed, as the size of x doubles, `optim` with its internal derivatives shows an increase in time of 174%, `optim` with derivatives from `numDeriv` is 166% slower and the method with the analytical gradient reveals an increase in time of 67%, whereas the time with derivatives from ADOL-C increases by 65%. Table 4 reports the time taken for tracing the cost function.

4.2 Stochastic gradient descent and stochastic quasi-Newton in Python

Models in machine learning applications are trained by using optimization algorithms. In some applications, a full-batch (sample average approximation) approach is feasible and appropriate. In most large-scale learning problems, however, one must employ stochastic approximation algorithms that update the prediction model based on a relatively small subset of the training data [9]. Stochastic gradient descent (SGD) and stochastic quasi-Newton (SQN) are two such optimization algorithms.

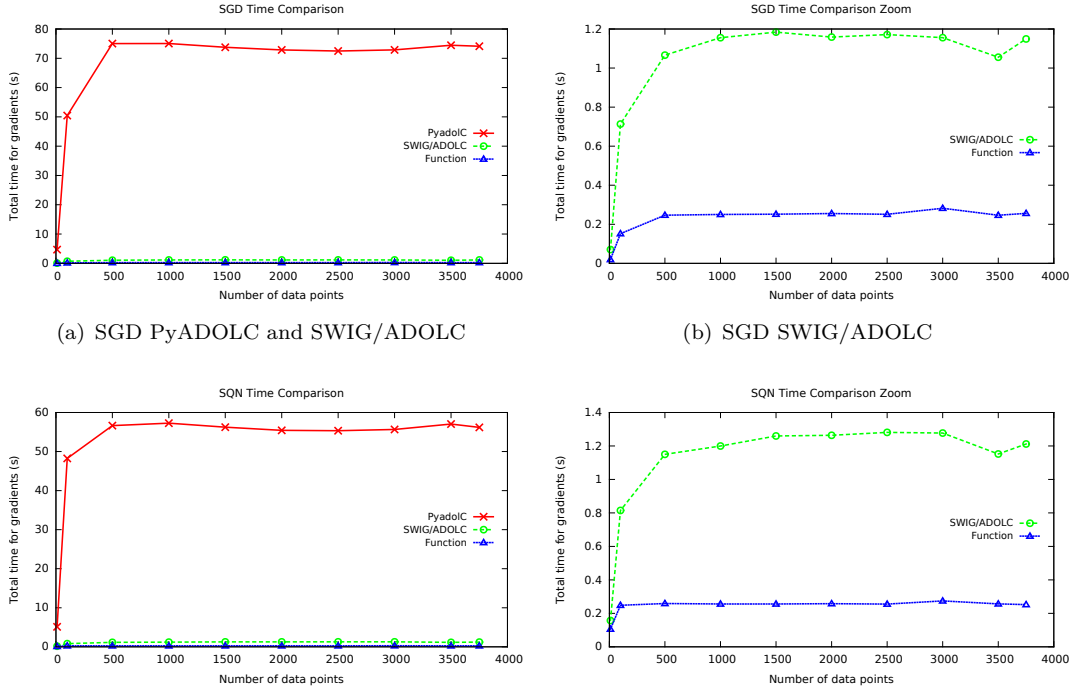
SGD is a machine learning algorithm that aims to find the global optimum of an objective function by updating a weight vector following the negative gradient of the objective. SGD normally updates using only one data point, but it can also use a small batch of sample data points (often called mini-batch gradient descent). The algorithm for SGD runs for a user-supplied number of iterations, on each iteration finding the gradient with respect to the current weights and a randomly selected batch of the data and then updating the weights according to this gradient and a dynamic learning rate, which gets smaller as the algorithm converges to the optimal value.

The SQN method is similar to SGD, also incorporating curvature estimates to increase the rate of convergence. The Hessian-vector product can be used to compute these curvature estimates. The SQN method has four main steps. First, for every iteration specified by the user, a random sample of the correct size from the test set is chosen. Second, as in SGD, the stochastic gradient is calculated. Third, the weight vector is updated using the curvature estimate, if it has already been calculated, to alter the direction of the update if necessary. Fourth, every certain number of iterations, the correction pairs for the curvature estimate are updated with the Hessian-vector product of the current weights, another batch of data, and the change in the weight vector.

We have used an implementation [7] of the SGD and SQN approaches outlined in [9]. The code already computed fast and exact analytic gradients for the loss functions. We differentiated the loss functions in the code using PyADOL-C and the ADOL-C interfaces generated using SWIG (SWIG/ADOL-C). We compared their performance by measuring the total amount of time for calculating every gradient and Hessian-vector product the code needs to use. We tested the times over a wide variety of data subsets, using different numbers of samples and attributes for those samples. The elements of each subset used in each step were taken from a pregenerated random set of sample indices, so that the objective found by each function and the times that were recorded could be reasonably compared.

Figure 10(a) compares the performance of the application using SGD where the derivatives are obtained by using either PyADOLC or SWIG/ADOL-C. The time taken to compute the original function being differentiated are shown for comparison. For clarity, the time taken for derivative computation using SWIG/ADOL-C and the time taken to compute the original function being differentiated are shown in Figure 10(b). Clearly SWIG/ADOL-C is significantly faster than PyADOLC. We attribute some of this performance gain to the new features in ADOL-C that obviate the need for retracing the code being differentiated. However, the glue code generated by SWIG and the handwritten `boost::python` code are also vastly different. Similar behavior is observed in Figures 10(c) and 10(d) when the application employs SQN.

Next, we varied the number of attributes in the problem. Figures 11(a) and 11(c) show the large time difference for computing gradients using PyADOLC and SWIG/ADOL-C for SGD and SQN, respectively. For clarity, the SWIG/ADOL-C results are shown separately in Figures 11(b) and 11(d).



(a) SGD PyADOLC and SWIG/ADOLC

(b) SGD SWIG/ADOLC

(c) SQN PyADOLC and SWIG/ADOLC

(d) SQN SWIG/ADOLC

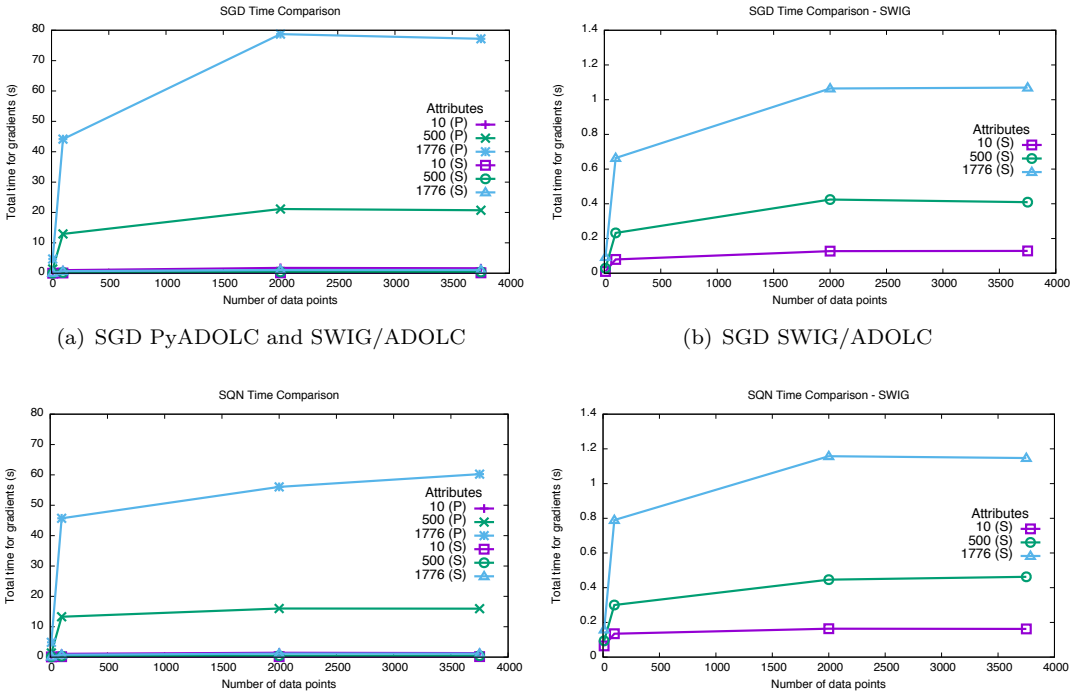
Figure 10. Performance results for SQN and SGD using PyADOLC and SWIG/ADOL-C using 1,776 attributes. The x-axis shows how many data points were used in the subset tested. The dashed lines are results using the SWIG/ADOLC, the solid lines with PyADOLC.

5. Conclusions and Further Work

We have successfully generated interfaces for the ADOL-C library, an operator overloading AD tool, for the scripting languages R and Python. The software development tool SWIG was used to generated the interface in a mostly automated manner. SWIG preprocesses C/C++ header files and generates C++ glue code between the target language and the original library, which is then compiled and linked to the original library. Some initial preparation of input files and massaging of the generated code for our purposes was required, but these are fairly mechanical. Thus, only minimal maintenance effort is required to keep these scripting interfaces to the library current and in sync with the developments in the C/C++ code of the library.

The use of ADOL-C in R to compute derivatives for an optimization problem leads to a huge performance improvement in the optimization routine over the default method that uses numerical approximation. Indeed the ADOL-C-computed derivatives perform just as well as manually programmed derivative code, which is intractable and error prone as the problems become more complex and also if higher-order derivatives are needed. The tracing of the cost-function, although, remains slow due to limitations in the language. This is however in most cases a one time process, and traces can be made persistent over multiple sessions.

Similarly, the use of ADOL-C in Python to generate derivatives for machine learning applications leads to a performance improvement over the existing Python interface. Some of this can be attributed to the use of new features in the ADOL-C library, which are not available in PyADOLC, which uses a somewhat older version of the C++ library because



(a) SGD PyADOLC and SWIG/ADOLC

(b) SGD SWIG/ADOLC

(c) SQN PyADOLC and SWIG/ADOLC

(d) SQN SWIG/ADOLC

Figure 11. Performance results for SQN and SGD using PyADOLC and SWIG/ADOL-C using different numbers of attributes. The different colors represent how many attributes from the data set were used in that test, as described in the legend, and the x-axis shows how many data points were used in the subset tested.

of the requirement for updating the `boost::python`-based code manually in PyADOL-C to track the changes in the underlying C/C++ code.

Because of limitations in the current version of SWIG, however,some features of ADOL-C are not yet available in the scripting languages. We will continue to develop the interface for R to support user-provided derivative code for functions that cannot be overloaded. Such functions are supported in the C++ version of the ADOL-C library using the so-called external functions. The major development effort in this regard will be to manually write glue code to connect the external function's API to the R interface. We will also look at other high-level languages supported by SWIG and provide interfaces to ADOL-C in these languages for use by interesting applications being developed in those languages.

Acknowledgments

We thank Prasanna Balaprakash, Paul Hovland, Joseph Wang, and Richard Beare for their suggestions.

Funding

This work was funded in part by a grant from DAAD Project Based Personnel Exchange Programme and by support from the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357.

References

[1] M.S. Alnæs, J. Blechta, J. Hake, A. Johansson, B. Kehlet, A. Logg, C. Richardson, J. Ring, M.E. Rognes, and G.N. Wells, *The FEniCS Project Version 1.5*, Archive of Numerical Software 3 (2015).

[2] D.M. Beazley, *SWIG : An Easy to Use Tool for Integrating Scripting Languages with C and C++*, in the 4th Annual Tcl/Tk Workshop, Monterey, CA (1996), Available at `http://swig.org/papers/Tcl96/tcl96.html`.

[3] D.M. Beazley, *Using SWIG to control, prototype, and debug C programs with Python*, in the 4th International Python Conference, Livermore, CA (1996), Available at `http://swig.org/papers/Py96/python96.html`.

[4] D.M. Beazley and P.S. Lomdahl, *Feeding a Large-Scale Physics Application to Python*, in the 6th International Python Conference, San Jose, CA (1997), Available at `http://swig.org/papers/Py97/beazley.html`.

[5] B.M. Bell, *CppAD: A package for differentiation of C++ algorithms*, http://www.coin-or.org/CppAD/Doc/cppad.htm.

[6] J. Bessac, E.M. Constantinescu, and M. Anitescu, *Stochastic simulation of predictive space-time scenarios of wind speed using observations and physical models*, arXiv preprint arXiv:1511.09416 (2016).

[7] A. Bou-Rabee, https://github.com/nitromannitol/stochastic-quasi-newton.

[8] P.A. Brodtkorb, *Numdifftools*, `https://pypi.python.org/pypi/Numdifftools`.

[9] R.H. Byrd, S.L. Hansen, J. Nocedal, and Y. Singer, *A stochastic quasi-Newton method for large-scale optimization*, SIAM Journal on Optimization 26 (2016), pp. 1008–1031, Available at `http://dx.doi.org/10.1137/140954362`.

[10] R. Giering and T. Kaminski, *Applying TAF to generate efficient derivative code of Fortran 77-95 programs*, in *Proceedings of GAMM 2002*.

[11] R. Goedman, G. Grothendieck, S. Højsgaard, and A. Pinkus, *Ryacas - An R interface to the YACAS computer algebra system*, `http://cran.r-project.org/web/packages/Ryacas/vignettes/Ryacas.pdf` (2014).

[12] A. Griewank, K. Kulshreshtha, and A. Walther, *On the numerical stability of algorithmic differentiation*, Computing 94 (2012), pp. 125–149, doi:10.1007/s00607-011-0162-z.

[13] A. Griewank and A. Walther, *Principles and Techniques of Algorithmic Differentiation, Second Edition*, SIAM, 2008.

[14] L. Hascoet and V. Pascual, *The Tapenade automatic differentiation tool: Principles, model, and specification*, ACM Trans. Math. Softw. 39 (2013), pp. 20:1–20:43.

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

[16] E. Kalnay, *Atmospheric Modeling, Data Assimilation and Predictability*, Cambridge University Press, 2003.

[17] M. Lamboni, B. Iooss, A.L. Popelin, and F. Gamboa, *Derivative-based global sensitivity measures: General links with Sobol' indices and numerical tests*, Mathematics and Computers in Simulation 87 (2013), pp. 45–54.

[18] A. Logg and G.N. Wells, *DOLFIN: Automated Finite Element Computing*, ACM Transactions on Mathematical Software 37 (2010), doi:10.1145/1731022.1731030.

[19] A. Logg, G.N. Wells, and J. Hake, *DOLFIN: A C++/Python finite element library*, in *Automated Solution of Differential Equations by the Finite Element Method*, Lecture Notes in Computational Science and Engineering, Vol. 84, chap. 10, Springer, 2012.

[20] J. Lotz, K. Leppkes, and U. Naumann, *dco/c++ - Derivative Code by Overloading in C++*, Tech. Rep. 2011,06, Aachener Informatik-Berichte, 2011.

[21] S.H.K. Narayanan, B. Norris, and B. Winnicka, *ADIC2: Development of a component source transformation system for differentiating C and C++* , Procedia Computer Science 1 (2010), pp. 1845–1853, ICCS 2010.

[22] M. Sagebaum, T. Albring, and N.R. Gauger, *CoDiPack*, http://www.scicomp.uni-kl.de/software/-codi/.

[23] A. Saltelli, M. Ratto, T. Andres, F. Campolongo, J. Cariboni, D. Gatelli, M. Saisana, and S. Tarantola, *Global Sensitivity Analysis: The Primer*, Wiley, 2008.

[24] Theano Development Team, *Theano: A Python framework for fast computation of mathematical expressions*, arXiv e-prints abs/1605.02688 (2016), Available at `http://arxiv.org/abs/1605.02688`.

[25] J. Utke, U. Naumann, M. Fagan, , N. Tallent, M. Strout, P. Heimbach, C. Hill, and C. Wunsch,

OpenAD/F: A modular open-source tool for automatic differentiation of Fortran codes, ACM Trans. Math. Softw. 34 (2008), pp. 18:1–18:36.

[26] S.F. Walter, *PyADOLC*, `https://github.com/b45ch1/pyadolc`.

[27] S.F. Walter, *AD in Python with Application in Science and Engineering*, in Eighth EuroAD Workshop, The Numerical Algorithms Group, Oxford, UK (2009).

[28] S.F. Walter, *Algorithmic Differentiation in Python with PYADOLC and PYCPPAD*, in EuroScipy Conference, Leipzig, Germany (2009).

[29] S.F. Walter and L. Lehmann, *Algorithmic differentiation in Python with AlgoPy*, J. Comput. Sci. 4 (2013), pp. 334—344.

[30] S.F. Walter, A. Schmidt, and S. Körkel, *Adjoint-based optimization of experimental designs with many control variables*, Journal of Process Control 24 (2014), pp. 1504–1515.

[31] A. Walther and A. Griewank, *Getting started with ADOL-C*, in *Combinatorial Scientific Computing*, U. Naumann and O. Schenk, eds., Chapman-Hall, 2012, pp. 181–202.

[32] *Automatic Differentiation in R*, `https://github.com/quantumelixir/radx` (2014).

[33] *Symbolic and algorithmic derivatives of simple expressions*, `https://stat.ethz.ch/R-manual/R-devel/library/stats/html/deriv.html`.

[34] *The R Project for Statistical Computing*, `http://www.r-project.org/`.

[35] *Package numDeriv*, `https://cran.r-project.org/web/packages/numDeriv/numDeriv.pdf`.

[36] *Scientific Computing Tools for Python – SciPy.org*, `http://scipy.org/about.html`.

[37] *SWIG website*, `http://www.swig.org/`.

[38] *Calculus – SymPy 1.0 Documentation*, `http://docs.sympy.org/latest/tutorial/calculus.html`.

[39] *ADMB - Template Model Builder*, `http://www.admb-project.org/developers/tmb` (2014).