Technische Universität Berlin

Institut für Mathematik

Algorithmic Sensitivity Analysis in the Climate Model CLIMBER 2

Thomas Slawig

Technical Report 29-2006

Preprint-Reihe des Instituts für Mathematik
Technische Universität Berlin

Algorithmic Sensitivity Analysis in the Climate Model CLIMBER 2

Thomas Slawig

Contents

1 Introduction

This report summarizes the results of the project ALGOSENSE performed at the Institut für Mathematik, Technische Universität Berlin, from July 2001 to June 2002.

Aim of the project was to analyze the applicability of tools for Algorithmic (or Automatic) Differentiation (AD) to two climate models developed at the Potsdam Institute for Climate Impact Research (PIK). These were the so-called Box Model, a small model of the North Atlantic stream, and the more complex model CLIMBER 2 which is a so-called *model of intermediate complexity* consisting of atmosphere, ocean, ice, and vegetation components. Applications that are considered start from pure sensitivity calculations over uncertainty estimations to optimization runs. First and higher order derivatives are of interest.

The outline of this report is the following: In the next section we describe the basic tools and techniques of Algorithmic Differentiation. The following two sections deal with the two models studied in this project. In each of them the corresponding model and its special features important for Algorithmic Differentiation are briefly introduced. Then the used Algorithmic Differentiation tools and technical details of the AD process are presented. At last numerical results are given. Further emphasis is put on the necessary code preparations to apply the AD tools. The last section of the report gives a summary and deals with the perspectives and opportunities of the application of Algorithmic Differentiation to these and maybe other climate models.

Some information regarding AD tools are outdated, they reflect the standard of 2002.

Acknowledgments

Part of the work was financed by PIK in the project ALGOSENSE. Computations were performed by Sven Peth and Sebastian Schlenkrich. Some scripts were written by Cezar Ionescu.

We would like to thank the researchers at PIK for the interesting theme of investigation, their support and collaboration during the last year, specifically Prof. Dr. Stefan Rahmstorf, Prof. Dr. Rupert Klein, Dr. Hermann Held, and Dr. Victor Brovkin.

We are indebted to Dr. Ralf Giering and Dr. Thomas Kaminski from FastOpt GbR Hamburg for their extensive support in using Tamc and Taf.

2 Basics of Algorithmic Differentiation

Algorithmic (or Automatic) Differentiation (AD) is a software technology that provides a means to compute the derivative of a function given in the form of a computer programme.

One characteristic advantage of an AD-generated derivative is that it is *exact*, i.e. it does not incorporate any approximation errors. This is an important difference compared to other ways of computing (or approximating) derivatives, for example finite difference (FD) calculations.

In this section we first describe the basic concepts of AD. Then we explain the difference between the two basic modes (forward and reverse). The decision between them is crucial for efficient derivative computations, specifically for large-scale applications as complex climate models. It basically depends on the relation of the number input and output variables. Then we describe the two ways of code generation used by the different AD tools (source transformation and operator overloading). This choice affects the efficiency of the code as well. We continue with an overview of available AD tools and end this section with the consequences of these considerations on the project.

2.1 Basic concept

In this section we want to briefly describe the basic idea of Algorithmic Differentiation: Let us consider a function

$$y = F(x), \quad F : \mathbb{R}^n \to \mathbb{R}^m,$$

realized in a computer programme with

- a vector of independent (or input) variables $x \in \mathbb{R}^n$
- and a vector of dependent (or output) variables $y \in \mathbb{R}^m$.

The two vectors x and y shall incorporate the relevant input and output variables, respectively, in the sense that we are interested in computing the derivative of y with respect to x. The function F may depend on other quantities and may produce additional output, but since in this context these additional quantities are not relevant we skip them.

The function F can be represented as a concatenation

$$F = F_k \circ \ldots \circ F_2 \circ F_1$$

of k elementary intrinsic functions and operators F_i (e.g. $\sin, \cos, +, -, *$ etc.) of the used programming language. Note that k usually is very large.

Each F_i can be differentiated *exactly* by standard rules of calculus. Using the chain rule of differentiation the derivative of the concatenated function F then can be written as the matrix product

$$y' := \frac{dF}{dx}(x) = \frac{dF_k}{dx_{k-1}}(x_{k-1}) \cdots \frac{dF_2}{dx_1}(x_1)\,\frac{dF_1}{dx}(x) \tag{2.1}$$

where the $x_i := F_i(x_{i-1}), (x_0 = x, x_k = y)$ denote intermediate variables. All these x_i lying on the path from x to y are thus needed to compute y' and therefore called *active*.

2.2 Forward and reverse mode

There are two alternative ways of evaluating (2.1), influencing fundamentally the performance of the derivative computation: It is possible to propagate the derivatives

- from F_1 to F_k in so-called *forward mode* or
- backwards from F_k to F_1 in *reverse mode.*

In the AD context the derivative code of a model generated by forward mode is called a *tangent linear model*, whereas one obtained by reverse mode is called an *adjoint model.*

The most important consequence can be easily seen by analyzing the dimensions of the matrices $\dfrac{dF_i}{dx_{i-1}}$ and their intermediate products in (2.1). It concerns the efficiency with respect to cputime and storage of the generated derivative code:

- **Forward mode** results in code whose **computational effort is proportional to the number** n **of input values**, compared to the time for a function evaluation. If n is large the proportional factor often is smaller than 1 since compiler optimization may take advantage of common subexpressions. Thus forward mode is preferable if $n < m$ or at least $n \not\gg m$.
- **Reverse mode** results in code whose **computational effort is proportional to the number** n **of output values.** Thus it is preferable in the opposite case, i.e. $n \ll m$.

  To estimate the computational effort of the reverse mode it has to be considered that either storage or recomputation (or a combination of both) of the intermediate values x_i becomes necessary. This can be easily seen in (2.1), too. For models with a huge number of time steps this fact has to be considered to obtain the desired efficiency. Optimal combination of both result in the placement of so-called checkpoints where intermediate values are stored and used for partial recomputations. These checkpointing strategies are current topics of research in the AD and control community.

  Moreover the reverse mode requires an invertible control flow of the programme, i.e. jump statements as e.g. `goto` statements have to be avoided or replaced.

Note that the effort of generating the derivative code itself is negligible. These considerations result in the following

Rule of thumb for applying Algorithmic Differentiation:

- Start with forward mode.
- If the relation $n \gg m$ is given and forward mode-generated code becomes too big (in the sense that the executable needs to much storage) or too slow, apply reverse mode and if necessary use checkpoint schemes.

Let us give two typical examples: If AD derivatives shall be used for optimization of a single-valued functional with a high number of control parameters the reverse mode is appropriate. If a model time step shall be linearized forward mode is sufficient.

2.3 Source transformation vs. Operator overloading

There are two ways of differentiating elementary functions and operators of a programming language, i.e. the F_i in (2.1). They also influence the efficiency of the generated AD code:

- In the so-called *source transformation* method additional variables
$$x_i' := \frac{dx_i}{dx}$$
are introduced. Based on the chain rule
$$\frac{dx_i}{dx} = \frac{dF_i}{dx_{i-1}}(x_{i-1})\,\frac{dx_{i-1}}{dx}$$
a corresponding derivative statement
$$x_i' = \frac{dF_i}{dx_{i-1}}(x_{i-1})\,x_{i-1}'$$
for every statement
$$x_i = F_i(x_{i-1})$$
is explicitly added in the source code. In this way a new source code computing both y and y' is generated.

  Second derivatives can be computed by applying the tool twice, usually first in reverse and then in forward mode.

- In the second approach based on *Operator Overloading* a new data type for the pairs (x_i, x_i') of values and derivatives is introduced. A library provides the implementations of the elementary functions and operators for this data type. Hence this approach does not generate new source code, but the original one is run with all active variables declared with a different type and linked with the AD library. During the run of the differentiated model a tape is generated which contains the values and functions that were used. Afterwards this tape is evaluated to compute the derivative, if desired also of higher order.

  Crucial points for efficiency are that overloaded operations are more difficult to optimize for a compiler, and that the tape generation takes time and storage. On the other hand once the tape is generated multiple evaluations (for example for higher order derivatives or at different input values) are cheap, if they do not lie on a different control flow branch of the code.

For more details on technical issues of AD see e.g.[GK98],[Gri00].

2.4 The seed matrix

Computing the object
$$\frac{dF}{dx} = \left(\frac{\partial F}{\partial x_i}\right)_{i=1,\dots,n}$$
for $x = (x_i)_{i=1,\dots,n} \in \mathbb{R}^n$ can be – depending on the dimension n – a quite time-consuming process. Sometimes it may not be necessary or desirable to compute all these partial derivatives. To avoid redundant computation and storage the so-called *seed matrix* $S \in \mathbb{R}^{n\times k}$ is used. Its entries
$$s_{ij}, \quad i = 1,\dots,n, j = 1,\dots,k,$$
and the dimension $k \leq n$ may be defined arbitrarily. An AD tool computes in forward mode
$$\frac{dF}{dx}S = \left(\sum_{i=1}^{n} \frac{\partial F}{\partial x_i} s_{ij}\right)_{j=1,\dots,k} \tag{2.2}$$
which is a n-dimensional vector if F is a scalar-valued function. Choosing S in an appropriate manner determines which derivative object the AD-generated code will compute:

- If $k = 1$ and S is defined as the l-th unit vector, i.e. $S = (0, \ldots, 0, 1, 0, \ldots, 0) \in \mathbb{R}^n$, then (2.2) gives the l-th partial derivative of F:

$$\frac{dF}{dx} S = \frac{\partial F}{\partial x_l}.$$

- If $k = n$ and S is the identity matrix in $\mathbb{R}^{n \times n}$, then (2.2) gives the full gradient, i.e. the vector of all partial derivatives.

- Consequently any number k of weighted linear combinations, i.e. directional derivatives can be computed by choosing the elements of the seed matrix appropriately.

If F is vector- or matrix-valued, relation (2.2) applies for every component of F. The gradient then becomes the Jacobian.

As an important consequence in view of performance it is not necessary to compute e.g. the whole Jacobian if only Jacobian-vector products are needed. This often is the case for example in optimization algorithms.

Basically the same is true for reverse mode calculations, with the only difference that the entries in the seed matrix are transposed since the derivative evaluation is started from the other end.

2.5 AD tools on the market

An overview of some of the available AD software is given in Tables 1 and 2. Both tables reflect our knowledge and assessment, other author may judge differently.

Among the free (for academic applications) tools for Fortran 77 ADIFOR 2 is very robust, requires no code preparation, but is restricted to forward mode. For its successor ADIFOR 3 only one application (performed by the developers) is known to the authors. TAMC which can be accessed remotely has restrictions on code size and is not developed any further. Its successor TAF is a commercial tool. Both have been frequently used in climate modeling since their developers stem from this research area. Odyssee is widely used in France, but the tool is not developed any further.

For C or C++ the choice of tools is very limited, the same is true for MATLAB.

2.6 Choice of tools for the project

Besides the considerations made in the last subsections the experience with the tools listed above and the connections and collaborations of the *Forschungsgruppe Optimierung bei partiellen Differentialgleichungen* at TU Berlin with the tool developers were additional criteria in the choice and strategy during the project:

- Since there is only one AD tool for MATLAB free available, ADMAT was used for the Box Model.

- Following the strategy suggested above, for CLIMBER the forward mode of ADIFOR 2 was used first. Later on TAMC and TAF were tested in forward and reverse mode. Furthermore ADOL-C was applied because it is free and allows higher order derivatives that can be useful in uncertainty computations and optimization. Furthermore we wanted to have a comparison with a tool based on operator overloading.

	Adifor 2	Adifor 3	Odyssee	TAMC	TAF
developer	Rice University Argonne Nat. Lab.		INRIA	R. Giering	FastOpt GbR
Fortran standard	77	77+90	77+90	77+90	77+95
add. features		some MPI			some MPI
technique	ST	ST	ST	ST	ST
modes	fwd	fwd+rev (1 D)	fwd+rev	fwd+rev	fwd+rev
usage	local	local	local	remote	local
availability	free	developer version	free	free	commercial
successful applications	++	?	++	++	+
... in climate models	+	−	+	++	+
future development	−	+	−	−	+

Table 1: AD tools for Fortran. ST: Source transformation, fwd: forward mode, rev: reverse mode

	Adic	Adol-C	ADMAT
developer	Argonne Nat. Lab.	TU Dresden	Cornell Univ.
language	C	C++	MATLAB ≥ 5
technique	ST	OO	OO
modes	fwd	fwd+rev	fwd+rev
usage	local	library	library
availability	free	free	free
successful applications	?	+	+
... in climate models*	?	?	?
future development	+	+	−

Table 2: AD tools for C, C++, and MATLAB. ST: Source transformation, OO: Operator overloading, fwd: forward mode, rev: reverse mode, *: at the beginning of this project

3 Algorithmic Differentiation of Climber 2

The second part of the project was devoted to the application of Algorithmic Differentiation to the more complex climate model Climber in version 2. The task of deriving a tangent linear or adjoint version of this model was much more challenging than for the Box Model since Climber is a coupled model incorporating ocean, ice, vegetation and atmosphere components.

In this section we first give a brief description of the model. Then we describe the state with respect to AD at the beginning of the project and the aims concerning Climber. Afterwards we present the different tools for Algorithmic Differentiation we used for this model. These tools required different levels of changes made to the source code of the model. Moreover during the application of AD to Climber several critical programme sections in the source code were detected. Most of them were changed during the runtime of the project. These changes are described in detail. Other crucial programme sections might be responsible for severe difficulties that were observed in the AD process, specifically in reverse mode. This is pointed out in more details in the applications that are presented next. We give an overview and detailed descriptions of all applications that were studied for Climber. For each we describe the use of the AD tool and necessary code preparations. Then we present the numerical results concerning accuracy and performance.

3.1 Description of the model

The model climber was developed at PIK. Therefore here only a short description is given, for more details we refer the reader to [PGB98]. We want to emphasize the important features of the model concerning automatic differentiation. The first one is the high number of times steps in a typical model run. Since the model takes time steps of one day a number of 360000 steps is necessary for computing 1000 years model time. The second point is the coupled model structure. Climber consists of an atmosphere, an ocean, an ice, a special coupling, a vegetation, and an averaging and output component. These components can be combined in a flexible way using flags that are set in an input file. Atmospheric and coupling component are called every time step (i.e. every day model time). The ocean usually is called every five days. The vegetation component is called at the end of every year.

Climber can be run in two ways:

- Starting from arbitrary initial conditions the model is run into an equilibrium (or stationary) state. This may take several thousand years model time.
- Starting e.g. from a stationary state read from an input file transient runs are performed. These can be of shorter model run time.

Obviously implementation details are important in the AD process as well. Climber in the provided version uses some special compiler-dependent Fortran features. They may lead to difficulties in the AD process, specifically in reverse mode. Among them are jump statements, implicit variable initializations, and more. Furthermore some numerical instabilities were encountered that do not influence the usual model code, but lead to useless results in the derivative computations. These points are studied in more details in Section 3.5.

The overall length of the code we obtained (without comments and empty lines) was 10200 lines Fortran.

3.2 State at the beginning of the project

At the beginning of the project a version of Climber 2 was provided by PIK. There was a collection of test examples for derivative calculations. At PIK the model was successfully differentiated using

ADIFOR with one input and one output variable.

3.3 Aims of this part of the project

The aims of the project concerning CLIMBER can be summarized as follows. The different tasks and aims are listed with respect to difficulty and complexity:

- Test the applicability of AD tools on the model at all. Since the model is far more complex and challenging at first pure sensitivity calculation were in the focus of interest. Optimization runs (as for the Box Model) using the computed derivatives were not our focus. Nevertheless a successful optimization strategy is based on a efficient derivative calculation and should be possible once the latter is done.
- Typical scenarios with distributed input variables and one output variable should be tested.
- Performance issues, concerning both computational time and storage requirements, were crucial points of investigation. This is due to the high number of variables, the complex coupled structure, and the necessity of runs with long model time to compute stationary states.
- User-friendliness or easy usability of the AD tools for the researchers at PIK who are not familiar with this technology was another goal. This point specifically refers to current and future code changes, model extensions, and replacement of model components.
- Higher order derivatives are of interest for the uncertainty group at PIK. Of course they only make sense if satisfying first order derivative code could be generated.

3.4 Used AD tools – overview

Three Algorithmic Differentiation tools were used for CLIMBER: We started with ADIFOR because of its robustness and our experience in using it. As described in the second section of this report ADIFOR is only capable of forward mode. Since in our second application we had a high number of output, but only one input variables, there was the necessity of using a reverse mode tool. Here we tested TAMC and its successor TAF. Furthermore we algorithmically generated a C version of CLIMBER and tested the tool ADOL-C working with Operator Overloading. This was done to test the competitiveness of a tool based on this method, specifically in computing higher derivatives that may be interesting for the uncertainty studies also performed at PIK.

In the following subsections we describe the usage and code preparations that were necessary for these three tools.

3.4.1 ADIFOR

ADIFOR in version 2 requires only small code preparation. The tool takes all Fortran 77 code, only one named do-loop had to be replaced. ADIFOR can be obtained free of charge for academic purposes under [Adifor]. It is installed at PIK.

For the use of ADIFOR one has to perform the following steps:

1. In a *composition file* that must have the suffix `.cmp` simply all source files that are needed to compute the output variable that shall be differentiated from the input variable have to be listed.

```
AD_TOP=climber
AD_PMAX=1
AD_IVARS=cco2
AD_DVARS=tsga
AD_PROG=climber.cmp
```

Table 3: Necessary variables in a `.adf` file .

2. In a second file with the suffix `.adf`, e.g. `climber.adf`, variables and options for ADIFOR have to be set. They define the top-level subroutine, the names of input and output variables, the maximal dimension etc. A simple example is given in Table 3.

   The variables set in the file are:

   - `AD_TOP`: sets the top-level routine
   - `AD_PMAX`: max. number of independent/input variables
   - `AD_IVARS`: name(s) of independent or input variable(s)
   - `AD_DVARS` (same as `AD_OVARS`): name(s) of dependent or output variable(s)
   - `AD_PROG`: file with the list of source code files

   More variables or options may be useful or necessary. Detailed examples are given in the applications.

3. The top-level routine must not be the main programme and the dependent/output variable `tsga` has to be a global variable or a parameter in the top-level subroutine.

4. ADIFOR is then invoked in a shell with

   ```
   Adifor -AD_SCRIPT=climber.adf
   ```

   where `climber.adf` is the file described above in Table 3.

   ADIFOR analyzes the original programme and generates the derivative code. All variables that represent derivatives obtain the prefix `g_`, and for all *active* subroutines/functions lying on the path from input to output variable corresponding new subroutines/functions with the same prefix `g_` are generated.

5. The user than has to write a simple main programme or modify the existing one. In it the seed matrix has to be initialized, compare Section 2.2. The standard variable name for the seed matrix is the name of the input variable with prefix `g_`, in our example in Table 3 thus `g_cco2`. Moreover the `g_` version of the top-level subroutine has to be called. For examples see again the applications below.

ADIFOR offers the opportunity to invoke special routines that report on exceptions due to points of non-differentiability. For details see [AdiMan]. A further advantage of ADIFOR is that points of non-differentiability (e.g. evaluation of the square-root at zero) are detected and the critical values are treated separately to avoid arithmetic overflow.

A disadvantage of ADIFOR is that common blocks are regarded as completely active even if only one variable in them is active. This leads to redundant derivative variables and may result in a very huge executable programme. We had this problem in our second ADIFOR application, see Section 3.9.

3.4.2 Tamc and Taf

Tamc is a source transformation tool developed by Ralf Giering. It is capable of forward and reverse mode, and can be accessed remotely by invoking a small script available at [Tamc]. The tool is not developed any further, its successor Taf [Taf] is now commercially distributed by the company FastOpt. Since Taf and Tamc have the same roots the usage of the tools is similar. Tamc has the bottleneck of the capacity of the server, so sometimes a complex code as Climber leads to an error due to lack of storage on the server. We had the opportunity to use FastOpt's Taf server which has no such problems. A commercial license of course is installed on the customer's computer and will not lead to that kind of problems anyway.

The necessary steps to differentiate a programme are listed below:

1. Tamc and Taf require that code from Fortran include files have to be explicity included. In the package of Tamc, Taf a script is provided. We extended this an finally wrote our own one named `pre_taf` to perform these and some other changes.

2. As for Adifor the top-level subroutine must not be the main programme and the output variable must be global or a parameter.

3. Both tools are steered via options in which input and output variables, top-level subroutine and differentiation mode are specified. A typical call of Taf in a shell has the form

   ```
   taf -toplevel climber -input cco2 -output tsga -forward *.f
   ```

   for forward or

   ```
   taf -toplevel climber -input cco2 -output tsga -reverse *.f
   ```

   for reverse mode. Both tools produce log files (named `tamc_output` and `taf.log`, repscetively) which report on errors or give warnings. Specifically in reverse mode these can be used to detect hot spots in view of performance. In Forward mode the variables are named with the prefix `g_`, in reverse mode with `ad`. The generated derivative source code files get the suffixes `_ftl`, `_ad`, respectively.

4. The last step is the same as for Adifor: The main programme has to be modified to initialize the seed matrix and call the generated new top-level routine.

An advantage of Tamc and taf is that the tools can parse and write Fortran 90 code which may lead to more readable code. Moreover they generate code that is explicitly typed and very nicely written. A further plus concerning storage use is that common blocks are *not* entirely declared as active if one variable is active as it is done by Adifor, see the subsection above.

3.4.3 Odyssee

We only briefly tested the French AD tool Odyssee which allows forward and reverse mode. It is available at [Odyssee] free of charge and can be installed under Sun Solaris and Linux. The Solaris version even has a graphical user interface. Otherwise the tool is used in command line mode. We did not further studied the tool since it has some problems with the Climber code. Moreover the analysis took quite long. Since the tool is not developed any further we did not continue the application of Odyssee on Climber.

3.4.4 Adol-C

This tool is based on operator overloading and is developed and maintained at the Insititut für Wissenschaftliches Rechnen at TU Dresden. It offers forward and reverse mode and derivatives of arbitrary high order. The conceptual difference of an operator overloading tool has several consequences. The first one is that Adol-C is not invoked as the tools described in the last two subsections. Moreover it does not generate new code. Adol-C is a library that can be obtained for free from [Adolc]. This library is linked with the model source code. In the source code some changes have to be made. Basically three steps have to be performed:

1. Declare all variables that lie on the path from input to output as `adouble`, i.e. *active double* instead of C++ `double`.
2. Insert `trace on`, `trace off` statements around the relevant code parts, i.e. those where the output value is computed from the input variable. These start and stop the taping of the relevant operations and variables.
3. Extract the derivative(s) from the tape by calling special routines provided by Adol-C.

3.5 Critical code sections in Climber and resulting changes in the source code

The original idea of AD is to alter as little as possible in the model source code that should be differentiated. In fact it turns out that maybe

- an AD tool requires some code preparation,
- an AD tool is much more strict with the language standard than the compiler itself is,
- numerical instabilities are detected when running the derivative code,
- some source code changes are required to revert the control flow graph for reverse mode,
- some changes turn out to be necessary to increase performance.

The Fortran programming language has some features that are on one hand flexible, but on the other hand error prone and thus not recommended. Moreover most Fortran compilers allow inconsistencies that are detected by AD tools since may disturb code analysis and the derivative code generation. In this subsection we list the problematic code sections detected in Climber and describe how they were replaced. All changes were tested successfully against the results of the original Climber version. If changes in the results occurred they are documented. We give an overview of all problems and changes made in Table 4.

A general change we made for convenience was to put every subroutine/function in a separate file with the name of the routine as filename.

3.5.1 Treatment of include files

Tamc and Taf require explicit incorporation of the source of Fortran include file `*.inc` in the Fortran source files `*.f`. This can be done by a script provided with the tool or with one we have written.This change is not recommended for other tools since it decreases the readability of the code.

see Section	type of problem/change	ADIFOR	ADIFOR performance	reverse mode general	TAMC TAF general
3.5.1	include files	−−	−−	−−	++
3.5.2	splitting of common blocks	+	++	−−	−−
3.5.3	declaration mismatches	++	++	++	++
3.5.4	multiply defined variables	+	+	+	+
3.5.5	reserved word as variable	+	+	+	++
3.5.6	function as parameter	++	++	++	++
3.5.7	IBM intrinsic functions	-	-	-	++
3.5.8	numerical instabilities	++	++	++	++
3.5.9	non-differentiabilities	-	-	-	++
3.5.10	jump statements	-	-	++	-
3.5.11	reorganization of time loop	-	-	+	-

Table 4: Overview of problems and corresponding changes in CLIMBER source code. Changes are ++: required, +: recommended, -: optional, −−: not recommended if not necessary.

3.5.2 Splitting of common blocks

We split up the common blocks such that every variable is in one alone. The names of variable and common block are the same. This was necessary only to obtain performance with ADIFOR, see Section 3.9.2. For other tools it was not necessary, but had no negative effect either. This was also performed by a script.

3.5.3 Parameter list and dimension mismatches

Many Fortran compilers allow mismatches between the numbers of variables in a parameter list in the definition of the subroutine/function and its actual calling statement. AD tools detect these mismatches and report on them as errors. The same is true for mismatches in the dimensions of common block variables.

Both of these inconsistencies were detected in CLIMBER. They could be eliminated easily:

1. The parameter `btime` was eliminated in the calls of the subroutines `sinsol` and `solcon` in subroutine `climber`. The declaration of both subroutine has no parameter list, whereas the call in `climber` passes one parameter. This parameter is redundant since it is already in a common block.

2. The dimension of variable `PCO2M` was adjusted: In subroutines `PCO2` and `DATA_CO2` it was declared with different dimensions (1000 and 1200). The dimension was changed to 1200 in `PCO_2`, too.

3.5.4 Multiply defined variables

Variables were multiply used with different dimensions in different common blocks. This is not generally a problem, but unsafe, and thus was changed:

1. Double appearance of variable `TSUR_i` eliminated:
   Variable occurs twice as three-dimensional array in common block `coup` in include file `coup.inc` and as scalar variable in common block `semic` in file `semi.inc`. First one, i.e the array, was renamed as `tsur_i2`. This refers to the include file `coup.inc` and the subroutines `init_coup, sflux_o, trans_a, restart, rewrite, coupler`.

2. Double appearance of variable `USUR_I` eliminated:
   Variable occurs twice as array and a scalar. Array was renamed as `USUR_I2` in include file `coup.inc` and subroutines `coupler` and `sflux_o`.

3. Double appearance of variable `V2` eliminated:
   Variable occurs twice as array and a scalar. Scalar variable was renamed as `v2u` in include file `bio.inc` and subroutines `initcpar`, `ccparam`.

4. Double appearance of variable `PRCS_I` eliminated:
   Variable occurs twice as array and a scalar. Array variable was renamed as `prcs_i2` in include file `coup.inc` and subroutines `coupler`, `trans_o`.

5. Double appearance of variable `PRC_I` eliminated:
   Variable occurs twice as array and a scalar. Array variable was renamed as `prc_i2` in include file `coup.inc` and subroutines `coupler`, `trans_o`.

6. Double appearance of variable `TAM_I` eliminated:
   Variable occurs twice as array and a scalar. Array variable was renamed as `tam_i2` in include file `coup.inc` and subroutines `coupler` and `sflux_o`.

7. Triple appearance of variable `sst_i` eliminated:
   Variable occurs as scalar, 2-, and 3-dimensional array.

   (a) The 2-dimensional array `sst_i` in subroutines `coupler, sdown, sflux_o, ocn_dat` and include file `coup.inc` was renamed to `sst_i2`.

   (b) Scalar variable `sst_i` in subroutine `inter_sic` renamed as `sst_i3`.

3.5.5 Keyword used as variable name

In subroutine `ocnout` a variable is called `SAVE` which is a reserved Fortran keyword. The name was changed to `SAVE1`. Most compilers do not recognize that as an error, but the AD tools do.

3.5.6 Function passed as parameter

This lead to errors both with Adifor and Tamc/Taf. In subroutine `incche` the subroutine `rtsafe` is called and the external function `hydfunc` is passed to it as parameter `funcd`. Since this is the only occurrence, i.e. `rtsafe` calls always `hydfunc`, the parameter `funcd` was eliminated and `hydfunc` was called directly in `rtsafe`.

3.5.7 Use of Ibm xl Fortran intrinsics

In subroutine `dfluxo` the functions `derf` and `srand` computing random numbers are called. Tamc and Taf had problems with them since they are Ibm xl Fortran intrinsics. They were put in comments during the AD process.

3.5.8 Numerical instabilities

In a derivative additional numerical instabilities may occur. A simple example is the function $f(x) = 1/x$. If the value of x is close to the smallest machine number then the value of the function is a machine number, but the derivative $f'(x) = -1/x^2$ will give Infinity. Problems of this type occurred in Climber.

Operations leading to numerical instabilities and floating point exceptions detected in Climber were:

1. Division by zero occurred in subroutine `dyno` in the lines

   ```
   VCW1=-ZZ(1)/(HOCY(i,n)-ZZ(1))*VCW(i,1,n)
   ```

   for some indices. In the original the resulting value was not used. The code section was replaced by an `if-else-endif` statement.

2. Division by zero occurred in in subroutine `dyno` in the lines

   ```
   vsum=vsum/HOCY(i,n)
   ```

   for some indices. In the original the resulting value was not used. The line was moved to the following loop which has an entry condition avoiding the problematic index values.

3. The code lines in subroutine `ccdyn`

   ```
   dst=forshare_st-fd*exp(-1./t2t)-st(lat,lon)
   st(lat,lon)=st(lat,lon)+dst
   ```

   which led to cancellation for some values of `lat`, `lon` were replaced by

   ```
   dst=forshare_st-fd*exp(-1./t2t)-st(lat,lon)
   st(lat,lon)=forshare_st-fd*exp(-1./t2t)
   ```

   This caused a slight change of results in Climber. Nevertheless the new results should be more accurate.

3.5.9 Points of non-differentiability

A typical point of non-differentiability is the evaluation of the square-root, e.g. for calculation of norms, at the point $x = 0$ where the function is not differentiable, its derivative would give Infinity. The same is true for the functions $f(x) = x^\alpha$ with $\alpha < 1$. Adifor automatically treats this problem by an `if`-statement, for the use of other tools we changed the corresponding code sections themselves.

This refers to the subroutines `berger`, `bergor`, `calc_k1`, `calc_k2`, `calc_kb`, `calc_kw`, `cdrg`, `crisa`, `geo_ocn`, `geo_upd`, `mbiodyn`, `orbit`, `secm`, `trns`, `u2d`, `u3d`, and `zslp`.

3.5.10 Jump statements

We already mentioned in Section 2.2 that jump statements in the code flow lead to problems when using reverse mode. In Fortran the following statements are critical:

- `goto` statements
- `return` in functions used before the end of the function in an `if` block
- `break` statement in an `if` block.

Tamc and Taf in reverse mode can treat most of these jumps. If this is not possible the error `irreducible control graph` appears in the output file.

Nevertheless for our reverse mode computations with Tamc and Taf we replaced all jumps by equivalent features, making use of the Fortran 90 `do while` statement. Also arithmetic if statements which occurred in one Climber subroutine `berger` were replaced.

Comment lines including the original code where inserted. This refers to subroutines `berger`, `cadj`, `dfluxo`, `geo_ocn`, `lwr_s`, `orbit`, `root`, `sdown`, `u3d`, `zlsp`.

3.5.11 Reorganization of the main time loop

In Climber some components, e.g. the ocean model, are called only every five days (i.e time-steps). This is realized in the programme by an integer flag that is set to one every step can be divided by five without remainder. This realization makes it very difficult to analyze the code's flow graph backwards as it is necessary for reverse mode AD. Thus we changed the realization of the time loop subroutine `time_loop` itself and the subroutine `climber` that realizes one year model time. They now consist of three nested loops, namely

- an innermost performing five days model time, i.e. five time steps and then calling the ocean.
- a middle loop which performs 72 calls of the innermost and thus represents one year model time, calling at the end vegetation, averaging and output routines. Both are realized in a subroutine `climber_mod` replacing the original `climber`.
- Finally the outer loop counting the years model time in `time_loop_mod` which replaces the original `time_loop`.

Moreover the averaging routine `gaver` was moved from the output in subroutine `out` to `climber_mod` to separate averaging (where output values to be differentiated are computed) and the pure writing of output.

The detailed list of changes is:

1. Subroutine `time_loop` was replaced by `time_loop_mod`, see Table 5:
   (a) The main counter in the routine was changed from days to years (now `inyr`).
   (b) `time_loop_mod` calls a new subroutine `climber_mod` (see below) and passes the current value of the year `inyr` as parameter to it.
2. The subroutine `climber` was replaced by `climber_mod`, see Table 6:
   (a) `climber_mod` now computes one year model time (instead of one day in `climber`)
   (b) There are two main loops in `climber_mod`: An outer loop computing one year in 72 steps of each 5 days and an inner loop for 5 days is realized as a call to the new subroutine `day_1` (see below). At the end of the outer loop routines are called that are needed only once a year (e.g. vegetation).
   (c) A new subroutine `day_1` performs one day time-step, see Table 7.
3. The separation of output and calculation was achieved in the following way: The averaging routine `gaver` was moved to the main loop in `climber_mod`. The same was done for subroutine `slr` which now is also called in `climber_mod`. Subroutine `out` now has only real output statements and becomes passive for AD.

3.5.12 Implicit typing and initialization

Fortran allows implicit typing, i.e. variables do not have to be declared explicitly. AD tools do not need explicit typing, but it is highly recommended not to use this opportunity because of programme errors that are often introduced just by simple typing errors in the programme source code.

A more dangerous feature that for example the IBM Fortran compiler (`xlf, f77`) in its standard configuration uses is the automatic initialization with zeros of all variables in the beginning when the programme is started. This feature leads to non-portability to compilers that do not initialize automatically (e.g. the `g77`). But this feature becomes even more dangerous in combination with the effect described in the next subsection.

```
      SUBROUTINE TIME_LOOP_mod
...
      integer inyr
*********************************************************************
*  NYRMX  - number of years
*  inyr   - count unit of years
**********************************************************************
      do inyr=0,NYRMX-1
         call climber_mod(inyr)
      enddo
      return
      end
```

Table 5: Additional parameter in subroutine `time_loop_mod`.

```
      SUBROUTINE CLIMBER_mod(inyr)
...
      integer inyr,day1,day5
...
      if (inyr.eq.0) nts=1
      do day5=1,72
         do day1=1,5
           call day_1(day5,day1)
         enddo
         if (KOCN.ne.0) then
            ...         ! unchanged
         endif
      enddo
      if (KTVM.ne.0) call CLIM_TVM
      if (KBIO.eq.1) call TVM
      if (KTIME.eq.1.or.KSOLC.eq.1) call SINSOL
      call SLR
      call gaver(kendy) ! call was previously in OUT
      call slr          ! call was previously in OUT
      call out          ! output separrated from averaging
      nts=nts+1
      return
      end
```

Table 6: New structure of main time loop in subroutine `climber_mod`.

3.5.13 Static use of automatic variables

The Fortran 77 standard defines that local variables, i.e. variables used in a subroutine/function that are not in a common block and not passed as parameters, have the status of so-called *automatic* variables. That means they are created every time the subroutine/function is called and (and least should be) destroyed after it is finished. This implies that the values of these variables are lost after the finishing of the subroutine. If one wants such a variable to retain its value from one call of a subroutine/function to the next (e.g. in a time loop as in CLIMBER), they have to be declared as *static* variables. This in Fortran is done by giving the variable a `save` attribute.

```
        Subroutine day_1(day5,day1)
...
        integer inyr
        integer day5,day1

        if (.not.((day5.eq.1).and.(day1.eq.1))) then
            if (KTVM.ne.0) call CLIM_TVM
            call gaver(kendy) ! call previosly in OUT
c            call OUT         ! only writing
            nts=nts+1         ! day counting
        endif
...
c=======================================================
c Modules
            call PCO2
            call ATM
            call COUPLER
       return
       end
```

Table 7: Inner most loop computing 5 days model time in new subroutine `day_1`.

Compilers allow to implicitly define all local variables as static or automatic. The IBM Fortran compiler has the `-qautosave`, `-qnoautosave` options.

It turned out that both the `xlf` and the `g77` compilers save local variables defined as automatic, i.e. without explicit `save` statement nor `-qautosave` compiler option, from one subroutine call to the next one. We believe that this is due to the fact that once the programme is loaded, the physical space the variable occupies remains the same during the whole programme lifetime, and that the variable in fact is never destroyed explicitly.

This e.g. is true for the annual global mean temperature `tsga` and is used in CLIMBER to compute the sum of temperatures over the daily time steps in one year. This variable is no global variable in a common block nor passed as parameter. Nevertheless in every time step and every call of the corresponding subroutine `gaver` the value of `tsga` is preserved. Obviously this is important for the correctness of the intended value of the variable. But it is obviously as well that making use of this feature is rather dangerous, for example if the subroutine is used in a programme with a more dynamic structure.

An AD tool that has to detect dependencies between variables, and furthermore to conform with the language standard, may generate code that does not compute the correct derivative. A possible step for the future thus seems to use totally explicit typing and initialization of all variables, a strategy that is highly recommended in programming anyway.

3.6 A guide for programming in view of Algorithmic Differentiation

In this section we want to summarize the most important guidelines for model developers and programmers in view of the use of a possible algorithmic differentiation of a model:

- Use explicit typing.
- Use explicit variable initialization.

- If variables should be used as static, i.e. they should keep their value from one call of a subroutine to the next, make them global or declare them as `save`. This refers to Fortran programmes.
- For reverse mode avoid all jump statements, i.e. `goto`, `break`, `exit` statements. Avoid `return` statements in an `if` block in subroutines/functions. Use `do while` loops instead.
- Avoid old Fortran features as arithmetic `if`'s.
- If periodically called subroutines/functions arise in loops, do not realize this with flags that are set to 1 or 0 depending on the index of the loop. Use nested loops instead.
- For reverse mode keep in mind whether your programme's control flow graph can be reversed.
- Check your model code for numeric instabilities before you differentiate it. You may use the options (on IBM's `xlf`):

  ```
  -g -qflttrap="ov:zero:inv:en" -qsigtrap=xl__ieee
  ```

  which displays the code lines where overflow (`ov`), division by zero (`zero`) and invalid operations, i.e. operations on infinite values (`inv`) are performed. For more details see the `xlf` manual or the info data base.

Summarizing we can say that the more structured and safely written a programme is, the easier is it to pass through an AD tool successfully.

3.7 Applications – Overview

The first and simplest application that was tested with CLIMBER was to compute the derivative of the global mean temperature (`tsga` in the model code) with respect to changes in CO_2 emissions (`cco2` in the model). Since the value of the CO_2 is fixed once at the beginning of the model run this application has one input and one output variables. Therefore it is the easiest case for AD.

The second application was to compute the derivative of the global mean temperature (again `tsga`) with respect to the zonal area of forest (`stte` in the model code). Since the zonal area of forest is a weighted sum of the variable `st` in fact the derivative of the global mean temperature `tsga` with respect to the latter has to be computed. The variable `st` is a two-dimensional array of size 126. Therefore this is a classical application for the reverse mode. Nevertheless we started with ADIFOR and forward mode because of the robustness of the tool, and to test its limits with respect to performance and storage requirements. Moreover this gives us the opportunity to obtain results for the derivatives that can be used for comparison.

CLIMBER contains several more variables which are similarly computed as the global mean temperature `tsga`. Thus it should be easily possible to obtain similar results in cases where the global mean temperature `tsga` is replaced by those.

3.8 Application 1: Single input – single output variable (forward mode ADIFOR)

The aim of this application was to compute the sensitivity of the global mean temperature (`tsga` in the model code) with respect to CO_2 (`cco2`). This example is the simplest case for an AD tool since it has only one input and one output variable. Thus here the forward mode is preferable.

```
AD_TOP=climber_mod
AD_PMAX=1
AD_IVARS=cco2
AD_DVARS=tsga
AD_PROG=climber.cmp
AD_SCALAR_GRADIENTS=TRUE
AD_EXCEPTION_FLAVOR=reportonce
```

Table 8: File `climber.adf` for application 1.

3.8.1 Use of Adifor

In this case the file `climber.adf` looks as in Table 8.

Here we used two options additional to those already described in Section 3.4.1, namely:

- `AD_SCALAR_GRADIENTS`: set to `TRUE` if the derivative/gradient is a scalar (only possible for `AD_PMAX` = 1, but in this case recommended for performance reasons).
- `AD_EXCEPTION_FLAVOR`: set to `REPORTONCE` this option enables the user to obtain non-differentiability points in derivative code (if a special subroutine from the Adifor library is called)

To ensure that the dependent/output variable `tsga` is global an additional common block named `tsga` was introduced in `climber_mod`.

The seed matrix S = `g_cco2` in this case is just a scalar and was set to 1. The routine were the derivative code is invoked is sketched in Table 9.

```
      common /g_cco2/g_cco2

c     seed "matrix":
      g_cco2=1.

      do inyr=0,NYRMX-1
         call g_climber_mod(inyr)
      enddo
```

Table 9: Relevant lines of the subroutine `time_loop_mod` calling the AD-generated subroutine `g_climber_mod`.

3.8.2 Comparison with finite difference derivatives

We compared the results obtained with the AD-generated derivative with finite difference computations. The results are depicted in Figure 1 for 10 years model time with and without restart from the equilibrium state. These two figures show the dependency of the finite difference (FD) derivatives with respect to the used step size. As already pointed out before the choice of the step size is not easy. Here the value $h = 10$ showed the less oscillating behavior. The result of the AD derivative fits closely to the FD derivative with this step-size. Figure 2 shows the FD derivative with this (in this case somehow appropriate) step size for a 1000 year run with restart.

Summarizing the result obtained with AD are quite reasonable and show their superiority compared to finite diffrence approximations were the appropritae step-size is always difficult to find. In that sense also in performance their is a gain using AD here since it avoids tests with different FD step-sizes.

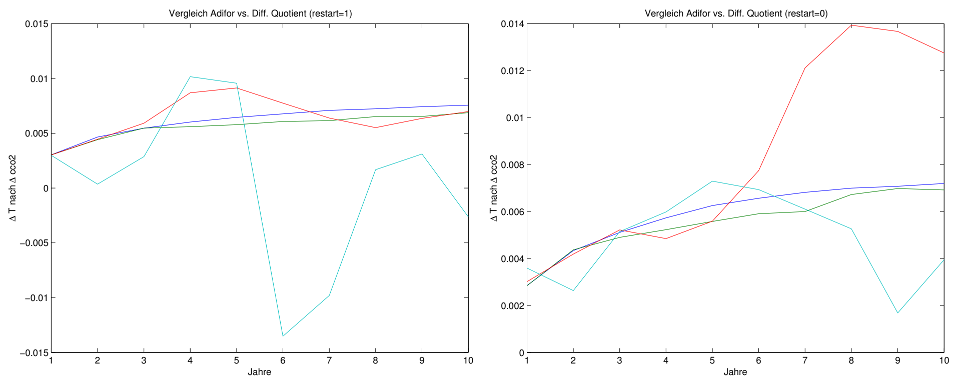

Figure 1: Derivative of global mean temperature with respect to CO_2 at the value cco2=280 with (left) and without (right) restart from an equilibrium state. Derivative obtained by AD (in dark blue) and central finite differences with step-size $h = 0.1$ (bright blue), $h = 1$ (red), $h = 10$ (green).

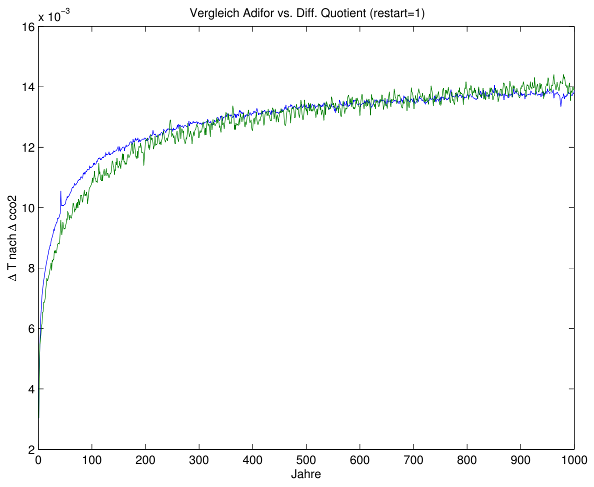

Figure 2: AD- (in dark blue) and FD-generated derivatives (in green) with step-size $h = 10$ of global mean temperature with respect to CO_2 with restart.

3.9 Application 2: Multiple input – single output variable (forward mode Adifor)

This second application is much more challenging than the one described above. Aim was to compute the sensitivity of the annual global mean temperature `tsga` with respect to the zonal area of forest (variable `stte` in the model code). The latter is computed in subroutine `gaver` as a linear combination of variable `st`:

$$\mathtt{stte}_i = \sum_{n=1}^{\mathtt{ns}} \mathtt{st}_{in}\mathtt{carea}_{in}(1 - \mathtt{frglc}_{in}), \tag{3.3}$$

see the code fragment in Figure 10.

```
      do i=1,IT
         do n=1,NS
            STTE(i)=STTE(i)+ST(i,n)*carea(i,n)*(1.-FRGLC(i,n))
         enddo
      enddo
```

Table 10: Calculation of `stte` as linear combination of `st`. The variables `carea` and `frglc` are constants.

Thus we had do define variable `st` as input variable. Because of the dimensions of input and output variable and the facts we pointed out in Section 2.2, this is a classical application for reverse mode. Nevertheless we started with ADIFOR; reverse mode was tested later on.

3.9.1 Computation of the full gradient

At first we computed the full gradient of `tsga` with respect to all components of `st`. Since `st` is a matrix itself the resulting object

$$\mathtt{g\_tsga} = \frac{d\mathtt{tsga}}{d\mathtt{st}} = \left(\frac{\partial \mathtt{tsga}}{\partial \mathtt{st}_{ij}}\right)_{ij} \in \mathbb{R}^{\mathtt{it}\times\mathtt{ns}}$$

is a matrix as well. Nevertheless we refer to it as the gradient since the mapping to be differentiated in this case is

$$\begin{aligned} F \ : \ \mathbb{R}^{\mathtt{it}\times\mathtt{ns}} &\to \mathbb{R} \\ \mathtt{st} &\mapsto \mathtt{tsga}. \end{aligned}$$

In this case the file `climber.adf` looks as in Table 11. The variable `st` has the dimension $\mathtt{it}\cdot\mathtt{ns} = 126$.

```
AD_TOP=climber_mod
AD_PMAX=126
AD_IVARS=st
AD_DVARS=tsga
AD_PROG=climber.cmp
AD_SCALAR_GRADIENTS=FALSE
AD_EXCEPTION_FLAVOR=performance
```

Table 11: File `climber.adf` for computation of the full gradient of `tsga` with respect to `st`.

Computation of the full gradient implies that the seed “matrix” S is in fact not a matrix anymore but formally a tensor of fourth order, i.e. $S \in \mathbb{R}^{\mathtt{it}\times\mathtt{ns}\times\mathtt{it}\times\mathtt{ns}}$. Following the definition in (2.2) we have

$$\frac{dF}{dx} S = \frac{d\mathtt{tsga}}{d\mathtt{st}} S = \left(\sum_{i=1}^{\mathtt{it}} \sum_{j=1}^{\mathtt{ns}} \frac{\partial F}{\partial \mathtt{st}_{ij}} s_{ijkl}\right)_{k=1,\ldots,\mathtt{it},l=1,\ldots,\mathtt{ns}} . \tag{3.4}$$

To compute the full gradient we have to set S to the identity tensor, i.e.

$$s_{ijkl} = \begin{cases} 1, & (i,j) = (k,l) \\ 0, & \text{elsewhere} \end{cases}$$

In reality the storage of the seed matrix is done in a slightly different way: ADIFOR requires that the seed matrix is provided as an array that has one index more than the input variable itself. The additional index (let us call it r) in fact represents the first two indices i, j in (3.4) using the formula

$$(i,j) \mapsto i + (j-1) \cdot \texttt{it} \;=:\; r.$$

This is the way Fortran stores a two-dimensional array of size `it` $\times$ `ns` (rows $\times$ columns), namely column-wise.

This implies that the initialization of the seed matrix in our case had to be done as

$$s_{rkl} \;=\; \texttt{g\_st}_{rkl} = \begin{cases} 1, & r = k + (l-1) \cdot \texttt{it} \\ 0, & \text{elsewhere} \end{cases}$$

The realization in code in this case can be found in Table 12.

```
      parameter (g_pmax_=126)
      real g_st(g_pmax_, it, ns)
      common /g_st/ g_st
      g_st=0.0
      do k=1,it
         do l=1,ns
            g_st((l-1)*it+k,k,l)=1.0
         enddo
      enddo
```

Table 12: Seed matrix used by ADIFOR for the full gradient. The fourth line is a Fortran 90 array assignment.

Note that the total number of elements in the seed matrix is $\texttt{it}^2\texttt{ns}^2 = 15876$. Moreover several objects of this or similar size are generated in the AD process. It is obvious that this may lead to storage problems. A disadvantage of ADIFOR in this case is (as already mentioned) that the tool makes an entire common block active, even if only some variables in it really lie on the computational graph from input to output variable. As a result the executable compiled from the derivative code generated by ADIFOR could not be loaded into memory.

3.9.2 Code preparations to save storage in the derivative code

To overcome the above mentioned high requirements of at least partially redundant storage we separated all common blocks. This avoids the generation of unnecessary active variables. All common block variables were put in a separate common block named as the variable itself. Moreover we compiled and run the derivative code on a 64-bit machine (using the IBM compiler's `-q64` option) and set

```
limit memoryuse unlimited
limit datasize unlimited
```

to provide sufficient memory. Note that the libraries containing the ADIFOR exception handler routines are not available in 64 bit version. Thus the `AD_EXCEPTION_FLAVOR` has to be set to `performance` in this case. It was then possible to compute the full gradient.

Concerning performance the relation between the time used for a function evaluation and a gradient computation was 1 : 22, where both values were obtained with `-O3` optimization flag. This already much better than the theoretical value of 1 : 126 for forward mode or finite difference gradient computation.

3.9.3 Comparison with finite difference derivatives

We tested the accuracy of the partial derivatives of `tsga` with respect to `st`. The results for one component of the gradient can be seen in Figures 3 and 4. Both pictures indicate convergence of the finite difference derivatives to the AD derivative. Moreover they give an impression of the significant oscillations and dependency on the step-size of the FD derivative, whereas the AD derivatives are rather smooth.

3.9.4 Computation of the weighted gradient/directional derivative

The original aim of this application was not to compute the full gradient of the annual global mean temperature `tsga` with respect to `st`, but with respect to the *zonal* area of forest `stte` defined as the linear combination (3.3). Thus we are interested in

$$\texttt{g\_tsga} = \frac{d\texttt{tsga}}{d\texttt{stte}} = \left(\frac{\partial\texttt{tsga}}{\partial\texttt{stte}_i}\right)_i \in \mathbb{R}^{\texttt{it}}.$$

From (3.3) we deduce that `tsga` does not directly depend on `stte`, but both `tsga` and `stte` depend on `st`. The influence of a change in `stte` on the change in `tsga` thus depends on the sum of the changes induced by variation in the corresponding components of `st`. We therefore have

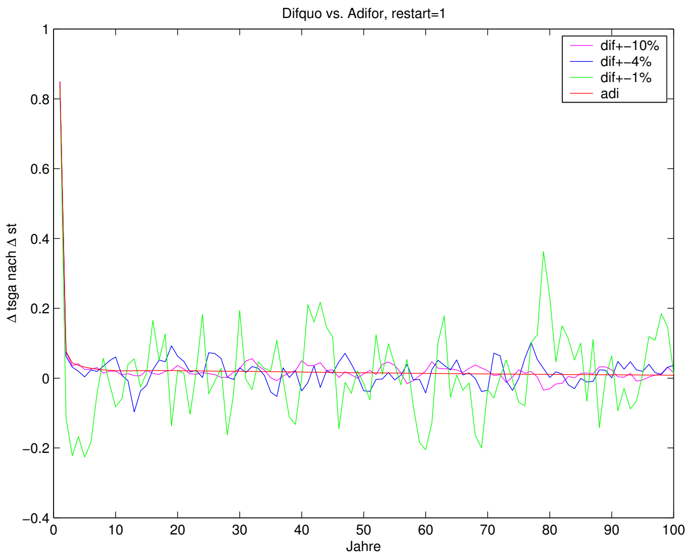

Figure 3: Comparison of AD and FD derivative of `tsga` with respect to $\texttt{st}_{21}$ with restart from an equilibrium state in a run over 100 years model time. Red: ADIFOR, magenta: central FD with step-size $0.1 \cdot \texttt{st}_{21}$ blue: $0.04 \cdot \texttt{st}_{21}$, green: $0.01 \cdot \texttt{st}_{21}$.

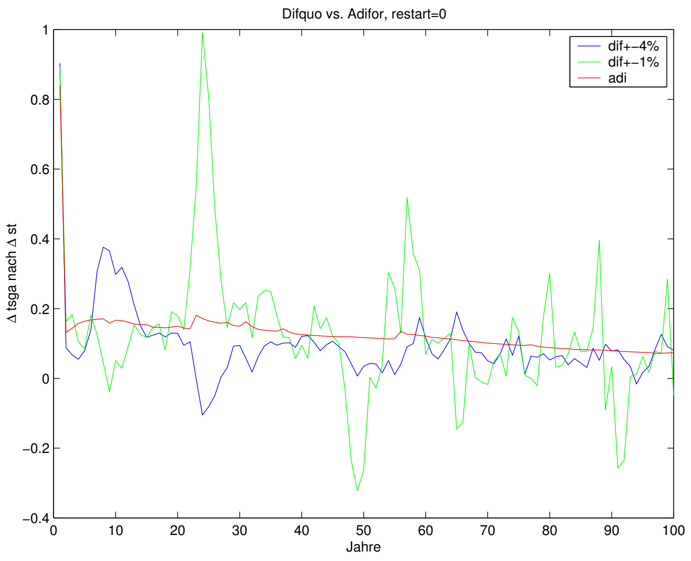

Figure 4: Same as in Figure 3, but without restart.

to differentiate `tsga` with respect to `st`, but we do not need the complete gradient of dimension $\mathtt{it} \cdot \mathtt{ns} = 126$, but only the $\mathtt{it} = 18$ weighted combinations have to be computed.

The file `climber.adf` now looks as in Table 13. The only difference to the one in Table 11 is the lower dimension of the gradient objects defined in `AD_PMAX`.

```
AD_TOP=climber_mod
AD_PMAX=18
AD_IVARS=st
AD_DVARS=tsga
AD_PROG=climber.cmp
AD_SCALAR_GRADIENTS=FALSE
AD_EXCEPTION_FLAVOR=performance
```

Table 13: File `climber.adf` for computation of the weighted gradient.

Now we use the seed matrix

$$S \;=\; \mathtt{g\_st} = (\mathtt{g\_st}_{ikl})_{ikl} \in \mathbb{R}^{\mathtt{it}\times\mathtt{it}\times\mathtt{ns}}$$

and initialize it as follows:

$$s_{ikl} \;:=\; \mathtt{g\_st}_{ikl} := \left\{ \begin{array}{cc} \mathtt{carea}_{kl}(1-\mathtt{frglc}_{kl}), & i=k \\ 0, & i \neq k. \end{array} \right\} \quad \begin{array}{rcl} i,k &=& 1,\ldots,\mathtt{it}, \\ l &=& 1,\ldots,\mathtt{ns}. \end{array}$$

The realization in the code is shown in Table 14.

This initialization saves a factor $\mathtt{ns} = 7$ storage and of course also some computational effort compared to the computation of the full gradient. The relation between the time used for a function evaluation and a computation of this weighted gradient (or directional derivative) is now about $1:7$, where again both values are obtained with `-O3` optimization flag.

```
      parameter (g_pmax_= 18)
      real g_st(g_pmax_, it, ns)
      common /g_st/ g_st
      g_st=0.0
      do i=1,it
         do k=1,ns
            g_st(i,i,k)=carea(i,k)*(1.-FRGLC(i,k))
         enddo
      enddo
```

Table 14: Seed matrix used by ADIFOR for the weighted gradient

3.9.5 Numerical results of sensitivity calculations

Here we show the result of the sensitivity calculations with respect to the whole two-dimensional array `st` in Figure 5 and with respect to the weighted array `stte` in Figure 6.

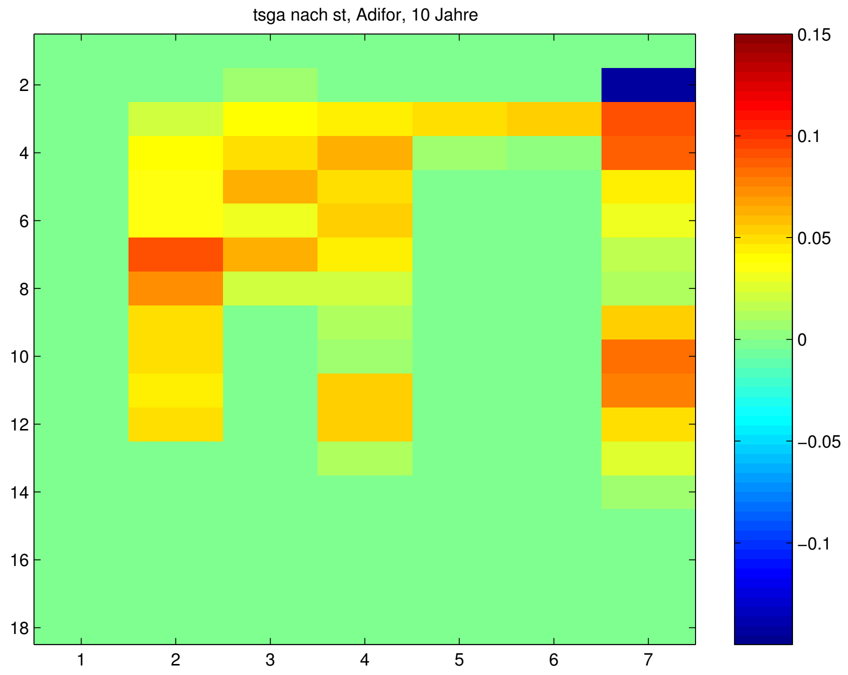

Figure 5: AD derivative of `tsga` with respect to `st` (with restart) after 10 years model time.

3.10 Results with TAMC and TAF

Let us briefly recall the considerations already made in Section 2.2: The application we considered in the last section, namely to compute the sensitivity of the annual global mean temperature which is a scalar variable (`tsga`) with respect to the variable `st` which is an array with 126 elements, is a classical case for the reverse mode of AD.

It is nevertheless recommended to analyze a code with forward mode first, even if one plans to use reverse mode for the above reasons.

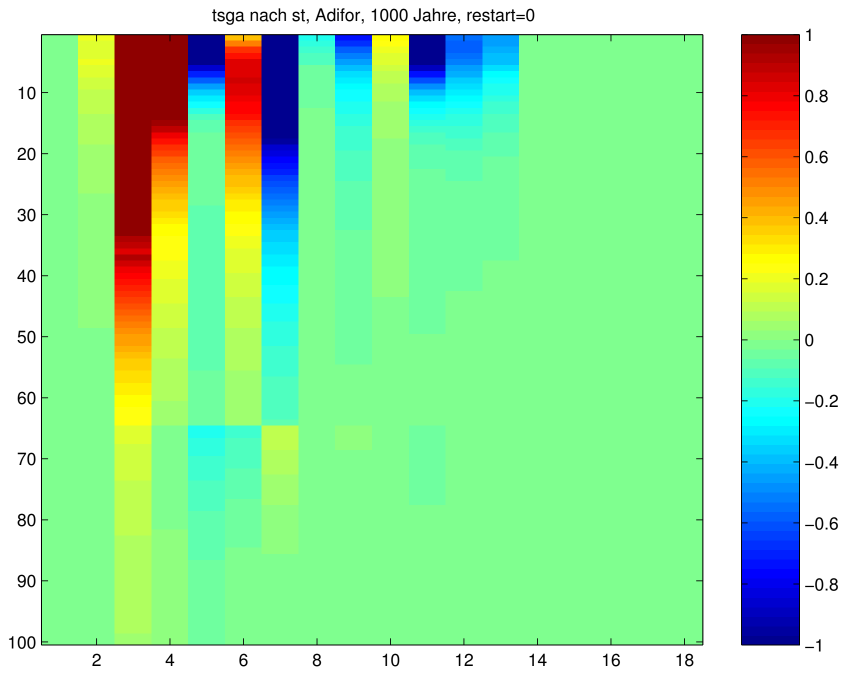

Figure 6: AD derivative of `tsga` with respect to `stte(1:18)` without restart for 1000 years in steps (on the vertical axis) of 10 years model time.

3.10.1 Forward mode

Using a script all necessary changes for TAMC and TAF (we from now on mention only the latter) were made, e.g. the incorporation of the Fortran include files in the `.f` files themselves. The script moreover puts all needed source file in one named `taf.f`.

Then we applied TAF in forward mode by calling first

```
taf -toplevel climber -input cco2 -output tsga -forward taf.f
```

which was our first test for ADIFOR, too (compare Section 3.8).

Initialization of the seed matrix and building the new main programme for forward mode is done in the same way as for ADIFOR in this case with one input and one output variable. Even the name of the subroutines (with prefix `g_`) and active variables are the same, the file names have the suffix `_ftl` for *forward tangent linear*.

Unfortunately the results were not correct compared to the ones obtained by ADIFOR and finite difference computations in Section 3.8.2.

3.10.2 Reverse mode

Even though the results with the forward mode were not promising we tested the reverse mode (our original goal) by invoking

```
taf -toplevel climber -input st -output tsga -reverse taf.f
```

after we made the code preparations that are necessary. They were already mentioned in Section 3.5. Most important among them are

- the elimination of jump statements

- and the reorganization of the main time loop.

The files TAF generates in reverse mode have the suffix `_ad` and contain subroutines with prefix `ad` (referring to *adjoint* code or model). The seed matrix initialization in this case is simple: In reverse mode it has the dimension $m \times m$ where m is again the number of output variables. Its variable name in TAF is the name of the output variable with the prefix `ad`. In our case the seed matrix thus is just a scalar and was initialized as

$$S = \texttt{adtsga} \ = \ 1.0$$

The result of the derivative of `tsga` with respect to `st` is to be found in the two-dimensional array `adst` generated by TAF.

Unfortunately TAF had problems here, too: The generated code did not run without errors. We were in contact with the developers from FastOpt Hamburg. Some problems could be solved, but at the end of this project no successful run could be finished.

3.10.3 Perspectives using TAF

The results presented above at first sound very disappointing. Nevertheless it has to be taken into account that TAF is successfully working for several other climate models, compare the TAMC and TAF homepages [Tamc],[Taf]. It was also successfully applied to and used for optimization in a fluid mechanics code at TU Berlin. The generated code was very efficient and fast. Moreover the tool is one of the rare tools that are under steady development. Summarizing it will take some more time , effort, and collaboration with the developers themselves to obtain a running AD version of CLIMBER differentiated with TAF. We will come back to this point in the summary of this report, see Section 4.

3.11 Results with ADOL-C

The operator overloading tool ADOL-C is one of the most modern AD software. It is under steady further development. Moreover it is free and provides the opportunity to compute higher order derivatives with not much additional effort. This was the reason why we included it in our investigations.

Since it is a tool working for the C/C++ language we needed a C version of CLIMBER. Its derivation is described in the next section. Afterwards we describe how the tool is used which – since it is based on operator overloading – is more similar to ADMAT than to ADIFOR or TAF. We close this section with the remaining problems and perspectives of the use of ADOL-C.

3.11.1 Generating the C version of CLIMBER

We used the tool `f2c` (see [f2c]) to generate a first basic C version of CLIMBER. The code this tool produces is not very readable. Moreover it did not run at first and when it did (after some changes in write and read statements) it produced wrong results. Therefore a debugging process was necessary which was successful in the end. Moreover we made additional changes to delete the use of the `f2c` libraries from the code. We do not want to mention the changes we made in detail, since they are not that important with respect to AD.

At the end we obtained a C version of CLIMBER that produces the same results as the original Fortran version and is independent from any (machine-dependent) `f2c` libraries.

3.11.2 Use of Adol-C

Since Adol-C only uses double variables a first additional change we had to made was to change the C version of Climber to double precision variables. This version was tested against the old one. Some minor changes in the results occur, but they were to be expected due to the bigger accuracy and different rounding. The main task when preparing code for Adol-C is that all active variables, i.e. those who lie on the path from input to output variable, have to be declared as such. They should be of type `adouble`. Since in such a complex model as Climber it is not easy to distinguish active from passive variables we in a first approach simply set all variables to be active.

3.11.3 Remaining problems and perspectives using Adol-C

The problems occurred when using Adol-C are that the tape generation took too much storage. This clearly is due to the fact that all variables have been made active (`adouble`) in the first step. Thus a tape necessary for an evaluation of the AD derivative in forward or reverse mode could not successfully generated. The available storage was exceeded already after some days model time. The minimal time would be one year to record all model components at least once. The logical consequence is to write a tool or script that detects active variables and only declares them as `adouble`. This should be possible to write, even more since some scripts analyzing Fortran code are available at PIK, written by Cezar Ionescu for different purposes. We already made use of them for preparing code for Adifor.

4 Summary and Perspectives

The results obtained for Climber can be separated in two parts:

- The forward mode computations with Adifor were successful. The tool is free, robust and not very difficult to use. After some minor changes in the code that improve efficiency it was possible to generate derivative code with respect to up to 126 independent variables. The results are reliable and show their superiority compared to finite difference derivatives. Nevertheless forward mode comes to its limitations when the number of independent variables becomes too high.
- For reverse mode applications at first all code changes that are necessary prior to applying a tool at all were completed.

  Concerning the tested tools the results are the following:

  - Among the reverse mode tools Taf by now did not work in a satisfying manner. The generated code did not run successfully, even in forward mode there are errors in the results. Further investigations will be necessary if the tool should be used. On the other hand there are results for other models differentiated by Taf which are very satisfying, specifically concerning performance issues, compare e.g. [HS02].

    The problems that Taf have might be induced by some Fortran features Climber uses. Here we refer to the points described in Sections 3.5.12 and 3.5.13. A recommended first step would be a full explicit typing and initializing throughout Climber.
  - Adol-C at first needs a detection of the active variables to become feasible. The question how effective it is is difficult to answer in the current state of the investigation. The advantage of Adol-C of course is its free availability.

A general point when thinking of reverse mode is the incorporation of so-called *checkpointing schemes* to exploit computer power and storage in an optimal way in long time model runs, even more if parallel machines are available. This topic is under current study in the AD and optimization/control community. The group developing ADOL-C, for example, is working in this area, see [GW00].

References

[Adifor] ADIFOR homepage, Argonne National Laboratory, Argonne IL, USA, [http://www-unix.mcs.anl.gov/autodiff/ADIFOR]

[AdiMan] ADIFOR User's Guide, Argonne Nat. Lab., IL, USA: [http://www-unix.mcs.anl.gov/autodiff/ADIFOR/AdiforDocs.html].

[Admat] Homepage of Arun Verma/ADMAT, Cornell Univ., Ithaca, NY, USA. : [http://www.tc.cornell.edu/ averma/AD]

[Adolc] ADOL-C homepage, Institut für Wissenschaftliches Rechnen, Technische Universität Dresden: [http://www.math.tu-dresden.de/wir/project/adolc/index.html]

[f2c] f2c available from netlib: [http://elib.zib.de/netlib/f2c/index.html]

[GK98] R. Giering, and T. Kaminski (1998): Recipes for Adjoint Code Construction. *ACM Transactions on Math. Software*, **24**(4), 437–474.

[Gri00] A. Griewank (2000): *Evaluating Derivatives - Principles of Algorithmic Differentiation*, SIAM Frontiers in Appl. Math., Philadelphia, USA.

[GW00] A. Griewank, A. Walther (200): Revolve: An Implementation of checkpointing for the reverse or adjoint mode of differentiation. *ACM Transactions on Mathematical Software* **26**(1), pp. 19 - 45.

[HS02] M. Hinze, T. Slawig (2002): Adjoint gradients compared to gradients from algorithmic differentiation in instantaneous control of the Navier-Stokes equations, *TU Berlin, Institut für Mathematik, Technical report* 735-2002.

[Odyssee] Odyssee homepage. INRIA France: [http://www-sop.inria.fr/safir/SAM/Odyssee]

[Opt96] Optimization Toolbox For Use with MATLAB User's Guide Version 1, The Mathworks Inc., Natick, MA, USA 1996.

[Opt00] Optimization Toolbox For Use with MATLAB User's Guide Version 2, The Mathworks Inc., Natick, MA, USA 2000.

[PGB98] V. Petoukhov, A. Ganopolski, V. Brovkin, M. Claussen, A. Eliseev, C. Kubatzki, S. Rahmstorf: CLIMBER-2: A climate system model of intermediate complexity. Part I: Model description and performance for present climate, *PIK Report No. 35.*

[Rah96] S. Rahmstorf (1996): On the freshwater forcing and the transport of the Atlantic thermohaline circulation, *Climate Dynamics* **12**: 799-811.

[Tamc] TAMC homepage: [http://www.autodiff.com/tamc]

[Taf] TAF homepage, FastOpt GbR Hamburg, Germany: [http://www.fastopt.de]

[Ver] A. Verma: ADMAT: Automatic Differentiation in MATLAB using object oriented methods, [http://www.tc.cornell.edu/ averma/AD/admatoo.ps]