

# 程序代写代做 CS编程辅导



ADOL-C:<sup>1</sup>

The Automatic Differentiation  
Library for C/C++ Programs Written in C/C++

Version 1.10-trunk, January 2019

Andrea Walther<sup>2</sup> and Andreas Griewank<sup>3</sup>

## WeChat: cstutorcs

Abstract

The C++ package ADOL-C described here facilitates the evaluation of first and higher derivatives of vector functions that are defined by computer programs written in C or C++. The resulting derivative evaluation routines may be called from C, C++, Fortran, or any other language that can be linked with C.

The numerical values of derivative vectors are obtained free of truncation errors at a small multiple of the run time and random access memory required by the given function evaluation program. Derivative matrices are obtained by columns, by rows or in sparse format. For solution curves defined by ordinary differential equations, special routines are provided that evaluate the Taylor coefficient vectors and their Jacobians with respect to the current state vector. For explicitly or implicitly defined functions derivative tensors are obtained with a complexity that grows only quadratically in their degree. The derivative calculations involve a possibly substantial but always predictable amount of data. Since the data is accessed strictly sequentially it can be automatically paged out to external files.

Email: [tutores@163.com](mailto:tutores@163.com)

QQ: 749389476

**Keywords:** Computational Differentiation, Automatic Differentiation, Chain Rule, Overloading, Taylor Coefficients, Gradients, Hessians, Forward Mode, Reverse Mode, Implicit Function Differentiation, Inverse Function Differentiation

**Abbreviated title:** Automatic differentiation by overloading in C++

---

<sup>1</sup>The development of earlier versions was supported by the Office of Scientific Computing, U.S. Department of Energy, the NSF, and the Deutsche Forschungsgemeinschaft. During the development of the current version Andrea Walther and Andreas Kowarz were supported by the grant Wa 1607/2-1 of the Deutsche Forschungsgemeinschaft

<sup>2</sup>Institute of Mathematics, University of Paderborn, 33098 Paderborn, Germany

<sup>3</sup>Department of Mathematics, Humboldt-Universität zu Berlin, 10099 Berlin, Germany

# 程序代写代做 CS编程辅导

CONTENTS

## Contents



<b>1</b>	<b>Preparing a C++ Program for Differentiation</b>	<b>4</b>
1.1	Introduction	4
1.2	Declarative Programming	4
1.3	Marking the Tape	5
1.4	Selecting Independent and Dependent Variables	6
1.5	A Subprogram as an Active Section	6
1.6	Overloaded Operators and Functions	7
1.7	Reusing the Tape for Arbitrary Input Values	10
1.8	Conditional Assignments	10
1.9	Step-by-Step Modification Procedure	12
<b>2</b>	<b>Email: tutorcs@163.com</b>	<b>13</b>
2.1	Examining the Tape and Predicting Storage Requirements	14
2.2	Customizing ADOL-C	15
2.3	Warnings and Suggestions for Improved Efficiency	17
<b>3</b>	<b>Easy-To-Use Drivers</b>	<b>19</b>
3.1	Drivers for Optimization and Nonlinear Equations	19
3.2	Drivers for Ordinary Differential Equations	22
3.3	Drivers for Sparse Jacobians and Sparse Hessians	24
3.4	Higher Derivative Tensors	30
3.5	Derivatives of Implicit and Inverse Functions	33
3.6	Drivers of the Toolbox for Lie Derivatives	35
<b>4</b>	<b>Basic Drivers for the Forward and Reverse Mode</b>	<b>39</b>
4.1	Drivers for Abs-Normal Form	44
<b>5</b>	<b>Overloaded Forward and Reverse Calls</b>	<b>46</b>
5.1	The Scalar Case	47
5.2	The Vector Case	48

# 程序代写代做 CS编程辅导

## CONTENTS

3

5.3	Dependent functions . . . . .	50
<b>6</b>	<b>Advanced algorithmic differentiation in ADOL-C</b>	<b>51</b>
6.1	Differentiation of dependent functions . . . . .	51
6.2	Differentiation of functions . . . . .	53
6.3	Advanced differentiation of time integration processes . . . . .	54
6.4	Advanced algorithmic differentiation of fixed point iterations . . . . .	57
6.5	Advanced algorithmic differentiation of OpenMP parallel programs . . . . .	58
<b>WeChat: cstutorcs</b>		
<b>7</b>	<b>Traceless forward differentiation in ADOL-C</b>	<b>59</b>
7.1	Modifying the Source Code . . . . .	60
7.2	Compiling and Linking the Source Code . . . . .	64
7.3	Concluding Remarks for the Traceless Forward Mode Variant . . . . .	66
<b>Email: tutorcs@163.com</b>		
<b>8</b>	<b>Traceless forward differentiation in ADOL-C using Cuda</b>	<b>67</b>
8.1	Modifying the source code . . . . .	67
8.2	Compiling and Linking the Source Code . . . . .	72
<b>QQ: 749389476</b>		
<b>9</b>	<b>Installing and Using ADOL-C</b>	<b>72</b>
9.1	Generating the ADOL-C Library . . . . .	72
9.2	Compiling and Linking the Example Programs . . . . .	74
9.3	Description of Important Header Files . . . . .	74
9.4	Compiling and Linking C/C++ Programs . . . . .	75
9.5	Interfacing with Python and R . . . . .	76
9.6	Adding Quadratures as Special Functions . . . . .	76
<b>https://tutorcs.com</b>		
<b>10</b>	<b>Example Codes</b>	<b>77</b>
10.1	Speelpenning's Example ( <code>speelpenning.cpp</code> ) . . . . .	78
10.2	Power Example ( <code>powexam.cpp</code> ) . . . . .	79
10.3	Determinant Example ( <code>detexam.cpp</code> ) . . . . .	81
10.4	Ordinary Differential Equation Example ( <code>odexam.cpp</code> ) . . . . .	82

# 程序代写代做 CS编程辅导

4 1 PREPARING A SECTION OF C OR C++ CODE FOR DIFFERENTIATION

## 1 Preparing a Section of C or C++ Code for Differentiation

### 1.1 Introduction

The package ADOL-C acronym stands for automatic differentiation by OverLoading in C++. In contrast to source transformation, overloading does not generate intermediate source code. As starting points to retrieve further information on techniques and application of automatic differentiation, as well as on other AD tools, we refer to the book [8]. Furthermore, the web page <http://www.autodiff.org> of the AD community forms a rich source of further information and pointers.

ADOL-C facilitates the simultaneous evaluation of arbitrarily high directional derivatives and the gradients of these Taylor coefficients with respect to all independent variables. Relative to the cost of evaluating the underlying function, the cost for evaluating any such scalar-vector pair grows as the square of the degree of the derivative but is still completely independent of the numbers  $m$  and  $n$ .

This manual is organized as follows. This section explains the modifications required to convert undifferentiated code to code that compiles with ADOL-C. Section 2 covers aspects of the tape of recorded data that ADOL-C uses to evaluate arbitrarily high order derivatives. The discussion includes storage requirements and the tailoring of certain tape characteristics to fit specific user needs. Descriptions of easy-to-use drivers for a convenient derivative evaluation are contained in Section 3. Section 4 offers a more mathematical characterization of the different modes of AD to compute derivatives. At the same time, the corresponding drivers of ADOL-C are explained. The overloaded derivative evaluation routines using the forward and the reverse mode of AD are explained in Section 5. Advanced differentiation techniques as the optimal checkpointing for time integrations, the exploitation of fixed point iterations, the usages of external differentiated functions and the differentiation of OpenMP parallel programs are described in Section 6. The traceless forward mode is presented in Section 7. Section 9 details the installation and use of the ADOL-C package. Finally, Section 10 furnishes some example programs that incorporate the ADOL-C package to evaluate first and higher-order derivatives. These and other examples are distributed with the ADOL-C source code. The user should simply refer to them if the more abstract and general descriptions of ADOL-C provided in this document do not suffice.

### 1.2 Declaring Active Variables

The key ingredient of automatic differentiation by overloading is the concept of an *active variable*. All variables that may be considered as differentiable quantities at some time during the program execution must be of an active type. ADOL-C uses one active scalar type, called `adouble`, whose real part is of the standard type `double`. Typically, one will declare the independent variables and all quantities that directly or indirectly depend on

# 程序代写代做 CS编程辅导

## 1.3 Marking Active Sections

5

them as *active*. Consider, for example, as part of a computation, there is no implicit type conversion from **adouble** to any of these passive types; thus, **failure to declare variables of type adouble** will result in a compilation error message. In data flow terminology, the set of active variables names in a function is called the *active set*.



Variables that do not depend on the independent variables but enter, for example, in one of the *passive* types **double**, **float**, or **int**. There is no implicit type conversion from **adouble** to any of these passive types; thus, **failure to declare variables of type adouble** will result in a compilation error message. In data flow terminology, the set of active variables names in a function is called the *active set*.

The real component of an **adouble**  $x$  can be extracted as  $x.value()$ . In particular, such explicit conversions are needed for the standard output procedure **printf**. The output stream operator **<<** is overloaded such that first the real part of an **adouble** and then the string “(a)” is added to the stream. The input stream operator **>>** can be used to assign a constant value to an **adouble**. Naturally, **adoubles** may be components of vectors, matrices, and other arrays, as well as members of structures or classes.

The C++ class **adouble**, its member functions, and the overloaded versions of all arithmetic operations, comparison operators, and most ANSI C functions are contained in the file **adouble.cpp** and its header **<adolc/adouble.h>**. The latter must be included for compilation of all program files containing **adoubles** and corresponding operations.

Email: [tutorcs@163.com](mailto:tutorcs@163.com)

## 1.3 Marking Active Sections

All calculations involving active variables that occur between the void function calls

**trace\_on(tag,keep)**      and      **trace\_off(file)**

<https://tutorcs.com>

are recorded on a sequential data set called *tape*. Pairs of these function calls can appear anywhere in a C++ program, but they must not overlap. The nonnegative integer argument *tag* identifies the particular tape for subsequent function or derivative evaluations. Unless several tapes need to be kept, *tag* = 0 may be used throughout. The optional integer arguments *keep* and *file* will be discussed in Section 2. We will refer to the sequence of statements executed between a particular call to **trace\_on** and the following call to **trace\_off** as an *active section* of the code. The same active section may be entered repeatedly, and one can successively generate several traces on distinct tapes by changing the value of *tag*. Both functions **trace\_on** and **trace\_off** are prototyped in the header file **<adolc/taputil.h>**, which is included by the header **<adolc/adouble.h>** automatically.

Active sections may contain nested or even recursive calls to functions provided by the user. Naturally, their formal and actual parameters must have matching types. In particular, the functions must be compiled with their active variables declared as **adoubles** and with the header file **<adolc/adouble.h>** included. Variables of type **adouble** may be declared outside an active section and need not go out of scope before the end of an active section. It is not necessary – though desirable – that free-store **adoubles** allocated within an active section be deleted before its completion. The values of all **adoubles** that exist at the

# 程序代写代做 CS编程辅导

6 1 PREPARING A SECTION OF C OR C++ CODE FOR DIFFERENTIATION

beginning and end of the section. QR codes are automatically recorded by `trace_on` and `trace_off`, respectively.



## 1.4 Selecting Independent and Dependent Variables

One or more active variables  $x$  may be read in or initialized to the values of constants or passive variables  $p$  as independent variables. Other active variables that are similarly initialized may be considered as temporaries (e.g., a variable that accumulates the partial sums of a scalar product after being initialized to zero). In order to distinguish an active variable  $x$  as independent, ADOL-C requires an assignment of the form

WeChat: cstutorcs

$x \ll= px \quad // px$  of any passive numeric type .

This special initialization ensures that  $x.value() = px$ , and it should precede any other assignment to  $x$ . However,  $x$  may be reassigned other values subsequently. Similarly, one or more active variables  $y$  must be distinguished as dependent by an assignment of the form

Email: tutorcs@163.com

which ensures that  $py = y.value()$  and should not be succeeded by any other assignment to  $y$ . However, a dependent variable  $y$  may have been assigned other real values previously, and it could even be an independent variable as well. The derivative values calculated after the completion of an active section always represent **derivatives of the final values of the dependent variables with respect to the initial values of the independent variables**.

<https://tutorcs.com>

The order in which the independent and dependent variables are marked by the `<=` and `>=` statements matters crucially for the subsequent derivative evaluations. However, these variables do not have to be combined into contiguous vectors. ADOL-C counts the number of independent and dependent variable specifications within each active section and records them in the header of the tape.

## 1.5 A Subprogram as an Active Section

As a generic example let us consider a C(++) function of the form shown in Figure 1.

If `eval` is to be called from within an active C(++) section with  $x$  and  $y$  as vectors of `adoubles` and the other parameters passive, then one merely has to change the type declarations of all variables that depend on  $x$  from `double` or `float` to `adouble`. Subsequently, the subprogram must be compiled with the header file `<adolc/adouble.h>` included as described in Section 1.2. Now let us consider the situation when `eval` is still to be called with integer and real arguments, possibly from a program written in Fortran77, which does not allow overloading.

# 程序代写代做 CS编程辅导

## 1.6 Overloaded Operators and Functions

7

```
void eval() {  
    // number of independents and dependents  
    // independent variable vector  
    // dependent variable vector  
    // integer parameters  
    // real parameters  
    // beginning of function body  
    double t = 0; // local variable declaration  
    for (int i=0; i < n; i++) // begin of computation  
        t += z[i]*x[i]; // continue  
    ..... // continue  
    y[m-1] = t/m; // end of computation  
} // end of function
```

WeChat: cstutorcs

## Assignment Project Exam Help

Figure 1: Generic example of a subprogram to be activated

To automatically compute derivatives of the dependent variables with respect to the independent variables  $x$ , we can make the body of the function into an active section. For example, we may modify the previous program segment as in Figure 2. The renaming and doubling up of the original independent and dependent variable vectors by active counterparts may seem at first a bit clumsy. However, this transformation has the advantage that the calling sequence and the computational part, i.e., where the function is really evaluated, of `eval` remain completely unaltered. If the temporary variable  $t$  had remained a `double`, the code would not compile, because of a type conflict in the assignment following the declaration. More detailed example codes are listed in Section 10.

## 1.6 Overloaded Operators and Functions

As in the subprogram discussed above, the actual computational statements of a C(++) code need not be altered for the purposes of automatic differentiation. All arithmetic operations, as well as the comparison and assignment operators, are overloaded, so any or all of their operands can be an active variable. An `adouble x` occurring in a comparison operator is effectively replaced by its real value `x.value()`. Most functions contained in the ANSI C standard for the math library are overloaded for active arguments. The only exceptions are the non-differentiable functions `fmod` and `modf`. Otherwise, legitimate C code in active sections can remain completely unchanged, provided the direct output of active variables is avoided. The rest of this subsection may be skipped by first time users who are not worried about marginal issues of differentiability and efficiency.

The modulus `fabs(x)` is everywhere Lipschitz continuous but not properly differentiable at the origin, which raises the question of how this exception ought to be handled. For-

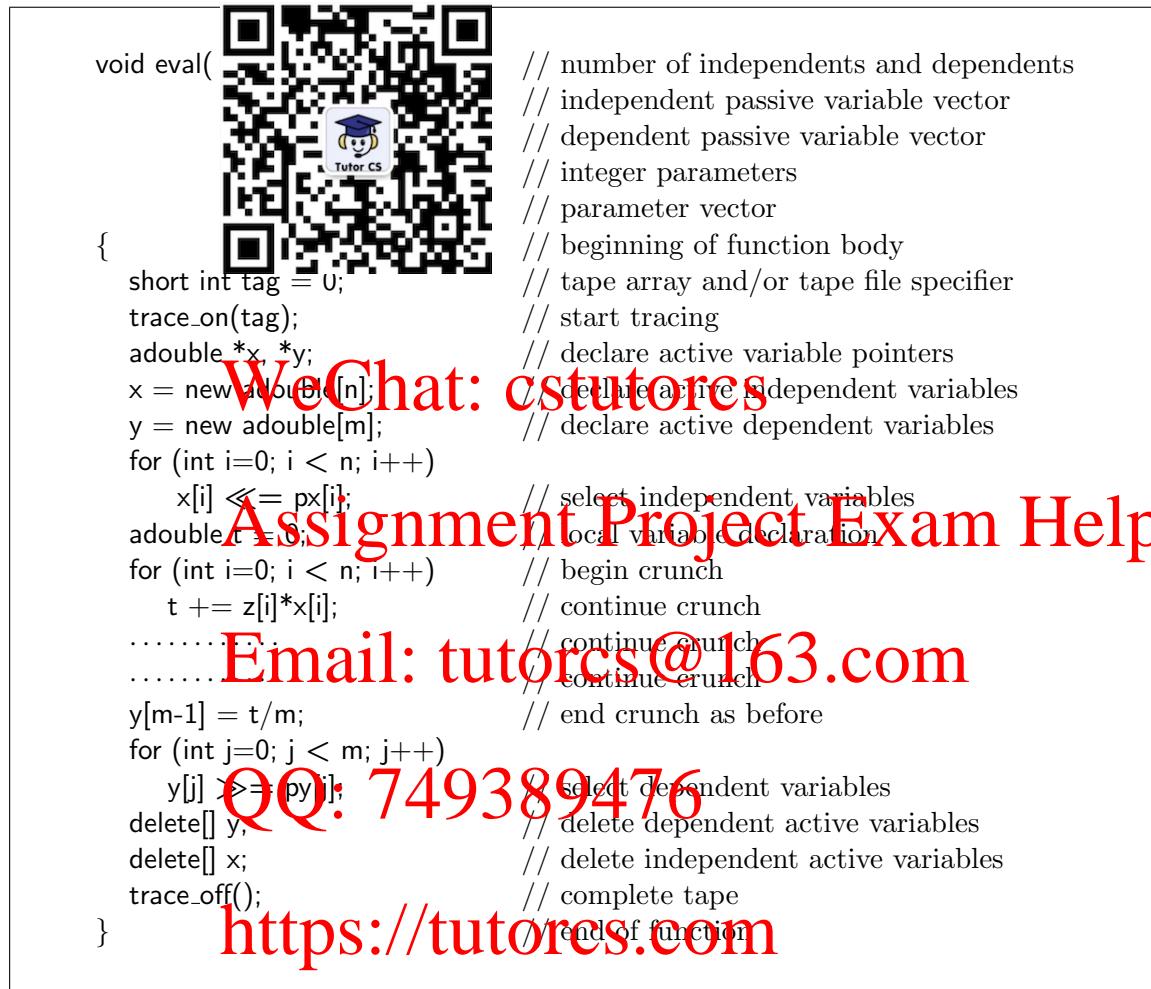


Figure 2: Activated version of the code listed in Figure 1

tunately, one can easily see that `fabs(x)` and all its compositions with smooth functions are still directionally differentiable. These directional derivatives of arbitrary order can be propagated in the forward mode without any ambiguity. In other words, the forward mode as implemented in ADOL-C computes Gateaux derivatives in certain directions, which reduce to Fréchet derivatives only if the dependence on the direction is linear. Otherwise, the directional derivatives are merely positively homogeneous with respect to the scaling of the directions. For the reverse mode, ADOL-C sets the derivative of `fabs(x)` at the origin somewhat arbitrarily to zero.

We have defined binary functions `fmin` and `fmax` for `adouble` arguments, so that function and derivative values are obtained consistent with those of `fabs` according to the identities

$$\min(a, b) = [a + b - |a - b|]/2 \quad \text{and} \quad \max(a, b) = [a + b + |a - b|]/2 .$$

# 程序代写代做 CS编程辅导

## 1.6 Overloaded Operators and Functions

9

These relations do not hold if  $a$  or  $b$  is infinite, in which case  $fmin$  or  $fmax$  and their derivatives may return different values. It should be noted that the directional differentiation of  $fmin$  and  $fmax$  can yield different results from the corresponding assignment statement based on the sign of  $a - b$ . For example, the statement

b)  $c = a$ ; else  $c = b$ ;

yields for  $a = b$  and  $a \neq b$  the incorrect directional derivative value  $c' = b'$  rather than the correct  $c' = a'$ . Therefore this form of conditional assignment should be avoided by use of the function  $fmin(a, b)$ . There are also versions of  $fmin$  and  $fmax$  for two passive arguments and mixed passive/active arguments; these are handled by implicit conversion. On the function class obtained by composing the modulus with real analytic functions, the concept of directional differentiation can be extended to the propagation of unique one-sided Taylor expansions. The branches taken by  $\text{fabs}$ ,  $fmin$ , and  $fmax$ , are recorded on the tape.

## Assignment Project Exam Help

The functions  $\text{sqrt}$ ,  $\text{pow}$ , and some inverse trigonometric functions have infinite slopes at the boundary points of their domains. At these marginal points the derivatives are set by ADOL-C to either  $\pm\text{InfVal}$ , 0 or  $\text{NoNum}$ , where  $\text{InfVal}$  and  $\text{NoNum}$  are user-defined parameters, see Section 2.2. On IEEE machines  $\text{InfVal}$  can be set to the special value  $\text{Inf} = 1.0/0.0$  and  $\text{NoNum}$  to  $\text{NaN} = 0.0/0.0$ . For example, at  $a=0$  the first derivative  $b'$  of  $b = \text{sqrt}(a)$  is set to

$$\text{QQ: } 749389476 \quad b' = \begin{cases} \text{InfVal} & \text{if } a' > 0 \\ 0 & \text{if } a' = 0 \\ \text{NoNum} & \text{if } a' < 0 \end{cases} .$$

In other words, we consider  $a$  and consequently  $b$  as a constant when  $a'$  or more generally all computed Taylor coefficients are zero.

<https://tutorcs.com>

The general power function  $\text{pow}(x, y) = x^y$  is computed whenever it is defined for the corresponding double arguments. If  $x$  is negative, however, the partial derivative with respect to an integral exponent is set to zero. The derivatives of the step functions  $\text{floor}$ ,  $\text{ceil}$ ,  $\text{frexp}$ , and  $\text{lDEXP}$  are set to zero at all arguments  $x$ . The result values of the step functions are recorded on the tape and can later be checked to recognize whether a step to another level was taken during a forward sweep at different arguments than at taping time.

Some C implementations supply other special functions, in particular the error function  $\text{erf}(x)$ . For the latter, we have included an `adouble` version in `<adouble.cpp>`, which has been commented out for systems on which the `double` valued version is not available. The increment and decrement operators `++`, `--` (prefix and postfix) are available for `adoubles`. Ambiguous statements like `a += a++;` must be avoided because the compiler may sequence the evaluation of the overloaded expression differently from the original in terms of `doubles`.

As we have indicated above, all subroutines called with active arguments must be modified or suitably overloaded. The simplest procedure is to declare the local variables of the function as active so that their internal calculations are also recorded on the tape. Unfortunately, this approach is likely to be unnecessarily inefficient and inaccurate if the original

subroutine evaluates mathematical problems such as the solution of a particular mathematical problem. Examples are implicit functions, quadratures, and solutions of ordinary differential equations. Often the numerical methods for evaluating such special functions and their internal workings are not at all differentiable. By going through such an adaptive procedure, one can obtain derivatives in the mathematical definition of the special function. Currently this differentiation is implemented only for user-supplied quadratures as described in Section 9.6.



## 1.7 Reusing the Tape for Arbitrary Input Values

In some situations it may be desirable to calculate the value and derivatives of a function at arbitrary arguments by using a tape of the function evaluation at one argument and reevaluating the function and its derivatives using the given ADOL-C routines. This approach can significantly reduce run times, and it also allows to port problem functions, in the form of the corresponding tape files, into a computing environment that does not support C++ but does support C or Fortran. Therefore, the routines provided by ADOL-C for the evaluation of derivatives can be used at arguments  $x$  other than the point at which the tape was generated, provided there are no user defined quadratures and all comparisons involving `adoubles` yield the same result. The last condition implies that the control flow is unaltered by the change of the independent variable values. Therefore, this sufficient condition is tested by ADOL-C and if it is not met the ADOL-C routine called for derivative calculations indicates this contingency through its return value. Currently, there are six return values, see Table 1.

<https://tutorcs.com>

+3	The function is locally analytic.
+2	The function is locally analytic but the sparsity structure (compared to the situation at the taping point) may have changed, e.g. while at taping arguments <code>fmax(a,b)</code> returned <code>a</code> we get <code>b</code> at the argument currently used.
+1	At least one of the functions <code>fmin</code> , <code>fmax</code> or <code>fabs</code> is evaluated at a tie or zero, respectively. Hence, the function to be differentiated is Lipschitz-continuous but possibly non-differentiable.
0	Some arithmetic comparison involving <code>adoubles</code> yields a tie. Hence, the function to be differentiated may be discontinuous.
-1	An <code>adouble</code> comparison yields different results from the evaluation point at which the tape was generated.
-2	The argument of a user-defined quadrature has changed from the evaluation point at which the tape was generated.

Table 1: Description of return values

# 程序代写代做 CS编程辅导

1.8 Conditional Assignments

11

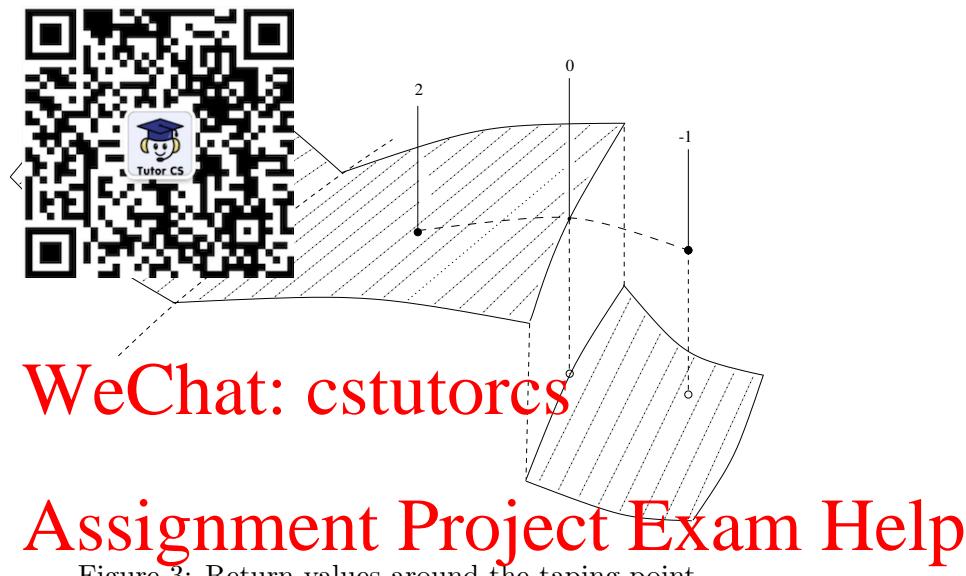


Figure 3: Return values around the taping point

Email: [tutorcs@163.com](mailto:tutorcs@163.com)

In Figure 3 these return values are illustrated. If the user finds the return value of an ADOL-C routine to be negative the taping process simply has to be repeated by executing the active section again. The crux of the problem lies in the fact that the tape records only the operations that are executed during one particular evaluation of the function. It also has no way to evaluate integrals since the corresponding quadratures are never recorded on the tape. Therefore, when there are user-defined quadratures the retaping is necessary at each new point. If there are only branches conditioned on `adouble` comparisons one may hope that retaping becomes unnecessary when the points settle down in some small neighborhood, as one would expect for example in an iterative equation solver.

QQ: 749389476

<https://tutorcs.com>

## 1.8 Conditional Assignments

It appears unsatisfactory that, for example, a simple table lookup of some physical property forces the re-recording of a possibly much larger calculation. However, the basic philosophy of ADOL-C is to overload arithmetic, rather than to generate a new program with jumps between “instructions”, which would destroy the strictly sequential tape access and require the infusion of substantial compiler technology. Therefore, we introduce the two constructs of conditional assignments and active integers as partial remedies to the branching problem.

In many cases, the functionality of branches can be replaced by conditional assignments. For this purpose, we provide a special function called `condassign(a,b,c,d)`. Its calling sequence corresponds to the syntax of the conditional assignment

$$a = (b > 0) ? c : d;$$

# 程序代写代做 CS编程辅导

## 12 1 PREPARING A SECTION OF C OR C++ CODE FOR DIFFERENTIATION

which C++ inherits from C. However, here the arguments are restricted to be active or passive scalar arguments. In C, expression arguments are evaluated before the test on b, which is different in C++ where the test is part of the conditional assignment or the code segment.

Suppose the code segment contains the code segment

0)  $a = c$ ; else  $a = d$ ;

Here, only one of the expressions (or, more generally, program blocks)  $c$  and  $d$  is evaluated, which exactly constitutes the problem for ADOL-C. To obtain the correct value  $a$  with ADOL-C, one may first execute both branches and then pick either  $c$  or  $d$  using `condassign(a,b,c,d)`. To maintain consistency with the original code, one has to make sure that the two branches do not have any side effects that can interfere with each other or may be important for subsequent calculations. Furthermore the test parameter  $b$  has to be an `adouble` or an `adouble` expression. Otherwise the test condition  $b$  is recorded on the tape as a *constant* with its run time value. Thus the original dependency of  $b$  on active variables gets lost, for instance if  $b$  is a comparison expression, see Section 1.6. If there is no `else` part in a conditional assignment, one may call the three argument version `condassign(a,b,c)`, which is logically equivalent to `condassign(a,b,c,a)` in that nothing happens if  $b$  is non-positive. The header file `<adolc/adouble.h>` contains also corresponding definitions of `condassign(a,b,c,d)` and `condassign(a,b,c)` for passive double arguments so that the modified code without any differentiation can be tested for correctness.

A generalization of this concept for more than two branches, e.g., akin to a `switch` statement or a cascade of `if...else if`, may be done by enabling `ADOLC_ADVANCED_BRANCHING` and performing selection on elements of an `advect` with active indices.

<https://tutorcs.com>

### 1.9 Step-by-Step Modification Procedure

To prepare a section of given C or C++ code for automatic differentiation as described above, one applies the following step-by-step procedure.

1. Use the statements `trace_on(tag)` or `trace_on(tag,keep)` and `trace_off()` or `trace_off(file)` to mark the beginning and end of the active section.
2. Select the set of active variables, and change their type from `double` or `float` to `adouble`.
3. Select a sequence of independent variables, and initialize them with `<<=` assignments from passive variables or vectors.
4. Select a sequence of dependent variables among the active variables, and pass their final values to passive variable or vectors thereof by `>>=` assignments.
5. Compile the codes after including the header file `<adolc/adouble.h>`.

Typically, the first attempt to perform standard I/O of active variables. Since activated by a mechanical application of the procedure above, the following only to advanced users.



## 2 Number and Controlling the Buffer

The trace generated by the execution of an active section may stay within a triplet of internal arrays or it may be written out to three corresponding files. We will refer to these triplets as the tape array or tape file, in general tape which may subsequently be used to evaluate the underlying function and its derivatives at the original point or at alternative arguments. If the active section involves user-defined quadratures it must be executed and re-taped at each new argument. Similarly, if conditions on adouble values lead to a different program branch being taken at a new argument, the evaluation process also needs to be re-taped at the new point. Otherwise, direct evaluation from the tape by the routine function (Section 3.1) is likely to be faster. The use of quadratures and the results of all comparisons on adoubles are recorded on the tape so that function and other forward routines stop and return appropriate flags if their use without prior re-taping is unsafe. To avoid any re-taping certain types of branches can be recorded on the tape through the use of conditional assignments described before in Section 1.8.

QQ: 749389476

Several tapes may be generated and kept simultaneously. A tape array is used as a triplet of buffers or a tape file is generated if the length of any of the buffers exceeds the maximal array lengths of OBUFSIZE, VBUFSIZE or LBUFSIZE. These parameters are defined in the header file `<adolc/internal/usparms.h>` and may be adjusted by the user in the header file before compiling the ADOL-C library, or on runtime using a file named `.adolcrc`. Lines in this file must have the form

"VARIABLE" = "VALUE"

where the quotation marks are mandatory. The filesystem folder, where the tapes files may be written to disk, can be changed by changing the definition of `TAPE_DIR` in the header file `<adolc/dvlparms.h>` before compiling the ADOL-C library, or on runtime by defining `TAPE_DIR` in the `.adolcrc` file. By default this is defined to be the present working directory (.).

For simple usage, `trace_on` may be called with only the tape `tag` as argument, and `trace_off` may be called without argument. The optional integer argument `keep` of `trace_on` determines whether the numerical values of all active variables are recorded in a buffered temporary array or file called the taylor stack. This option takes effect if `keep = 1` and prepares the scene for an immediately following gradient evaluation by a call to a routine implementing the reverse mode as described in the Section 4 and Section 5. A file is used instead of an array if the size exceeds the maximal array length of `TBUFSIZE` defined in

# 程序代写代做 CS 编程辅导

## 2 NUMBERING THE TAPES AND CONTROLLING THE BUFFER

<adolc/internal.h>



may be adjusted in the same way like the other buffer sizes mentioned above. The gradients may be evaluated by a call to `gradient`, which includes a parameter `trace_on` for the creation of the temporary file. If omitted, the argument `keep_d` indicates that no temporary taylor stack file is generated.

By setting the argument file of `trace_off` to 1, the user may force a numbered tape file to be created if the tape array (buffer) does not overflow. If the argument file is chosen to be 0, so that the tape array is written onto a tape file only if the length of any of the buffers exceeds `[OLVT]BUFSIZE` elements.

After the execution of an active section, if a tape file was generated, i.e., if the length of some buffer exceeded `[OLV/T]BUFSIZE` elements or if the argument file of `trace_off` was set to 1, the files will be saved in the directory defined as `ADOLC_TAPE_DIR` (by default the current working directory) under filenames formed by the strings `ADOLC_OPERATIONS_NAME`, `ADOLC_LOCATIONS_NAME`, `ADOLC_VALUES_NAME` and `ADOLC_TAYLORS_NAME` defined in the header file <`adolc/dv1_params.h`> appended with the number given as the `tag` argument to `trace_on` and have the extension `.tap`.

Later, all problem-independent routines like `gradient`, `jacobian`, `forward`, `reverse`, and others expect as first argument a tag to determine the type on which their respective computational task is to be performed. By calling `trace_on` with different tape tags, one can create several tapes for various function evaluations and subsequently perform function and derivative evaluations on one or more of them.

For example, suppose one wishes to calculate for two smooth functions  $f_1(x)$  and  $f_2(x)$

$$f(x) = \max\{f_1(x), f_2(x)\}, \quad \nabla f(x),$$

and possibly higher derivatives where the two functions do not tie. Provided  $f_1$  and  $f_2$  are evaluated in two separate active sections, one can generate two different tapes by calling `trace_on` with `tag = 1` and `tag = 2` at the beginning of the respective active sections. Subsequently, one can decide whether  $f(x) = f_1(x)$  or  $f(x) = f_2(x)$  at the current argument and then evaluate the gradient  $\nabla f(x)$  by calling `gradient` with the appropriate argument value `tag = 1` or `tag = 2`.

### 2.1 Examining the Tape and Predicting Storage Requirements

At any point in the program, one may call the routine

```
void tapestats(unsigned short tag, size_t* counts)
```

# 程序代写代做 CS编程辅导

with counts being eleven integers. The first argument tag specifies the particular tape component of counts represent

- c[0]: number of independents, i.e. calls to  $\ll=$ ,
- c[1]: number of dependents, i.e. calls to  $\gg=$ ,
- c[2]: total number of live active variables,
- c[3]: taylor stack (number of overwrites),
- c[4]: tsize (a multiple of eight),

counts[5]: the total number of operations recorded,  
counts[6-13]: other internal information about the tape.

The values `maxlive = counts[1]` and `tsize = counts[3]` determine the temporary storage requirements during calls to the routines implementing the forward and the reverse mode. For a certain degree  $\deg \geq 0$ , the scalar version of the forward mode involves apart from the tape buffers an array of  $(\deg+1)*\text{maxlive}$  doubles in core and, in addition, a sequential data set called the value stack of `tsize*keep` `revreals` if called with the option `keep > 0`. Here the type `revreal` is defined as `double` or `float`. The latter choice halves the storage requirement for the sequential data set, which stays in core if its length is less than `TBUFSIZE` bytes and is otherwise written out to a temporary file. The parameter `TBUFSIZE` is defined in the header file `<adolc/internal/usrparms.h>`. The drawback of the economical `revreal = float` choice is that subsequent calls to reverse mode implementations yield gradients and other adjoint vectors only in single-precision accuracy. This may be acceptable if the adjoint vectors represent rows of a Jacobian that is used for the calculation of Newton steps. In its scalar version, the reverse mode implementation involves the same number of doubles and twice as many `revreals` as the forward mode implementation. The storage requirements of the vector versions of the forward mode and reverse mode implementation are equal to that of the scalar versions multiplied by the vector length.

WeChat: cstutorcs  
Assignment Project Exam Help  
Email: tutorcs@163.com  
QQ: 749389476  
<https://tutorcs.com>

## 2.2 Customizing ADOL-C

Based on the information provided by the routine `tapestats`, the user may alter the following types and constant dimensions in the header file `<adolc/internal/usrparms.h>` to suit his problem and environment.

**OBUFSIZE, LBUFSIZE, VBUFSIZE:** These integer determines the length of internal buffers (default: 524 288). If the buffers are large enough to accommodate all required data, any file access is avoided unless `trace_off` is called with a positive argument. This desirable situation can be achieved for many problem functions with an execution trace of moderate size. Primarily these values occur as an argument to `malloc`, so that setting it unnecessarily large may have no ill effects, unless the operating system prohibits or penalizes large array allocations. It is however recommended to leave the values in `<adolc/internal/usrparms.h>` unchanged and set them using the `.adolcrc` file in the current working directory at runtime.

# 程序代写代做 CS 编程辅导

## 2 NUMBERING THE TAPES AND CONTROLLING THE BUFFER

**TBUFSIZE:** This  the length of the internal buffer for a taylor stack (default: 5).

**TBUFNUM:** This  the maximal number of taylor stacks (default: 32).

**fint:** The integer  Fortran callable versions of functions.

**fdouble:** The float  used by Fortran callable versions of functions.

**inf\_num:** This together with **inf\_den** sets the “vertical” slope  $\text{InfVal} = \text{inf\_num}/\text{inf\_den}$  of special functions at the boundaries of their domains (default:  $\text{inf\_num} = 1.0$ ). On IEEE machines the default setting produces the standard **Inf**. On non-IEEE machines change these values to produce a small **InfVal** value and compare the results of two forward sweeps with different **InfVal** settings to detect a “vertical” slope.

**inf\_den:** See **inf\_num** (default: 0.0).

## Assignment Project Exam Help

**non\_num:** This together with **non\_den** sets the mathematically undefined derivative value  $\text{NoNum} = \text{non\_num}/\text{non\_den}$  of special functions at the boundaries of their domains (default:  $\text{non\_num} = 0.0$ ). On IEEE machines the default setting produces the standard **NaN**. On non-IEEE machines change these values to produce a small **NoNum** value and compare the results of two forward sweeps with different **NoNum** settings to detect the occurrence of undefined derivative values.

**non\_den:** See **non\_num** (default: 0.0).

**ADOLC\_EPS:** For testing on small numbers to avoid overflows (default: 10E-20).

**DIAG\_OUT:** File identifier used as standard output for ADOL-C diagnostics (default: **stdout**).

The following types and options may be set using the command-line options of the **./configure** script.

**locint:** The range of the integer type **locint** determines how many **adoubles** can be simultaneously alive (default: **unsigned int**). In extreme cases when there are more than  $2^{32}$  **adoubles** alive at any one time, the type **locint** must be changed to **unsigned long**. This can be done by passing **--enable-ulng** to **./configure**.

**revreal:** The choice of this floating-point type trades accuracy with storage for reverse sweeps (default: **double**). While functions and their derivatives are always evaluated in double precision during forward sweeps, gradients and other adjoint vectors are obtained with the precision determined by the type **revreal**. The less accurate choice **revreal = float** nearly halves the storage requirement during reverse sweeps. This can be done by passing **--disable-double** to **./configure**.

# 程序代写代做 CS编程辅导

## 2.3 Warnings and Suggestions for Improved Efficiency

17

ATRIG\_ERF: The implementation of the inverse hyperbolic functions and the error function are enabled by default (defined) by passing --enable-atrig-erf to ./configure

ADOLC\_USE\_CALLOC: The memory allocation routine used by ADOL-C. Malloc will be used if undefined. ADOLC\_USE\_CALLOC is defined by default to avoid incorporation of uninitialized memory. It can be set undefined by passing --enable-calloc-to ./configure.

ADOLC\_ADVANCED\_BRANCHING: Enables routines required for automatic branch selection (default: disabled). The boolean valued comparison operators with two `adouble` type arguments will not return boolean values anymore and may not be used in branch control statements (`if`, `while`, `for` etc.). Instead conditional assignments using `condassign` or selection operations on elements of `advector` type should be used. Enabling this option and rewriting the function evaluation using `condassign` or selections of `advector` elements will prevent the need for reworking the function in branch switches. This can be enabled by passing --enable-advanced-branching to ./configure.

Email: [tutorcs@163.com](mailto:tutorcs@163.com)

## 2.3 Warnings and Suggestions for Improved Efficiency

Since the type `adouble` has a nontrivial constructor, the mere declaration of large `adouble` arrays may take up considerable run time. The user should be warned against the usual Fortran practice of declaring fixed-size arrays that can accommodate the largest possible case of an evaluation program with variable dimensions. If such programs are converted to or written in C, the overloading in combination with ADOL-C will lead to very large run time increases for comparatively small values of the problem dimension, because the actual computation is completely dominated by the construction of the large `adouble` arrays. The user is advised to create dynamic arrays of `adoubles` by using the C++ operator `new` and to destroy them using `delete`. For storage efficiency it is desirable that dynamic objects are created and destroyed in a last-in-first-out fashion.

Whenever an `adouble` is declared, the constructor for the type `adouble` assigns it a nominal address, which we will refer to as its *location*. The location is of the type `locint` defined in the header file `<adolc/internal/usrparms.h>`. Active vectors occupy a range of contiguous locations. As long as the program execution never involves more than 65 536 active variables, the type `locint` may be defined as `unsigned short`. Otherwise, the range may be extended by defining `locint` as `(unsigned) int` or `(unsigned) long`, which may nearly double the overall mass storage requirement. Sometimes one can avoid exceeding the accessible range of `unsigned shorts` by using more local variables and deleting `adoubles` created by the `new` operator in a last-in-first-out fashion. When memory for `adoubles` is requested through a call to `malloc()` or other related C memory-allocating functions, the storage for these `adoubles` is allocated; however, the C++ `adouble` constructor is never called. The newly defined `adoubles` are never assigned a location and are not counted in the stack of live variables. Thus,

# 程序代写代做 CS 编程辅导

## 2 NUMBERING THE TAPES AND CONTROLLING THE BUFFER

any results depend on the order of creation of adoubles. If adoubles are created in one order and destroyed in another, the results will be incorrect. For these reasons **DO NOT use malloc or new to declare adoubles** (see the discussion of memory allocation in Chapter 1).

When an adouble is created, it is placed in a contiguous range of locations. If it is explicitly deleted, the destructor notices that the location(s) may have been reallocated by a subsequent (nominal) reallocation. In general, this is not done immediately. Instead, the destructor places the locations to be deallocated form a contiguous tail of all locations created.

As a consequence of this allocation scheme, the currently alive adouble locations always form a contiguous range of integers that grows and shrinks like a stack. Newly declared adoubles are placed on the top so that vectors of adoubles obtain a contiguous range of locations. While the C++ compiler can be expected to construct and destruct automatic variables in a last-in-first-out fashion, the user may upset this desirable pattern by deleting free-store adoubles too early or too late. Then the adouble stack may grow unnecessarily, but the numerical results will still be correct, unless an exception occurs because the range of locint is exceeded. In general, free-store adoubles should be deleted in a last-in-first-out fashion toward the end of the program block in which they were created. When this pattern is maintained, the maximum number of adoubles alive and, as a consequence, the randomly accessed storage space of the derivative evaluation routines is bounded by a small multiple of the memory used in the relevant section of the original program. Failure to delete dynamically allocated adoubles may cause that the maximal number of adoubles alive at one time will be exceeded in the same active section is called repeatedly. The same effect occurs if static adoubles are used.

To avoid the storage and manipulation of structurally trivial derivative values, one should pay careful attention to the naming of variables. Ideally, the intermediate values generated during the evaluation of a vector function should be assigned to program variables that are consistently either active or passive, in that all their values either are or are not dependent on the independent variables in a nontrivial way. For example, this rule is violated if a temporary variable is successively used to accumulate inner products involving first only passive and later active arrays. Then the first inner product and all its successors in the data dependency graph become artificially active and the derivative evaluation routines described later will waste time allocating and propagating trivial or useless derivatives. Sometimes even values that do depend on the independent variables may be of only transitory importance and may not affect the dependent variables. For example, this is true for multipliers that are used to scale linear equations, but whose values do not influence the dependent variables in a mathematical sense. Such dead-end variables can be deactivated by the use of the value function, which converts adoubles to doubles. The deleterious effects of unnecessary activity are partly alleviated by run time activity flags in the derivative routine hov\_reverse presented in Section 4.

The adouble default constructor sets to zero the associated value. This implies a certain overhead that may seem unnecessary when no initial value is actually given, however, the implicit initialization of arrays from a partial value list is the only legitimate construct



**WeChat: cstutors**  
**Assignment Project Exam Help**  
**Email: tutorcs@163.com**  
**QQ: 749389476**

# 程序代写代做 CS编程辅导

19

(known to us) the behavior. An array instantiation such as



```
adouble x[3]={2.0};
```

will initialize  $x[0]$  to 2.0 and (implicitly) the remaining array elements  $x[1]$  and  $x[2]$  to 0.0. According to the array element construction of the type changed instantiation

```
adouble x[3]={2.0};
```

will use the constructor `adouble(const double&)` for  $x[0]$  passing in 2.0 but will call the `adouble` default constructor for  $x[1]$  and  $x[2]$  leaving these array elements uninitialized *unless* the default constructor does implement the initialization to zero. The C++ constructor syntax does not provide a means to distinguish this implicit initialization from the declaration of any simple uninitialized variable. If the user can ascertain the absence of array instantiations such as the above then one can configure ADOL-C with the `--disable-stdczero` option , see Section 9.1, to avoid the overhead of these initializations.

**WeChat: cstutorcs**

**Assignment Project Exam Help**

**Email: tutorcs@163.com**

## 3 Easy-To-Use Drivers

For the convenience of the user, ADOL-C provides several easy-to-use drivers that compute the most frequently required derivative objects. Throughout, we assume that after the execution of an active section, the corresponding tape with the identifier `tag` contains a detailed record of the computational process by which the final values  $y$  of the dependent variables were obtained from the values  $x$  of the independent variables. We will denote this functional relation between the input variables  $x$  and the output variables  $y$  by

$$F : \mathbb{R}^n \mapsto \mathbb{R}^m, \quad x \rightarrow F(x) \equiv y.$$

The return value of all drivers presented in this section indicate the validity of the tape as explained in Section 1.7. The presented drivers are all C functions and therefore can be used within C and C++ programs. Some Fortran-callable companions can be found in the appropriate header files.

### 3.1 Drivers for Optimization and Nonlinear Equations

The drivers provided for solving optimization problems and nonlinear equations are prototyped in the header file `<adolc/drivers/drivers.h>`, which is included automatically by the global header file `<adolc/adolc.h>` (see Section 9.3).

The routine function allows to evaluate the desired function from the tape instead of executing the corresponding source code:

# 程序代写代做 CS 编程辅导

3 EASY-TO-USE DRIVERS

```

int function(double x[], double y[]);
short int tag; // tape identification
int m; // number of dependent variables m
int n; // number of independent variables n
double x[]; // independent vector x
double y[]; // dependent vector y = F(x)

```



If the original evaluation program is available this double version should be used to compute the function value in order to avoid the interpretative overhead.

For the calculation of whole derivative vectors and matrices up to order 2 there are the following procedures:

**Assignment Project Exam Help**  
**Email: tutorcs@163.com**  
**QQ: 749389476**  
**<https://tutorcs.com>**

```

int gradient(tag,n,x,g)
short int tag; // tape identification
int n; // number of independent variables n and m = 1
double x[n]; // independent vector x
double g[n]; // resulting gradient  $\nabla F(x)$ 

int jacobian(tag,m,n,x,J)
short int tag; // tape identification
int m; // number of dependent variables m
int n; // number of independent variables n
double x[n]; // independent vector x
double J[m][n]; // resulting Jacobian  $F'(x)$ 

int hessian(tag,n,x,H)
short int tag; // tape identification
int n; // number of independent variables n and m = 1
double x[n]; // independent vector x
double H[n][n]; // resulting Hessian matrix  $\nabla^2 F(x)$ 

```

The driver routine `hessian` computes only the lower half of  $\nabla^2 f(x_0)$  so that all values  $H[i][j]$  with  $j > i$  of `H` allocated as a square array remain untouched during the call of `hessian`. Hence only  $i + 1$  doubles need to be allocated starting at the position `H[i]`.

To use the full capability of automatic differentiation when the product of derivatives with certain weight vectors or directions are needed, ADOL-C offers the following four drivers:

```

int vec_jac(tag,m,n,repeat,x,u,z)
short int tag; // tape identification
int m; // number of dependent variables m

```

# 程序代写代做 CS编程辅导

3.1 Drivers for Optimization and Nonlinear Equations

21

```
int n;          // number of independent variables n
int repeat;    // indicate repeated call at same argument
double x[n];   // independent vector x
double u[m];   // range weight vector u
double z[n];   // result  $z = u^T F'(x)$ 
```

If a nonzero value  indicates that the routine `vec_jac` has been called at the same argument immediately before, the internal forward mode evaluation will be skipped and only reverse mode evaluation with the corresponding arguments is executed resulting in a reduced computational complexity of the function `vec_jac`.

WeChat: cstutorcs

```
int jac_vec(tag,m,n,x,v,z)
short int tag;           // tape identification
int m;                  // number of dependent variables m
int n;                  // number of independent variables n
double x[n];            // independent vector x
double v[n];            // tangent vector v
double z[n];            // result  $z = F(x)v$ 
```

Email: tutorcs@163.com

```
int hess_vec(tag,n,x,v,z)
short int tag;           // tape identification
int n;                  // number of independent variables n
double x[n];            // independent vector x
double v[n];            // tangent vector v
double z[n];            // result  $z = \nabla^2 F(x)v$ 
```

```
int hess_mat(tag,n,p,x,V,Z)
short int tag;           // tape identification
int n;                  // number of independent variables n
int p;                  // number of columns in V
double x[n];            // independent vector x
double V[n][p];         // tangent matrix V
double Z[n][p];         // result  $Z = \nabla^2 F(x)V$ 
```

```
int lagra_hess_vec(tag,m,n,x,v,u,h)
short int tag;           // tape identification
int m;                  // number of dependent variables m
int n;                  // number of independent variables n
double x[n];            // independent vector x
double v[n];            // tangent vector v
double u[m];            // range weight vector u
double h[n];            // result  $h = u^T \nabla^2 F(x)v$ 
```

Assignment Project Exam Help

QQ: 749389476

<https://tutorcs.com>

# 程序代写代做 CS编程辅导

22

3 EASY-TO-USE DRIVERS

The next procedure performs Newton steps only having the corresponding tape at hand:

```
int jac_solv()
short int tag; // tape identification
int n; // number of independent variables n
double x[i]; // independent vector x as
double b[n]; // in: right-hand side b, out: result w of F(x)w = b
int mode; // option to choose different solvers
```

## WeChat: cstutorcs

On entry, parameter `b` of the routine `jac_solv` contains the right-hand side of the equation  $F(x)w = b$  to be solved. On exit, `b` equals the solution `w` of this equation. If `mode = 0` only the Jacobian of the function given by the tape labeled with `tag` is provided internally. The LU-factorization of this Jacobian is computed to: `mode = 1`. The solution of the equation is calculated if `mode = 2`. Hence, it is possible to compute the LU-factorization only once. Then the equation can be solved for several right-hand sides `b` without calculating the Jacobian and its factorization again.

Email: [tutorcs@163.com](mailto:tutorcs@163.com)

QQ: 749389476

## 3.2 Drivers for Ordinary Differential Equations

When  $F$  is the right-hand side of an (autonomous) ordinary differential equation

$$x'(t) = F(x(t)),$$

we must have  $m = n$ . Along any solution path  $x(t)$  its Taylor coefficients  $x_j$  at some time, e.g.,  $t = 0$ , must satisfy the relation

$$x_{i+1} = \frac{1}{1+i} y_i.$$

with the  $y_j$  the Taylor coefficients of its derivative  $y(t) = x'(t)$ , namely,

$$y(t) \equiv F(x(t)) : \mathbb{R} \mapsto \mathbb{R}^m$$

defined by an autonomous right-hand side  $F$  recorded on the tape. Using this relation, one can generate the Taylor coefficients  $x_i$ ,  $i \leq \deg$ , recursively from the current point  $x_0$ . This task is achieved by the driver routine `forode` defined as follows:

# 程序代写代做 CS编程辅导

## 3.2 Drivers for Ordinary Differential Equations

23

```
int forode  
short int t; // tape identification  
int n; // number of state variables n  
double tau; // scaling parameter  
int dol; // degree on previous call  
int deg; // degree on current call  
double X[]; // Taylor coefficient vector X
```



If `dol` is positive, it is assumed that `forode` has been called before at the same point so that all Taylor coefficient vectors up to the `dol`-th are already correct.

WeChat: cstutorcs

Subsequently one may call the driver routine `reverse` or corresponding low level routines as explained in the Section 5 and Section 4, respectively, to compute the family of square matrices  $Z[n][n][deg]$  defined by

**Assignment Project Exam Help**

$$Z_j \equiv U \frac{\partial y_j}{\partial x_0} \in \mathbb{R}^{q \times n},$$

with `double** U = In`, the identity matrix of order  $n$ .

Email: tutorcs@163.com

For the numerical solutions of ordinary differential equations, one may also wish to calculate the Jacobians

QQ: 749389476

which exist provided  $F$  is sufficiently smooth. These matrices can be obtained from the partial derivatives  $\partial y_i / \partial x_0$  by an appropriate version of the chain rule. To compute the total derivatives  $B = (B_{ij})_{0 \leq i < d}$  defined in (1), one has to evaluate  $\frac{1}{2}d(d-1)$  matrix-matrix products. This can be done by a call of the routine `accode` after the corresponding evaluation of the `hov_reverse` function. The interface of `accode` is defined as follows:

```
int accode(n,tau,deg,Z,B,nz)  
int n; // number of state variables n  
double tau; // scaling parameter  
int deg; // degree on current call  
double Z[n][n][deg]; // partials of coefficient vectors  
double B[n][n][deg]; // result B as defined in (1)  
short nz[n][n]; // optional nonzero pattern
```

Sparsity information can be exploited by `accode` using the array `nz`. For this purpose, `nz` has to be set by a call of the routine `reverse` or the corresponding basic routines as explained below in Section 4 and Section 5, respectively. The non-positive entries of `nz` are then changed by `accode` so that upon return

$$B[i][j][k] \equiv 0 \quad \text{if } k \leq -nz[i][j].$$

# 程序代写代做 CS编程辅导

3 EASY-TO-USE DRIVERS

24

In other words,  $t^{[[k]]}$  have a sparsity pattern that fills in as  $k$  grows. Note, that there computational efficiency if a time-dependent ordinary differential equat



The prototype `<adolc/drivers.h>` is forode and accode is contained in the header file `<adolc/adolc.h>` includes this file automatically.

An example program using the procedures forode and accode together with more detailed information about the coding can be found in Section 10.4. The corresponding source code `odexam.cpp` is contained in the subdirectory `examples`.

## WeChat: cstutorcs

### 3.3 Drivers for Sparse Jacobians and Sparse Hessians

Quite often, the Jacobians and Hessians that have to be computed are sparse matrices. Therefore, ADOL-C provides additionally drivers that allow the exploitation of sparsity. The exploitation of sparsity is frequently based on *graph coloring* methods, discussed for example in [3] and [6]. The sparse drivers of ADOL-C presented in this section rely on the the coloring package ColPack developed by the authors of [3] and [6]. ColPack is not directly incorporated in ADOL-C, and therefore needs to be installed separately to use the sparse drivers described here. ColPack is available for download at <http://cscapes.cs.purdue.edu/coloringpage/software.htm>. More information about the required installation of ColPack is given in Section 9.

Assignment Project Exam Help  
Email: tutorcs@163.com  
QQ: 749389476

### Sparse Jacobians and Sparse Hessians

<https://tutorcs.com>

To compute the entries of sparse Jacobians and sparse Hessians, respectively, in coordinate format one may use the drivers:

```
int sparse_jac(tag,m,n,repeat,x,&nnz,&rind,&cind,&values,&options)
short int tag;                                // tape identification
int m;                                         // number of dependent variables m
int n;                                         // number of independent variables n
int repeat;                                    // indicate repeated call at same argument
double x[n];                                    // independent vector x
int nnz;                                       // number of nonzeros
unsigned int rind[nnz];                         // row index
unsigned int cind[nnz];                         // column index
double values[nnz];                            // non-zero values
int options[4];                                // array of control parameters
```

```
int sparse_hess(tag,n,repeat,x,&nnz,&rind,&cind,&values,&options)
```

# 程序代写代做 CS编程辅导

## 3.3 Drivers for Sparse Jacobians and Sparse Hessians

25

```
short int tag;           // tape identification
int n;                  // number of independent variables n and m = 1
int repeat;             // indicate repeated call at same argument
double x[i];            // independent vector x
int nnz;                // number of nonzeros
unsigned r_ind[nnz];    // row indices
unsigned c_ind[nnz];    // column indices
double values[nnz];    // non-zero values
int options[2];         // array of control parameters
```

WeChat: cstutorcs

Once more, the input variables are the identifier for the internal representation *tag*, if required the number of dependents *m*, and the number of independents *n* for a consistency check. Furthermore, the flag *repeat*=0 indicates that the functions are called at a point with a new sparsity structure, whereas *repeat*=1 results in the re-use of the sparsity pattern from the previous call. The current values of the independents are given by the array *x*. The input/output variable *nnz* stores the number of the nonzero entries. Therefore, *nnz* denotes also the length of the arrays *r.ind* storing the row indices, *c.ind* storing the column indices, and *values* storing the values of the nonzero entries. If *sparse.jac* and *sparse.hess* are called with *repeat*=0, the functions determine the number of nonzeros for the sparsity pattern defined by the value of *x*, allocate appropriate arrays *r.ind*, *c.ind*, and *values* and store the desired information in these arrays. During the next function call with *repeat*=1 the allocated memory is reused such that only the values of the arrays are changed. Before calling *sparse.jac* or *sparse.hess* once more with *repeat*=0 the user is responsible for the deallocation of the array *r.ind*, *c.ind*, and *values* using the function *free()*!

Email: tutorcs@163.com  
QQ: 749389476

For each driver the array *options* can be used to adapt the computation of the sparse derivative matrices to the special needs of application under consideration. Most frequently, the default options will give a reasonable performance. The elements of the array *options* control the action of *sparse.jac* according to Table 2.

The component *options*[1] determines the usage of the safe or tight mode of sparsity computation. The first, more conservative option is the default. It accounts for all dependences that might occur for any value of the independent variables. For example, the intermediate *c* = *max(a,b)* is always assumed to depend on all independent variables that *a* or *b* dependent on, i.e. the bit pattern associated with *c* is set to the logical OR of those associated with *a* and *b*. In contrast the tight option gives this result only in the unlikely event of an exact tie *a* = *b*. Otherwise it sets the bit pattern associated with *c* either to that of *a* or to that of *b*, depending on whether *c* = *a* or *c* = *b* locally. Obviously, the sparsity pattern obtained with the tight option may contain more zeros than that obtained with the safe option. On the other hand, it will only be valid at points belonging to an area where the function *F* is locally analytic and that contains the point at which the internal representation was generated. Since generating the sparsity structure using the safe version does not require any reevaluation, it may thus reduce the overall computational cost despite

# 程序代写代做 CS 编程辅导

3 EASY-TO-USE DRIVERS

coo		way of sparsity pattern computation
op		propagation of index domains (default)
		propagation of bit pattern
op		test the computational graph control flow
	0	safe mode (default)
	1	tight mode
options[2]	0	way of bit pattern propagation
	1	automatic detection (default)
	2	forward mode
		reverse mode
options[3]	0	way of compression
	1	column compression (default)
		row compression

WeChat: cstutorcs

Assignment Project Exam Help

Table 2: `sparse_jac` parameter options

Email: tutorcs@163.com

the fact that it produces more nonzero entries. The value of `options[2]` selects the direction of bit pattern propagation. Depending on the number of independent  $n$  and of dependent variables  $m$  one would prefer the forward mode if  $n$  is significant smaller than  $m$  and would otherwise use the reverse mode.

QQ: 749389476

The elements of the array `options` control the action of `sparse_hess` according to Table 3.

<https://tutorcs.com>

component	value	
<code>options[0]</code>	0	test the computational graph control flow
	1	safe mode (default)
		tight mode
<code>options[1]</code>	0	way of recovery
	1	indirect recovery (default)
		direct recovery

Table 3: `sparse_hess` parameter options

The described driver routines for the computation of sparse derivative matrices are prototyped in the header file `<adolc/sparse/sparsedrivers.h>`, which is included automatically by the global header file `<adolc/adolc.h>` (see Section 9.3). Example codes illustrating the usage of `sparse_jac` and `sparse_hess` can be found in the file `sparse_jacobian.cpp` and `sparse_hessian.cpp` contained in `examples/additional_examples/sparse`.

# 程序代写代做 CS编程辅导

## 3.3 Drivers for Sparse Jacobians and Sparse Hessians

27

### Computation of Jacobian matrices

ADOL-C offers a function `jac_pat` for determining the sparsity structure of a Jacobian matrix using the function `jac`.

```
int jac_pat(          // tape identification
    short int *t,      // number of dependent variables m
    int m,             // number of independent variables n
    int n;             // independent variables x_0
    double x[n];       // row compressed sparsity structure
    unsigned int* JP[m]; // array of control parameters
    int options[2];    // array of control parameters)
```

The sparsity pattern of the Jacobian is computed in a compressed row format. For this purpose, `JP` has to be an  $m$ -dimensional array of pointers to `unsigned int`s, i.e., one has `unsigned int* JP[m]`. During the call of `jac_pat`, the number  $\hat{n}_i$  of nonzero entries in row  $i$  of the Jacobian is determined for all  $1 \leq i \leq m$ . Then, a memory allocation is performed such that `JP[i-1]` points to a block of  $\hat{n}_i + 1$  `unsigned int`s for all  $1 \leq i \leq m$  and `JP[i-1][0]` is set to  $\hat{n}_i$ . Subsequently, the column indices of the  $j$  nonzero entries in the  $i$ -th row are stored in the components `JP[i-1][1], …, JP[i-1][j]`.

The elements of the array `options` control the action of `jac_pat` according to Table 4. The value of `options[0]` selects the way to compute the sparsity pattern. The component

component	value	
options[0]	0	way of sparsity pattern computation propagation of index domains (default)
	1	propagation of bit pattern
options[1]	0	test the computational graph control flow safe mode (default)
	1	tight mode
options[2]	0	way of bit pattern propagation automatic detection (default)
	1	forward mode
	2	reverse mode

Table 4: `jac_pat` parameter `options`

`options[1]` determines the usage of the safe or tight mode of bit pattern propagation. The first, more conservative option is the default. It accounts for all dependences that might occur for any value of the independent variables. For example, the intermediate `c = max(a,b)` is always assumed to depend on all independent variables that `a` or `b` dependent on, i.e. the bit pattern associated with `c` is set to the logical OR of those associated with `a` and `b`. In contrast the tight option gives this result only in the unlikely event of an exact tie

# 程序代写代做 CS编程辅导

3 EASY-TO-USE DRIVERS

28

`a = b.` Otherwise with the tight or the other hand,  $F$  is locally anal generated. Since any reevaluation



rn associated with  $c$  either to that of  $a$  or to that of  $b$ , depending on with the tight or the other hand,  $F$  is locally anal generated. Since any reevaluation

= b locally. Obviously, the sparsity pattern obtained are zeros than that obtained with the safe option. On at points belonging to an area where the function s the point at which the internal representation was ity structure using the safe version does not require he overall computational cost despite the fact that it produces more nonzero entries. The value of `options[2]` selects the direction of bit pattern propagation. Depending on the number of independent  $n$  and of dependent variables  $m$  one would prefer the forward mode if  $n$  is significant smaller than  $m$  and would otherwise use the reverse mode.

WeChat: cstutorcs

The routine `jac_pat` may use the propagation of bitpattern to determine the sparsity pattern. Therefore, a kind of “strip-mining” is used to cope with large matrix dimensions. If the system happens to run out of memory, one may reduce the value of the constant `PQ_STRIPMINE_MAX` following the instructions in `<adolc/sparse/sparse_rev.h>`.

The driver routine is prototyped in the header file `<adolc/sparse/sparsedrivers.h>`, which is included automatically by the global header file `<adolc/adolc.h>` (see Section 9.3). The determination of sparsity patterns is illustrated by the examples `sparse_jacobian.cpp` and `jacpatexam.cpp` contained in `examples/additional_examples/sparse`.

To compute the sparsity pattern of a Hessian in a row compressed form, ADOL-C provides the driver

QQ: 749389476

```
int hess_pat(tag, n, x, HP, options)
short int tag;           // tag identification
int n;                  // number of independent variables n
double x[n];            // independent variables x_0
unsigned int HP[];       // row compressed sparsity structure
int option;              // control parameter
```

where the user has to provide `HP` as an  $n$  dimensional array of pointers to `unsigned int`s. After the function call `HP` contains the sparsity pattern, where `HP[j][0]` contains the number of nonzero elements in the  $j$ th row for  $1 \leq j \leq n$ . The components `P[j][i]`,  $0 < i \leq P[j][0]$  store the indices of these entries. For determining the sparsity pattern, ADOL-C uses the algorithm described in [15]. The parameter `option` determines the usage of the safe (`option = 0`, default) or tight mode (`option = 1`) of the computation of the sparsity pattern as described above.

This driver routine is prototyped in the header file `<adolc/sparse/sparsedrivers.h>`, which is included automatically by the global header file `<adolc/adolc.h>` (see Section 9.3). An example employing the procedure `hess_pat` can be found in the file `sparse_hessian.cpp` contained in `examples/additional_examples/sparse`.

# 程序代写代做 CS编程辅导

## 3.3 Drivers for Sparse Jacobians and Sparse Hessians

29

Calculation of



To compute a column of the derivative matrix from a given sparsity pattern, one has to calculate an appropriate seed matrix. This seed matrix can be used as input for the derivative calculation. To facilitate the recovery of the derivative matrices for a sparsity pattern given in row compressed form, ADOL-C provides two drivers, which are based on the ColPack library:

```
int generate_seed_jac(m, n, JP, S, p)
int m;                                // number of dependent variables  $m$ 
int n;                                // number of independent variables  $n$ 
unsigned int JP[];                      // row compressed sparsity structure of Jacobian
double S[n][p];                        // seed matrix
int p;                                // number of columns in  $S$ 
```

WeChat: cstidores

The input variables to `generate_seed_jac` are the number of dependent variables  $m$ , the number of independent variables  $n$  and the sparsity pattern  $JP$  of the Jacobian computed for example by `jac_pat`. First, `generate_seed_jac` performs a distance-2 coloring of the bipartite graph defined by the sparsity pattern  $JP$  as described in [3]. The number of colors needed for the coloring determines the number of columns  $p$  in the seed matrix. Subsequently, `generate_seed_jac` allocates the memory needed by  $S$  and initializes  $S$  according to the graph coloring. The coloring algorithm that is applied in `generate_seed_jac` is used also by the driver `sparse_jac` described earlier.

Email: tutorcs@163.com  
QQ: 749389476

```
int generate_seed_hess(n, HP, S, p)
int n;                                // number of independent variables  $n$ 
unsigned int HP[];                      // row compressed sparsity structure of Jacobian
double S[n][p];                        // seed matrix
int p;                                // number of columns in  $S$ 
```

The input variables to `generate_seed_hess` are the number of independents  $n$  and the sparsity pattern  $HP$  of the Hessian computed for example by `hess_pat`. First, `generate_seed_hess` performs an appropriate coloring of the adjacency graph defined by the sparsity pattern  $HP$ : An acyclic coloring in the case of an indirect recovery of the Hessian from its compressed representation and a star coloring in the case of a direct recovery. Subsequently, `generate_seed_hess` allocates the memory needed by  $S$  and initializes  $S$  according to the graph coloring. The coloring algorithm applied in `generate_seed_hess` is used also by the driver `sparse_hess` described earlier.

The specific set of criteria used to define a seed matrix  $S$  depends on whether the sparse derivative matrix to be computed is a Jacobian (nonsymmetric) or a Hessian (symmetric). It also depends on whether the entries of the derivative matrix are to be recovered from the compressed representation *directly* (without requiring any further arithmetic) or *indirectly* (for example, by solving for unknowns via successive substitutions). Appropriate recovery

# 程序代写代做 CS编程辅导

30

3 EASY-TO-USE DRIVERS

routines are prov  
in the previous s  
the exploitation



used in the drivers `sparse_jac` and `sparse_hess` described with a detailed analysis of the employed drivers for and in the papers [4] and [5].

These driver routines are included automatically in `<adolc/sparse/sparsedrivers.h>`, which is included in the header file `<adolc/adolc.h>` (see Section 9.3). An example code illustrating the use of `generate_seed_jac` and `generate_seed_hess` can be found in the file `sparse_jac.c` contained in `examples/additional_examples/sparse`.

### 3.4 Higher Derivative Tensors

WeChat: cstutorcs

Many applications in scientific computing need second- and higher-order derivatives. Often, one does not require full derivative tensors but only the derivatives in certain directions  $s_i \in \mathbb{R}^n$ . Suppose a collection of  $p$  directions  $s \in \mathbb{R}^{n \times p}$  is given, which form a matrix

Assignment Project Exam Help

$$S = [s_1, s_2, \dots, s_p] \in \mathbb{R}^{n \times p}.$$

One possible choice is  $S = I_n$  with  $p = n$ , which leads to full tensors being evaluated. ADOL-C provides the function `tensor_eval` to calculate the derivative tensors

$$\nabla_S^k F(x_0) = \left. \frac{\partial^k}{\partial z^k} F(x_0 + Sz) \right|_{z=0} \in \mathbb{R}^{p^k} \quad \text{for } k = 0, \dots, d \quad (2)$$

simultaneously. The function `tensor_eval` has the following calling sequence and parameters:

```
void tensor_eval(tag,m,n,d,p,x,tensor,S)
short int tag;           // type identification
int m;                  // number of dependent variables m
int n;                  // number of independent variables n
int d;                  // highest derivative degree d
int p;                  // number of directions p
double x[n];            // values of independent variables x_0
double tensor[m][size]; // result as defined in (2) in compressed form
double S[n][p];          // seed matrix S
```

<https://tutorcs.com>

Using the symmetry of the tensors defined by (2), the memory requirement can be reduced enormously. The collection of tensors up to order  $d$  comprises  $\binom{p+d}{d}$  distinct elements. Hence, the second dimension of `tensor` must be greater or equal to  $\binom{p+d}{d}$ . To compute the derivatives, `tensor_eval` propagates internally univariate Taylor series along  $\binom{n+d-1}{d}$  directions. Then the desired values are interpolated. This approach is described in [9].

The access of individual entries in symmetric tensors of higher order is a little tricky. We always store the derivative values in the two dimensional array `tensor` and provide two different ways of accessing them. The leading dimension of the tensor array ranges over the

# 程序代写代做 CS编程辅导

## 3.4 Higher Derivative Tensors

31

component index .e.,  $F_{i+1}$  for  $i = 0, \dots, m - 1$ . The sub-arrays pointed to by `tensor[i]` have size  $\binom{p+d}{d}$  for all  $i$ . Each of them represents the symmetric function  $F_{i+1}$  in  $p$  variables. The  $\binom{p+d}{d}$  mixed partial derivatives are linearly ordered according to the tetrahedral scheme described by Knuth [1973]. In the familiar quadratic case  $d = 2$  the derivative with respect to  $z_j$  and  $z_k$  with  $j < k$  is stored at `tensor[i][l]` with  $l = k * (k + 1)/2 + j$ . At  $j = 0 = k$  we find the function value  $F_{i+1}$  itself and the gradient  $\nabla F_{i+1} = \partial F_{i+1}/\partial x_k$  is stored at  $l = k(k + 1)/2$  with  $j = 0$  for  $k = 1, \dots, p$ .

For general  $d$  we combine the variable indices to a multi-index  $j = (j_1, j_2, \dots, j_d)$ , where  $j_k$  indicates differentiation with respect to variable  $x_{j_k}$  with  $j_k \in \{0, 1, \dots, p\}$ . The value  $j_k = 0$  indicates no differentiation so that all lower derivatives are also contained in the same data structure as described above for the quadratic case. The location of the partial derivative specified by  $j$  is computed by the function

## Assignment Project Exam Help

```
int tensor_address(d,j)
int d;           // highest derivative degree d
int j[d];       // multi-index j
```

Email: tutorcs@163.com

and it may thus be referenced as `tensor[i][tensor_address(d, j)]`. Notice that the address computation does depend on the degree  $d$  but not on the number of directions  $p$ , which could theoretically be enlarged without the need to reallocate the original tensor. Also, the components of  $j$  need to be non-increasing. To some C programmers it may appear more natural to access tensor entries by successive dereferencing in the form `tensorentry[i][j1][j2]...[jd]`. We have also provided this mode, albeit with the restriction that the indices  $j_1, j_2, \dots, j_d$  are non-increasing. In the second order case this means that the Hessian entries must be specified in or below the diagonal. If this restriction is violated the values are almost certain to be wrong and array bounds may be violated. We emphasize that subscripting is not overloaded but that `tensorentry` is a conventional and thus moderately efficient C pointer structure. Such a pointer structure can be allocated and set up completely by the function

```
void** tensorsetup(m,p,d,tensor)
int m;           // number of dependent variables n
int p;           // number of directions p
int d;           // highest derivative degree d
double tensor[m][size]; // pointer to two dimensional array
```

Here, `tensor` is the array of  $m$  pointers pointing to arrays of size  $\geq \binom{p+d}{d}$  allocated by the user before. During the execution of `tensorsetup`,  $d - 1$  layers of pointers are set up so that the return value allows the direct dereferencing of individual tensor elements.

For example, suppose some active section involving  $m \geq 5$  dependents and  $n \geq 2$  independents has been executed and taped. We may select  $p = 2$ ,  $d = 3$  and initialize the

# 程序代写代做 CS编程辅导

3 EASY-TO-USE DRIVERS

$n \times 2$  seed matrix  has  $s_1$  and  $s_2$ . Then we are able to execute the code segment

```
double*** tensorentry(int m, int p, int d, double*** tensor);
tensor_eval(tensor, tensorentry, s1, s2);
```

This way, we evaluate the mixed partial derivative defined in (2) up to degree 3 in both directions  $s_1$  and  $s_2$  at some argument  $x$ . To allow the access of tensor entries by dereferencing the pointer structure `tensorentry` has been created. Now, the value of the mixed partial

WeChat: cstutorcs

$$\left. \frac{\partial^3 F_3(x + s_1 z_1 + s_2 z_2)}{\partial z_1^2 \partial z_2} \right|_{z_1=0=z_2}$$

can be recovered as

## Assignment Project Exam Help

`tensorentry[4][2][1][1]`      or      `tensor[4][tensor_address(d, j)]`,

Email: tutorcs@163.com

where the integer array `j` may equal  $(1,1,2)$ ,  $(1,2,1)$  or  $(2,1,1)$ . Analogously, the entry

`tensorentry[2][1][0][0]`      or      `tensor[2][tensor_address(d, j)]`

QQ: 749389476

with  $j = (1,0,0)$  contains the first derivative of the third dependent variable  $F_3$  with respect to the first differentiation parameter  $z_1$ .

Note, that the pointer structure `tensorentry` has to be set up only once. Changing the values of the array `tensor`, e.g. by a further call of `tensor_eval`, directly effects the values accessed by `tensorentry`. When no more derivative evaluations are desired the pointer structure `tensorentry` can be deallocated by a call to the function

```
int freetensor(int m, int p, int d, double *** tensorentry)
int m; // number of dependent variables m
int p; // number of independent variables p
int d; // highest derivative degree d
double*** tensorentry[m]; // return value of tensorsetup
```

that does not deallocate the array `tensor`.

The drivers provided for efficient calculation of higher order derivatives are prototyped in the header file `<adolc/drivers/taylor.h>`, which is included by the global header file `<adolc/adolc.h>` automatically (see Section 9.3). Example codes using the above procedures can be found in the files `taylorexam.C` and `accessexam.C` contained in the subdirectory `examples/additional_examples/taylor`.

# 程序代写代做 CS 编程辅导

## 3.5 Derivatives and Inverse Functions



Frequently, one needs to solve systems of equations for variables  $y \in \mathbb{R}^m$  that are implicitly defined as functions of some variables 

$\mathbb{R}^m$  with  $z = (y, x) \in \mathbb{R}^n$ .

Naturally, the  $n - m$  variables may not be partitioned in this regular fashion and we wish to provide flexibility for a convenient selection of the  $n - m$  truly independent variables. Let  $P \in \mathbb{R}^{(n-m) \times n}$  be a 0–1 matrix that picks out these variables so that it is a column permutation of the matrix  $\begin{bmatrix} I_m & 0_{m \times (n-m)} \end{bmatrix} \in \mathbb{R}^{(n-m) \times n}$ . Then the nonlinear system

WeChat: cstutorcs

$$G(z) = 0, \quad Pz = x,$$

has a regular Jacobian, wherever the implicit function theorem yields  $y$  as a function of  $x$ . Hence, we may also write

$$F(z) = \begin{pmatrix} G(z) \\ Pz \end{pmatrix} \equiv \begin{pmatrix} 0 \\ Pz \end{pmatrix} \equiv Sx, \quad (3)$$

where  $S = [0, I_p]^T \in \mathbb{R}^{n \times p}$  with  $p = n - m$ . Now, we have rewritten the original implicit functional relation between  $x$  and  $y$  as an inverse relation  $F(z) = Sx$ . In practice, we may implement the projection  $P$  simply by marking  $n - m$  of the independents also dependent.

Given any  $F : \mathbb{R}^n \mapsto \mathbb{R}^n$  that is locally invertible and an arbitrary seed matrix  $S \in \mathbb{R}^{n \times p}$  we may evaluate all derivatives of  $z \in \mathbb{R}^n$  with respect to  $x \in \mathbb{R}^p$  by calling the following routine:

<https://tutorcs.com>

```

void inverse_tensor_eval(tag,n,d,p,z,tensor,S)
short int tag;           // tape identification
int n;                  // number of variables  $n$ 
int d;                  // highest derivative degree  $d$ 
int p;                  // number of directions  $p$ 
double z[n];            // values of independent variables  $z$ 
double tensor[n][size]; // partials of  $z$  with respect to  $x$ 
double S[n][p];          // seed matrix  $S$ 

```

The results obtained in `tensor` are exactly the same as if we had called `tensor_eval` with `tag` pointing to a tape for the evaluation of the inverse function  $z = F^{-1}(y)$  for which naturally  $n = m$ . Note that the columns of  $S$  belong to the domain of that function. Individual derivative components can be accessed in `tensor` exactly as in the explicit case described above.

It must be understood that `inverse_tensor_eval` actually computes the derivatives of  $z$  with respect to  $x$  that is defined by the equation  $F(z) = F(z_0) + Sx$ . In other words the

base point at which no capability for the first place is



is differentiated is given by  $F(z_0)$ . The routine has solving systems of nonlinear equations  $F(z) = 0$  in task. However, the routine `jac_solv` described in Section 3.1 may

As an example

solving two nonlinear expressions



$$\begin{aligned} G_1(z_1, z_2, z_3, z_4) &= z_1^2 + z_2^2 - z_3^2 \\ G_2(z_1, z_2, z_3, z_4) &= \cos(z_4) - z_1/z_3 . \end{aligned}$$

The equations  $G(z) = 0$  describe the relation between the Cartesian coordinates  $(z_1, z_2)$  and the polar coordinates  $(z_3, z_4)$  in the plane. Now, suppose we are interested in the derivatives of the second Cartesian  $y_1 = z_2$  and the second (angular) polar coordinate  $y_2 = z_4$  with respect to the other two variables  $x_1 = z_1$  and  $x_2 = z_3$ . Then the active section could look simply like

## Assignment Project Exam Help

```

for (j=1; j < 5; j++)    z[j] <= zp[j];
g[1] = z[1]*z[1]+z[2]*z[2]-z[3]*z[3];
g[2] = cos(z[4]) - z[1]/z[3];
g[1] >>= gp[1];      g[2] >>= gp[2];
z[1] >>= zd[1];      z[3] >>= zd[2];

```

**QQ: 749389476**

where  $zd[1]$  and  $zd[2]$  are dummy arguments. In the last line the two independent variables  $z[1]$  and  $z[3]$  are made simultaneously dependent thus generating a square system that can be inverted (at most arguments). The corresponding projection and seed matrix are

$$P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \quad \text{and} \quad S^T = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} .$$

Provided the vector `zp` is consistent in that its Cartesian and polar components describe the same point in the plane the resulting tuple `gp` must vanish. The call to `inverse_tensor_eval` with  $n = 4$ ,  $p = 2$  and  $d$  as desired will yield the implicit derivatives, provided `tensor` has been allocated appropriately of course and  $S$  has the value given above. The example is untypical in that the implicit function could also be obtained explicitly by symbolic manipulations. It is typical in that the subset of  $z$  components that are to be considered as truly independent can be selected and altered with next to no effort at all.

The presented drivers are prototyped in the header file `<adolc/drivers/taylor.h>`. As indicated before this header is included by the global header file `<adolc/adolc.h>` automatically (see Section 9.3). The example programs `inversexam.cpp`, `coordinates.cpp` and `trigger.cpp` in the directory `examples/additional_examples/taylor` show the application of the procedures described here.

# 程序代写代做CS编程辅导

## 3.6 Drivers for Lie Derivatives

Nonlinear control systems often require certain types of Lie derivatives. These derivatives also appear in applications such as classical mechanics and relativity. Lie derivatives are total derivatives along a vector field. The drivers for calculating Lie derivatives in Algoic were developed by Klaus Röbenack [13] and his co-worker Wang and Mirko Franke [14]. They are prototyped as C functions in the header file `<algoic_lie.h>` and allow the calculation of Lie derivatives of scalar, vector and covector fields. In addition, the calculation of gradients of Lie derivatives of scalar fields is supported.

## WeChat: cstutorcs

We consider computational problems occurring in controller and observer design for nonlinear control systems

## Assignment Project Exam Help

with the scalar input  $u$ , the state  $\mathbf{x}$ , and the output  $\mathbf{y}$ , where the vector fields  $\mathbf{f} : \Omega \rightarrow \mathbb{R}^n$ ,  $\mathbf{g} : \Omega \rightarrow \mathbb{R}^n$  and the map  $\mathbf{h} : \Omega \rightarrow \mathbb{R}^m$  (for  $m = 1$  this map is the scalar field  $h$ ) are defined on an open subset  $\Omega \subseteq \mathbb{R}^n$ .

Email: tutorcs@163.com

### Lie Derivatives of Scalar Fields

Consider the initial value problem

QQ: 749389476

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}), \quad \mathbf{y} = \mathbf{h}(\mathbf{x}), \quad \mathbf{x}(0) = \mathbf{x}_0 \in \Omega$$

with the vector field  $\mathbf{f} : \Omega \rightarrow \mathbb{R}^n$ . The Lie derivative of the scalar field  $h : \Omega \rightarrow \mathbb{R}$  along the vector field  $\mathbf{f}$  are time derivatives of the curve  $y$ , i.e.,

$$\dot{y}(0) = L_{\mathbf{f}} h(\mathbf{x}_0), \quad \ddot{y}(0) = L_{\mathbf{f}}^2 h(\mathbf{x}_0), \quad \dots, \quad y^{(d)}(0) = L_{\mathbf{f}}^d h(\mathbf{x}_0).$$

Formally, the Lie derivative of the scalar field  $h$  along the vector field  $\mathbf{f}$  is defined by

$$L_{\mathbf{f}} h(\mathbf{x}) = \frac{\partial h(\mathbf{x})}{\partial \mathbf{x}} \mathbf{f}(\mathbf{x}).$$

Higher order Lie derivatives are given by the recursion

$$L_{\mathbf{f}}^{k+1} h(\mathbf{x}) = \frac{\partial L_{\mathbf{f}}^k h(\mathbf{x})}{\partial \mathbf{x}} \mathbf{f}(\mathbf{x}) \quad \text{with} \quad L_{\mathbf{f}}^0 h(\mathbf{x}) = h(\mathbf{x}).$$

To compute Lie derivatives

$$(L_{\mathbf{f}}^0 h(\mathbf{x}_0), \dots, L_{\mathbf{f}}^d h(\mathbf{x}_0))$$

we need the trace numbers of the active sections of the vector field  $\mathbf{f}$  and the scalar field  $h$ , the number of independent variables  $n$ , the initial value  $\mathbf{x}_0 \in \Omega$  and the highest derivative degree  $d$ . The values of the Lie derivatives are then stored in the one dimensional array `result` of length  $d + 1$ :

# 程序代写代做 CS 编程辅导

3 EASY-TO-USE DRIVERS

```

int lie_sca
short Tap // trace identification of vector field f
short Tap // trace identification of scalar field h
short n; // number of independent variables n and m = 1
double x0 // values of independent variables x0
short d; // highest derivative degree d
double re // resulting Lie derivatives of a scalar field

```

For a smooth vector-valued map  $\mathbf{h} : \Omega \rightarrow \mathbb{R}^m$  with  $\mathbf{h}(\mathbf{x}) = (h_1(\mathbf{x}), \dots, h_m(\mathbf{x}))^T$  we define the Lie derivative

$$L_{\mathbf{f}}^k \mathbf{h}(\mathbf{x}_0) = (L_{\mathbf{f}}^k h_1(\mathbf{x}_0), \dots, L_{\mathbf{f}}^k h_m(\mathbf{x}_0))^T \in \mathbb{R}^m \quad \text{for } k = 0, \dots, d, \quad (4)$$

component-wise by the Lie derivatives  $L_{\mathbf{f}}^k h_1(\mathbf{x}_0), \dots, L_{\mathbf{f}}^k h_m(\mathbf{x}_0)$  of the  $m$  scalar fields  $h_1, \dots, h_m : \Omega \rightarrow \mathbb{R}$ . Note that (4) should not be confused with the Lie derivative of a vector field. To compute Lie derivatives (4) of the vector-valued map  $\mathbf{h}$  we additionally need the number of dependent variables  $m$ :

**Email: tutorcs@163.com**

```

int lie_scalarcv(Tape_F, Tape_H, n, m, x0, d, result)
short Tape_F; // trace identification of vector field f
short Tape_H; // trace identification of vector-valued map h
short n; // number of independent variables n
short m; // number of dependent variables m
double x0[n]; // values of independent variables x0
short d; // highest derivative degree d
double result[m][d+1]; // resulting Lie derivatives of vectorial scalar fields

```

**QQ: 749389476**

**https://tutorcs.com**

## Gradients of Lie Derivatives of Scalar Fields

To compute the gradients

$$(dL_{\mathbf{f}}^0 h(\mathbf{x}_0), \dots, dL_{\mathbf{f}}^d h(\mathbf{x}_0))$$

of the Lie derivatives  $L_{\mathbf{f}}^0 h(\mathbf{x}_0), \dots, L_{\mathbf{f}}^d h(\mathbf{x}_0)$  of the scalar field  $h : \Omega \rightarrow \mathbb{R}$  along the vector field  $\mathbf{f} : \Omega \rightarrow \mathbb{R}^n$ , the following C function can be used:

```

int lie_gradientc(Tape_F, Tape_H, n, x0, d, result)
short Tape_F; // trace identification of vector field f
short Tape_H; // trace identification of scalar field h
short n; // number of independent variables n and m = 1
double x0[n]; // values of independent variables x0

```

# 程序代写代做CS编程辅导

## 3.6 Drivers of the Toolbox for Lie Derivatives

37

```
short d; // highest derivative degree d
double result[m][n][d+1]; // resulting gradients of Lie derivatives
                           // of a scalar field
```

For calculating the  $d+1$  Lie derivatives  $L_{\mathbf{f}}^0 \omega(\mathbf{x}_0), \dots, L_{\mathbf{f}}^d \omega(\mathbf{x}_0) \in \mathbb{R}^{m \times n}$  of Lie derivatives (4) of the vector-valued map  $\mathbf{h} : \Omega \rightarrow \mathbb{R}^{m \times n}$  along the vector field  $\mathbf{f}$ , one can use the following function, where the  $d+1$  matrices are stored in the three dimensional array `result`:

```
int lie_gradientcv(Tape_F, Tape_H, n, m, x0, d, result)
short Tape_F; // trace identification of vector field f
short Tape_H; // trace identification of vector-valued map h
short n; // number of independent variables n
short m; // number of dependent variables m
double x0[n]; // values of independent variables x_0
short d; // highest derivative degree d
double result[m][n][d+1]; // resulting jacobians of Lie derivatives
                           // of vectorial scalar fields
```

Email: [tutorcs@163.com](mailto:tutorcs@163.com)

Lie Derivatives of Covector Fields

QQ: 749389476

If we consider the elements of the real vector space  $\mathbb{R}^n$  as column vectors, the elements of the associated dual space  $(\mathbb{R}^n)^*$  can be represented by row vectors. These row vectors are called adjoint vectors in ADOL-C and covectors in differential geometry. In a program the user does not have to distinguish between a vector field  $\mathbf{f} : \Omega \rightarrow \mathbb{R}^n$  and a covector field  $\omega : \Omega \rightarrow (\mathbb{R}^n)^*$ , since for both fields the dependent variables are stored in one dimensional arrays. The Lie derivative of a covector field  $\omega$  is defined by

$$L_{\mathbf{f}} \omega(\mathbf{x}) = \omega(\mathbf{x}) \mathbf{f}'(\mathbf{x}) + \mathbf{f}^T(\mathbf{x}) \left( \frac{\partial \omega^T(\mathbf{x})}{\partial \mathbf{x}} \right)^T.$$

The  $d+1$  Lie derivatives  $L_{\mathbf{f}}^0 \omega(\mathbf{x}), \dots, L_{\mathbf{f}}^d \omega(\mathbf{x})$  of the covector field  $\omega$  along the vector field  $\mathbf{f}$  can be computed by

```
int lie_covector(Tape_F, Tape_W, n, x0, d, result)
short Tape_F; // trace identification of vector field f
short Tape_W; // trace identification of covector field omega
short n; // number of independent variables n
double x0[n]; // values of independent variables x_0
short d; // highest derivative degree d
double result[n][d+1]; // resulting Lie derivatives of a covector field
```

# 程序代写代做 CS编程辅导

3 EASY-TO-USE DRIVERS

Lie Derivatives



(Lie Brackets)

Lie derivatives ( $[f, g]$ ) of a scalar field  $h : \Omega \rightarrow \mathbb{R}$  along a vector field  $\mathbf{g} : \Omega \rightarrow \mathbb{R}^n$  along a vector field  $\mathbf{f} : \Omega \rightarrow \mathbb{R}^n$  are defined by

$$[f, g](x) = g'(x)f(x) - f'(x)g(x).$$

We can calculate  $\text{ad}_{\mathbf{f}}^0 g(x), \text{ad}_{\mathbf{f}}^1 g(x), \dots, \text{ad}_{\mathbf{f}}^d g(x)$  at the point  $x_0 \in \Omega$  using the C function listed below:

```
int lie_bracket(Tape_F, Tape_G, n, x0, d, result)
short Tape_F,                                // trace identification of vector field f
short Tape_G;                                // trace identification of vector field g
short n;                                     // number of independent variables n
double x0[n];                                // values of independent variables x
short d;                                     // highest derivative degree d
double result[n][d+1];                        // resulting Lie derivatives of a vector field
```

**WeChat: cstutorcs  
Assignment Project Exam Help  
Email: tutorcs@163.com**

Additional C++ Drivers for Lie Derivatives

For Lie derivatives of scalar fields, the C interface offers two different drivers, namely `lie_scalarc` for  $m = 1$  scalar field  $h : \Omega \rightarrow \mathbb{R}$  and `lie_scalarcv` for  $m \geq 1$  scalar fields  $h_1, \dots, h_m : \Omega \rightarrow \mathbb{R}$ , which are combined in a vector-valued map  $\mathbf{h} : \Omega \rightarrow \mathbb{R}^m$ . Using the polymorphism in C++, both C drivers are unified by the C++ driver `lie_scalar` for the computation of Lie derivatives of scalar fields.

```
int lie_scalar(Tape_F, Tape_H, n, x0, d, result)      // case m = 1
int lie_scalar(Tape_F, Tape_H, n, m, x0, d, result)    // case m ≥ 1
```

The same situation occurs with the gradients or jacobians of Lie derivatives of scalar fields. Here, the C drivers `lie_gradientc` and `lie_gradientcv` are unified by the C++ driver `lie_gradient`:

```
int lie_gradient(Tape_F, Tape_H, n, x0, d, result)      // case m = 1
int lie_gradient(Tape_F, Tape_H, n, m, x0, d, result)    // case m ≥ 1
```

The C++ drivers are also prototyped in the header file `<adolc_lie.h>`. An example how to use these drivers for the calculation of Lie derivatives can be found in the file `GantryCrane.cpp` in the directory `additional_examples/lie`.

## 4 Basic D

### Forward and Reverse Mode

In this section, we will discuss forward and reverse mode automatic differentiation. For better understanding, we start with a short description of the mathematical concepts.

Provided no numerical underflow or overflow occurs, no comparison including `fmax` or `fmin` yields a tie, `fabs` does not yield zero, and all special functions were evaluated in the interior of their domains, the functional relation between the input variables  $x$  and the output variables  $y$  denoted by  $y = F(x)$  is in fact analytic. In other words, we can compute arbitrarily high derivatives of the vector function  $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$  defined by the active section. We find it most convenient to describe and compute derivatives in terms of univariate Taylor expansions, which are truncated after the highest derivative degree  $d$  that is desired by the user. Let

$$x(t) = \sum_{j=0}^d x_j t^j : \mathbb{R} \rightarrow \mathbb{R}^n \quad (5)$$

denote any vector polynomial in the scalar variable  $t \in \mathbb{R}$ . In other words,  $x(t)$  describes a path in  $\mathbb{R}^n$  parameterized by  $t$ . The Taylor coefficient vectors

$$x_j = \frac{1}{j!} \left. \frac{\partial^j}{\partial t^j} x(t) \right|_{t=0}$$

QQ: 749389476

are simply the scaled derivatives of  $x(t)$  at the parameter origin  $t = 0$ . The first two vectors  $x_1, x_2 \in \mathbb{R}^n$  can be visualized as tangent and curvature at the base point  $x_0$ , respectively. Provided that  $F$  is  $d$  times continuously differentiable, it follows from the chain rule that the image path

$$y(t) \equiv F(x(t)) : \mathbb{R} \rightarrow \mathbb{R}^m \quad (6)$$

is also smooth and has  $(d + 1)$  Taylor coefficient vectors  $y_j \in \mathbb{R}^m$  at  $t = 0$ , so that

$$y(t) = \sum_{j=0}^d y_j t^j + O(t^{d+1}). \quad (7)$$

Also as a consequence of the chain rule, one can observe that each  $y_j$  is uniquely and smoothly determined by the coefficient vectors  $x_i$  with  $i \leq j$ . In particular we have

$$\begin{aligned} y_0 &= F(x_0) \\ y_1 &= F'(x_0)x_1 \\ y_2 &= F'(x_0)x_2 + \frac{1}{2}F''(x_0)x_1x_1 \\ y_3 &= F'(x_0)x_3 + F''(x_0)x_1x_2 + \frac{1}{6}F'''(x_0)x_1x_1x_1 \\ &\dots \end{aligned} \quad (8)$$

In writing down notation. It is w  
sions for the  $y_j$  i  
 $x_i$  with  $i \leq j$  g  
occur in automata  
so that storage a



have already departed from the usual matrix-vector number of terms that occur in these “symbolic” expressions—derivative tensors of  $F$  and the “input” coefficients  $\mathbf{h}_j$ . Fortunately, this exponential growth does not occur if the many terms are somehow implicitly combined and grow only quadratically in the bound  $d$  on  $j$ .

Provided  $F$  is  <img alt="black square" data-bbox

$$y_j = y_j(x_0, x_1, \dots, x_j) \in I\!\!R^m,$$

and their derivatives satisfy the identities

# WeChat: cstutorcs

$$\frac{\partial y_j}{\partial x_i} = \frac{\partial y_{j-i}}{\partial x_0} = A_{j-i}(x_0, x_1, \dots, x_{j-i}) \quad (9)$$

as established in [2]. This yields in particular

# [2]. This yields in particular

# Assignment Project Exam Help

$$\frac{\partial y_0}{\partial x_0} = \frac{\partial y_1}{\partial x_1} = \frac{\partial y_2}{\partial x_2} = \frac{\partial y_3}{\partial x_3} = A_0 = F'(x_0)$$

$$\frac{\partial y_1}{\partial x_0} = \frac{\partial y_2}{\partial x_1} = \frac{\partial y_3}{\partial x_2} = A_1 = F''(x_0)x_1$$

$$\frac{\partial y_2}{\partial x_0} = \frac{\partial y_3}{\partial x_1} = A_2 = F''(x_0)x_2 + \frac{1}{2}F'''(x_0)x_1x_1$$

$$\text{QQ: } \frac{\partial y_3}{\partial x_0} = A_3 + F'(x_0)x_3 + F'''(x_0)x_1x_2 + \frac{1}{6}F^{(4)}(x_0)x_1x_1x_1$$

•

The  $m \times n$  matrices  $A_k$ ,  $k = 0, \dots, d$ , are actually the Taylor coefficients of the Jacobian path  $F'(x(t))$ , a fact that is of interest primarily in the context of ordinary differential equations and differential algebraic equations.

Given the tape of an active section and the coefficients  $x_j$ , the resulting  $y_j$  and their derivatives  $A_j$  can be evaluated by appropriate calls to the ADOL-C forward mode implementations and the ADOL-C reverse mode implementations. The scalar versions of the forward mode propagate just one truncated Taylor series from the  $(x_j)_{j \leq d}$  to the  $(y_j)_{j \leq d}$ . The vector versions of the forward mode propagate families of  $p \geq 1$  such truncated Taylor series in order to reduce the relative cost of the overhead incurred in the tape interpretation. In detail, ADOL-C provides

```

int zos_forward(tag,m,n,keep,x,y)
short int tag;           // tape identification
int m;                  // number of dependent variables  $m$ 
int n;                  // number of independent variables  $n$ 
int keep;               // flag for reverse mode preparation
double x[n];            // independent vector  $x = x_0$ 
double y[m];            // dependent vector  $y = F(x_0)$ 

```

# 程序代写代做 CS编程辅导

41

for the **zero-order** driver. This driver computes  $y = F(x)$  with  $0 \leq \text{keep} \leq 1$ . The integer flag **keep** writes the first  $\text{keep}$  intermediate quantities as in the call to **trace\_on**: It determines if **zos\_forward** writes the first  $\text{keep}$  all intermediate quantities into a buffered temporary file, i.e., the value **keep** = 1 provides a buffer for a subsequent reverse mode evaluation. The value **keep** = 0 provides a buffer for **forward** or **fov\_reverse** as explained below.

To compute  $f$  one has

```
int fos_forward(tag,m,n,keep,x0,x1,y0,y1)
short int tag; // tape identification
int m; // number of dependent variables  $m$ 
int n; // number of independent variables  $n$ 
int keep; // flag for reverse mode preparation
double x0[n]; // independent vector  $x_0$ 
double x1[n]; // tangent vector  $x_1$ 
double y0[m]; // dependent vector  $y_0 = F(x_0)$ 
double y1[m]; // first derivative  $y_1 = F'(x_0)x_1$ 
```

WeChat: cstutorcs  
Assignment Project Exam

for the first-order scalar forward mode. Here one has  $0 \leq \text{keep} \leq 2$ , where

`keep = { 1 prepares for fos_reverse or fov_reverse  
        2 prepares for hos_reverse or hov_reverse }`

as explained below. For the first-order vector forward mode, ADOL-C provides

```
int fov_forward(tag,m,n,p,x0,X,y0,Y)
short int tag;           // tape identification
int m;                  // number of dependent variables  $m$ 
int n;                  // number of independent variables  $n$ 
int p;                  // number of directions
double x0[n];           // independent vector  $x_0$ 
double X[n][p];          // tangent matrix  $X$ 
double y0[m];           // dependent vector  $y_0 = F(x_0)$ 
double Y[m][p];          // first derivative matrix  $Y = F'(x)X$ 
```

For the computation of higher derivative, the driver

```

int hos_forward(tag,m,n,d,keep,x0,X,y0,Y)
short int tag;           // tape identification
int m;                  // number of dependent variables  $m$ 
int n;                  // number of independent variables  $n$ 
int d;                  // highest derivative degree  $d$ 
int keep;               // flag for reverse mode preparation
double x0[n];           // independent vector  $x_0$ 

```

# 程序代写代做CS编程辅导

42

4 BASIC DRIVERS FOR THE FORWARD AND REVERSE MODE

```
double X[  
double y0  
double Y[  
    ] // tangent matrix X  
    ] // dependent vector  $y_0 = F(x_0)$   
    ] // derivative matrix Y
```

implementing the forward mode. The rows of the matrix  $X$  must correspond to the independent variables in the order of their initialization by the  $\ll=$  operator. The columns of  $X = \{x_j\}_{j=1\dots d}$  represent Taylor coefficient vectors as in (5). The rows of the matrix  $Y$  must correspond to the dependent variables in the order of their selection by the  $\gg=$  operator. The columns of  $Y = \{y_j\}_{j=1\dots d}$  represent Taylor coefficient vectors as in (7), i.e., `hos_forward` computes the values  $y_0 = F(x_0)$ ,  $y_1 = F'(x_0)x_1$ ,  $\dots$ , where  $X = [x_1, x_2, \dots, x_d]$  and  $Y = [y_1, y_2, \dots, y_d]$ . Furthermore, one has  $0 \leq \text{keep} \leq d + 1$ , with

$\text{keep} \begin{cases} = 1 & \text{prepares for fos\_reverse or fov\_reverse} \\ > 1 & \text{prepares for hos\_reverse or hor\_reverse} \end{cases}$

Once more, there is also a vector version given by

```
int hov_forward(tag,m,n,d,p,x0,X,y0,Y)  
short int tag; // tape identification  
int m; // number of dependent variables m  
int n; // number of independent variables n  
int d; // highest derivative degree d  
int p; // number of directions p  
double x0[n]; // independent vector x0  
double X[n][p][d]; // tangent matrix X  
double y0[n]; // dependent vector  $y_0 = F(x_0)$   
double Y[m][p][d]; // derivative matrix Y
```

for the higher-order vector forward mode that computes  $y_0 = F(x_0)$ ,  $Y_1 = F'(x_0)X_1$ ,  $\dots$ , where  $X = [X_1, X_2, \dots, X_d]$  and  $Y = [Y_1, Y_2, \dots, Y_d]$ .

There are also overloaded versions providing a general forward-call. Details of the appropriate calling sequences are given in Section 5.

Once, the required information is generated due to a forward mode evaluation with an appropriate value of the parameter `keep`, one may use the following implementation variants of the reverse mode. To compute first-order derivatives one can use

```
int fos_reverse(tag,m,n,u,z)  
short int tag; // tape identification  
int m; // number of dependent variables m  
int n; // number of independent variables n  
double u[m]; // weight vector u  
double z[n]; // resulting adjoint value  $z^T = u^T F'(x)$ 
```

# 程序代写代做 CS编程辅导

43

as first-order scalar reverse mode implementation that computes the product  $z^T = u^T F'(x)$  after calling `zos_reverse`, `fos_forward`, or `hos_forward` with `keep=1`. The corresponding first-order vector reverse mode driver is given by

```
int fov_reverse(tag,m,n,q,U,Z);  
short int tag; // tape identification  
int m; // number of dependent variables m  
int n; // number of independent variables n  
int q; // number of weight vectors q  
double U[q][m]; // weight matrix U  
double Z[q][n]; // resulting adjoints Z = UF'(x)
```

that can be used after calling `zos_forward`, `fos_forward`, or `hos_forward` with `keep=1`. To compute higher-order derivatives, ADOL-C provides

## Assignment Project Exam Help

```
int hos_reverse(tag,m,n,d,u,Z);  
short int tag; // tape identification  
int m; // number of dependent variables m  
int n; // number of independent variables n  
int d; // highest derivative degree d  
double u[m]; // weight vector u  
double Z[1][d+1]; // resulting adjoints Z
```

as higher-order scalar reverse mode implementation yielding the adjoints  $z_0^T = u^T F'(x_0) = u^T A_0$ ,  $z_1^T = u^T F'(x_0)x_1 = u^T A_1$ , ..., where  $Z = [z_0, z_1, \dots, z_d]$  after calling `fos_forward` or `hos_forward` with `keep = d + 1 > 1`. The vector version is given by

```
int hov_reverse(tag,m,n,d,q,U,Z,nz);  
short int tag; // tape identification  
int m; // number of dependent variables m  
int n; // number of independent variables n  
int d; // highest derivative degree d  
double U[q][m]; // weight vector u  
double Z[q][n][d+1]; // resulting adjoints Z  
short int nz[q][n]; // nonzero pattern of Z
```

as higher-order vector reverse mode driver to compute the adjoints  $Z_0 = UF'(x_0) = UA_0$ ,  $Z_1 = UF''(x_0)x_1 = UA_1$ , ..., where  $Z = [Z_0, Z_1, \dots, Z_d]$  after calling `fos_forward` or `hos_forward` with `keep = d + 1 > 1`. After the function call, the last argument of `hov_reverse` contains information about the sparsity pattern, i.e. each `nz[i][j]` has a value that characterizes the functional relation between the  $i$ -th component of  $UF'(x)$  and the  $j$ -th independent value  $x_j$  as:



0 polynomial

1 rational

4 transcendental

5 non-smooth

Here, “trivial” means that there is no dependence at all and “linear” means that the partial derivative is a constant. “Rational” means that one or more terms in the function are dependent on other variables either. “Non-smooth” means that one or more terms in the function are discontinuous. “Transcendental” means that the path between  $x_i$  and  $y_j$  was evaluated at a point where it is not differentiable. The five labels 1, 2, 3, 4, 5 are pessimistic in that the actual functional relation may in fact be simpler, for example due to exact cancellations.

There are also overloaded versions providing a general reverse-call. Details of the appropriate calling sequences are given in the following Section 5.

WeChat: cstutorcs

## 4.1 Drivers for Abs-Normal Form

# Assignment Project Exam Help

In this subsection we consider functions  $y = F(c) : \mathbb{R}^l \rightarrow \mathbb{R}^m$  that are non-smooth because of the occurrence of the absolute value function. The drivers provided generate a piecewise-linear approximation of the function  $F$  in a point  $x_0$  and first order derivatives of this second order approximation. The piecewise-linear model will be called piecewise linearization in the following. Further information about the piecewise linearization you can find in [7].

The piecewise linearization is given in *abs-normal* form by

$$\text{QQ: } \begin{bmatrix} z \\ y \end{bmatrix} = \begin{bmatrix} b \\ Z \end{bmatrix} + \begin{bmatrix} L \\ J \end{bmatrix} \begin{bmatrix} x \\ |z| \end{bmatrix}.$$

Here  $z \in \mathbb{R}^s$  is a vector of  $s \geq 0$  *switching variables* and correspondingly the two vectors and four matrices specifying the function  $F$  have the form

$$c \in \mathbb{R}^s, \quad Z \in \mathbb{R}^{s \times n}, \quad L \in \mathbb{R}^{s \times s}, \quad b \in \mathbb{R}^m, \quad J \in \mathbb{R}^{m \times n}, Y \in \mathbb{R}^{m \times s}.$$

The matrix  $L$  is strictly lower triangular.

To compute the piecewise linearization  $\Delta F(x_0; x)$  the abs-normal mode has to be enabled by using

```
enableMinMaxUsingAbs()
```

before the *trace\_on()* command. If one is interested in the number  $s$  of absolute value functions occurring in the function evaluation, one can get it by

```
int get_num_switches(tag)
short int tag; // tape identification
```

after tracing the function  $F$ .

# 程序代写代做 CS编程辅导

## 4.1 Drivers for Abs-Normal Form

45

In abs-normal form  $y = F(x)$ . The driver computes the absolute value of  $y$  as  $|y| = \sigma_i * z_i$ . The signature vector  $\sigma$  is called signature.



are available. For a start there is a driver to evaluate the values of  $z$  which are the arguments of the intermediate evaluations of the absolute value function is interpreted in the zero order mode. The vector  $\sigma \equiv \{-1, 0, 1\}^s$  is

```
int zos_pl(x,y,z)
short int tag; // tape identification
int n; // number of independent variables n and m = 1
int m; // number of dependent variables m
int keep; // flag for reverse mode preparation
double x[n]; // independent vector x
double y[m]; // dependent vector y = F(x)
double z[s]; // argument of abs(z)
```

WeChat: cstutorcs  
Assignment Project Exam Help

Additionally, there are drivers for the forward mode to evaluate first order derivatives of the piecewise linearization. For first order derivatives another signature vector generated by

Email: tutorcs@163.com

```
int firstsign(z_i, Z[i])
double z[s]; // i-th component of argument of abs(z)
double Z[s][n]; // i-th row of first derivative Z = z'(x)X
```

has to be used.  $firstsign(u)$  of a vector  $u$  is defined as the  $sign()$  of the first non-vanishing component of  $u$  if that exists; otherwise the value is zero. More detailed information can be found in [7].

```
int fos_pl_forward(tag,m,n,x0,x1,y0,y1,z0,z2)
short int tag; // tape identification
int m; // number of dependent variables m
int n; // number of independent variables n
double x0[n]; // independent vector x
double x1[n]; // tangent vector x1
double y0[m]; // dependent vector y0 = F(x0)
double y1[m]; // first derivative y1 = F'(x0)x1
double z0[s]; // argument of abs(z0)
double z1[s]; // first derivative z1 = z'_0(x0)x1
```

```
int fov_pl_forward(tag,m,n,p,x,X,y,Y,z,Z)
short int tag; // tape identification
int m; // number of dependent variables m
```

# 程序代写代做 CS编程辅导

## 5 OVERLOADED FORWARD AND REVERSE CALLS

```

int n;           // number of independent variables n
int p;           // number of directions
double x[];      // independent vector x
double X[];      // tangent matrix X
double y[];      // dependent vector y = F(x)
double Y[];      // first derivative matrix Y = F'(x)X
double z[s];     // argument of abs(z)
double Z[s][p];  // first derivative matrix Z = z'(x)X

```



In contrast to the drivers above the reverse mode does not require any directions. It always returns one row of the Jacobian matrix.

```

int fov_pl_reverse(tag,m,n,s,rownum,z)
short int tag;          // tape identification
int m;                  // number of dependent variables m
int n;                  // number of independent variables n
int s;                  // number of sign switches
int rownum;             // required row no. of abs-normal form
double z[n];            // resulting adjoint value  $z^T = u^T F'(x)$ 

```

**QQ: 749389476**

One may also compute the sparsity pattern of the extended Jacobian matrix

```

#include <adolc/sparse/sparsedrivers.h>
int absnormal_jac_pat(tag,m,n,s,x,JP)
short int tag;          // tape identification
int m;                  // number of dependent variables m
int n;                  // number of independent variables n
int s;                  // number of sign switches
const double* x;        // point of evaluation
unsigned int** JP;      // sparsity pattern of abs-normal form

```

**Assignment Project Exam Help**

**Email: tutorcs@163.com**

**https://tutorcs.com**

## 5 Overloaded Forward and Reverse Calls

In this section, the several versions of the forward and reverse routines, which utilize the overloading capabilities of C++, are described in detail. With exception of the bit pattern versions all interfaces are prototyped in the header file `<adolc/interfaces.h>`, where also some more specialized forward and reverse routines are explained. Furthermore, ADOL-C provides C and Fortran-callable versions prototyped in the same header file. The bit pattern versions of forward and reverse introduced in the Section 5.3 are prototyped in the

# 程序代写代做 CS编程辅导

## 5.1 The Scalar Case

47

header file <adolc/tape.h>, which will be included by the header file <adolc/interface.h>.

### 5.1 The Scalar Case

Given any correctly formed tape identifier tag from within the generating program, or subsequently during another run, the following procedure:

```
int forward(tag,m,n,d,keep,X,Y)
short int tag;           // tape identification
int m;                  // number of dependent variables m
int n;                  // number of independent variables n
int d;                  // highest derivative degree d
int keep;               // flag for reverse sweep
double X[n][d+1];       // Taylor coefficients X of independent variables
double Y[m][d+1];       // Taylor coefficients Y as in (7)
```

WeChat: cstutorcs  
Assignment Project Exam Help  
Email: tutorcs@163.com

The rows of the matrix  $X$  must correspond to the independent variables in the order of their initialization by the  $\ll=$  operator. The columns of  $X = \{x_j\}_{j=0\dots d}$  represent Taylor coefficient vectors as in (5). The rows of the matrix  $Y$  must correspond to the dependent variables in the order of their selection by the  $\gg=$  operator. The columns of  $Y = \{y_j\}_{j=0\dots d}$  represent Taylor coefficient vectors as in (7). Thus the first column of  $Y$  contains the function value  $F(x)$  itself, the next column represents the first Taylor coefficient vector of  $F$ , and the last column the  $d$ -th Taylor coefficient vector. The integer flag `keep` determines how many Taylor coefficients of all intermediate quantities are written into the value stack as explained in Section 4. If `keep` is omitted, it defaults to 0.

The given `tag` value is used by `forward` to determine the name of the file on which the tape was written. If the tape file does not exist, `forward` assumes that the relevant tape is still in core and reads from the buffers. After the execution of an active section with `keep = 1` or a call to `forward` with any  $keep \leq d + 1$ , one may call the function `reverse` with  $d = keep - 1$  and the same tape identifier `tag`. When  $u$  is a vector and  $Z$  an  $n \times (d + 1)$  matrix `reverse` is executed in the scalar mode by the calling sequence

```
int reverse(tag,m,n,d,u,Z)
short int tag;           // tape identification
int m;                  // number of dependent variables m
int n;                  // number of independent variables n
int d;                  // highest derivative degree d
double u[m];            // weighting vector u
double Z[n][d+1];       // resulting adjoints Z
```

# 程序代写代做 CS编程辅导

5 OVERLOADED FORWARD AND REVERSE CALLS

to compute the adjoints  $Z_0 = U F'(x_0) = u^T A_0$ ,  $Z_1 = u^T F''(x_0)x_1 = u^T A_1$ , ..., where  $Z = [z_0, z_1, \dots, z_d]$



## 5.2 The Vec

When  $U$  is a matrix,  $Z$  is computed in the vector mode by the following calling sequence

```
int reverse(tag,m,n,d,q,U,Z,nz)
short int tag;           // tape identification
int m;                  // number of dependent variables m
int n;                  // number of independent variables n
int d;                  // highest derivative degree d
int q;                  // number of weight vectors q
double U[q][m];          // weight matrix U
double Z[q][n][d+1];    // resulting adjoints
short nz[q][n];          // nonzero pattern of Z
```

to compute the adjoints  $Z_0 = U F'(x_0) = u^T A_0$ ,  $Z_1 = u^T F''(x_0)x_1 = u^T A_1$ , ..., where  $Z = [Z_0, Z_1, \dots, Z_d]$ . When the arguments  $p$  and  $U$  are omitted, they default to  $m$  and the identity matrix of order  $m$ , respectively.

Through the optional argument `nz` or `reverse` one can compute information about the sparsity pattern of  $Z$  as described in detail in the previous Section 4.

The return values of `reverse` calls can be interpreted according to Table 1, but negative return values are not valid, since the corresponding forward sweep would have stopped without completing the necessary taylor file. The return value of `reverse` may be higher than that of the preceding `forward` call because some operations that were evaluated at a critical argument during the forward sweep were found not to impact the dependents during the reverse sweep.

In both scalar and vector mode, the degree  $d$  must agree with `keep - 1` for the most recent call to `forward`, or it must be equal to zero if `reverse` directly follows the taping of an active section. Otherwise, `reverse` will return control with a suitable error message. In order to avoid possible confusion, the first four arguments must always be present in the calling sequence. However, if  $m$  or  $d$  attain their trivial values 1 and 0, respectively, then corresponding dimensions of the arrays  $X$ ,  $Y$ ,  $u$ ,  $U$ , or  $Z$  can be omitted, thus eliminating one level of indirection. For example, we may call `reverse(tag,1,n,0,1.0,g)` after declaring `double g[n]` to calculate a gradient of a scalar-valued function.

Sometimes it may be useful to perform a forward sweep for families of Taylor series with the same leading term. This vector version of `forward` can be called in the form

```
int forward(tag,m,n,d,p,x0,X,y0,Y)
```

# 程序代写代做 CS编程辅导

5.2 The Vector Case

49

```
short int tag;           // tape identification
int m;                  // number of dependent variables  $m$ 
int n;                  // number of independent variables  $n$ 
int d;                  // highest derivative degree  $d$ 
int p;                  // number of Taylor series  $p$ 
double x0[n];           // values of independent variables  $x_0$ 
double X[m][n];         // Taylor coefficients  $X$  of independent variables
double y0[m];           // values of dependent variables  $y_0$ 
double Y[m][p][d];       // Taylor coefficients  $Y$  of dependent variables
```

WeChat: cstutorcs

where  $X$  and  $Y$  hold the Taylor coefficients of first and higher degree and  $x_0, y_0$  the common Taylor coefficients of degree 0. There is no option to keep the values of active variables that are going out of scope or that are overwritten. Therefore this function cannot prepare a subsequent reverse sweep. The return integer serves as a flag to indicate quadratures or altered comparisons as described above in Section 1.7.

Since the calculation of Jacobians is probably the most important automatic differentiation task, we have provided a specialization of `vector_forward` to the case where  $d = 1$ . This version can be called in the form

QQ: 749389476

```
int forward(tag,m,n,p,x,X,y,Y)
short int tag;           // tape identification
int m;                  // number of dependent variables  $m$ 
int n;                  // number of independent variables  $n$ 
int p;                  // number of partial derivatives  $p$ 
double x[n];
double X[n][p];
double y[m];
double Y[m][p];
// values of independent variables  $x_0$ 
// seed derivatives of independent variables  $X$ 
// values of dependent variables  $y_0$ 
// first derivatives of dependent variables  $Y$ 
```

<https://tutorcs.com>

When this routine is called with  $p = n$  and  $X$  the identity matrix, the resulting  $Y$  is simply the Jacobian  $F'(x_0)$ . In general, one obtains the  $m \times p$  matrix  $Y = F'(x_0)X$  for the chosen initialization of  $X$ . In a workstation environment a value of  $p$  somewhere between 10 and 50 appears to be fairly optimal. For smaller  $p$  the interpretive overhead is not appropriately amortized, and for larger  $p$  the  $p$ -fold increase in storage causes too many page faults. Therefore, large Jacobians that cannot be compressed via column coloring as could be done for example using the driver `sparse_jac` should be “strip-mined” in the sense that the above first-order-vector version of `forward` is called repeatedly with the successive  $n \times p$  matrices  $X$  forming a partition of the identity matrix of order  $n$ .

# 程序代写代做 CS编程辅导

50

OVERLOADED FORWARD AND REVERSE CALLS

## 5.3 Dependence Structures

The sparsity pattern of a dependence structure is needed to set up data structures for their storage and factorization and the evaluation of



ten needed to set up data structures for their storage and factorization [1]. Compared to the evaluation of  $\tilde{P} \in \{0, 1\}^{m \times n}$  in real arithmetic computing the Boolean matrix  $\tilde{P}_{ji}$  in the obvious way requires a little less run-time and memory.

The entry  $\tilde{P}_{ji}$  in the  $j$ -th row and  $i$ -th column of  $\tilde{P}$  should be 1 = *true* exactly when there is a data dependence between the  $i$ -th independent variable  $x_i$  and the  $j$ -th dependent variable  $y_j$ . Just like for real arguments one would wish to compute matrix-vector and vector-matrix products of the form  $\tilde{P}\tilde{v}$  or  $\tilde{u}^T\tilde{P}$  by appropriate forward and reverse routines where  $\tilde{v} \in \{0, 1\}^n$  and  $\tilde{u} \in \{0, 1\}^m$ . Here, multiplication corresponds to logical AND and addition to logical OR, so that algebra is performed in a semi-ring.

For practical reasons it is assumed that  $s = \text{sizeof}(int)$  such Boolean vectors  $\tilde{v}$  and  $\tilde{u}$  are combined to integer vectors  $v \in \mathbb{N}^n$  and  $u \in \mathbb{N}^m$  whose components can be interpreted as bit patterns. Moreover  $p$  or  $q$  such integer vectors may be combined column-wise or row-wise to integer matrices  $X \in \mathbb{N}^{n \times p}$  and  $U \in \mathbb{N}^{q \times m}$ , which naturally correspond to Boolean matrices  $\tilde{X} \in \{0, 1\}^{n \times p}$  and  $\tilde{U} \in \{0, 1\}^{q \times m}$ . The provided bit pattern versions of forward and reverse allow to compute integer matrices  $Y \in \mathbb{N}^{m \times p}$  and  $Z \in \mathbb{N}^{q \times m}$  corresponding to

$$\tilde{Y} = \tilde{U}\tilde{X} \quad \text{and} \quad \tilde{Z} = \tilde{U}\tilde{P}, \quad (10)$$

respectively, with  $\tilde{Y} \in \{0, 1\}^{m \times (sp)}$  and  $\tilde{U} \in \{0, 1\}^{(sq) \times n}$ . In general, the application of the bit pattern versions of forward or reverse can be interpreted as propagating dependences between variables forward or backward; therefore both the propagated integer matrices and the corresponding Boolean matrices are called *dependence structures*.

The bit pattern forward routine

```
int forward(tag,m,n,p,x,X,y,Y,mode)
short int tag;           // tape identification
int m;                  // number of dependent variables m
int n;                  // number of independent variables n
int p;                  // number of integers propagated p
double x[n];
unsigned long int X[n][p]; // dependence structure X
double y[m];
unsigned long int Y[m][p]; // dependence structure Y according to (10)
char mode;              // 0 : safe mode (default), 1 : tight mode
```

can be used to obtain the dependence structure  $Y$  for a given dependence structure  $X$ . The dependence structures are represented as arrays of `unsigned long int` the entries of which are

# 程序代写代做 CS编程辅导

51

interpreted as bit pattern  above. For example, for  $n = 3$  the identity matrix  $I_3$  should be passed as a  $3 \times 1$  array

$$\begin{pmatrix} 0000000 & 00000000 & 00000000_2 \\ 0000000 & 00000000 & 00000000_2 \\ 0000000 & 00000000 & 00000000_2 \end{pmatrix}$$

in the 4-byte long integer  parameter `mode` determines the mode of dependence analysis as explained already in Section 3.3.

A call to the corresponding bit pattern reverse routine

WeChat: cstutorcs

```
int reverse(tag,m,n,q,U,Z,mode)
short int tag;           // tape identification
int m;                  // number of dependent variables m
int n;                  // number of independent variables n
int q;                  // number of integers propagated q
unsigned long int U[q][m]; // dependence structure U
unsigned long int Z[q][n]; // dependence structure Z according to (10)
char mode;              // 0: safe mode (default), 1: tight mode
```

yields the dependence structure  $Z$  for a given dependence structure  $U$ .

QQ: 749389476

To determine the whole sparsity pattern  $P$  of the Jacobian  $F'(x)$  as an integer matrix  $P$  one may call `forward` or `reverse` with  $p \geq n/s$  or  $q \geq m/s$ , respectively. For this purpose the corresponding dependence structure  $X$  or  $U$  must be defined to represent the identity matrix of the respective dimension. Due to the fact that always a multiple of  $s$  Boolean vectors are propagated there may be superfluous vectors, which can be set to zero.

The return values of the bit pattern `forward` and `reverse` routines correspond to those described in Table 1.

One can control the storage growth by the factor  $p$  using “strip-mining” for the calls of `forward` or `reverse` with successive groups of columns or respectively rows at a time, i.e. partitioning  $X$  or  $U$  appropriately as described for the computation of Jacobians in Section 5.2.

## 6 Advanced algorithmic differentiation in ADOL-C

### 6.1 Differentiating parameter dependent functions

Normally the functions to be differentiated using AD are defined as mappings from  $\mathbb{R}^n$  to  $\mathbb{R}^m$ . The derivatives of the  $m$  dependents are then computed w.r.t. all the  $n$  independents. However sometimes the application requires only derivatives w.r.t. some of the independents

and the others one can use case is Lagrange variables as `adouble`. derivative information type `adouble` want to change the function may



ation procedure as parameters. One important such nonlinear optimization problem. Declaring such variables independent results in the evaluation of superfluous intime. However, if the parameters are not defined as `adouble` coded as constants onto the ADOL-C trace. One may some evaluations, however the overhead of retracing in such cases these constants may be specially marked as parameters during tracing using the function

```
mkparam(d)
double d;
```

## WeChat: cstutorcs

// constant value to be marked as parameter

This stores the constant in a different place on the trace than the other unmapped constants and enables the user to replace a whole vector of parameters with a new one before computing derivatives later on. All such marked parameter constants can be used in any evaluation expression where the original unmarked constants were used.

Sometimes the same parameter may be needed in various different expressions. During trace creation the marked parameters are given a running index as they are marked. This index may be saved by the user and used to access the same stored value in different expressions.

## QQ: 749389476

To save the index while marking a parameter use

```
locint mkparam_idx(d)
double d;
```

## https://tutorcs.com

// constant value to be marked

The resulting `locint` can be saved in a variable by the user and later used to access the saved parameter from in the trace in any expression

To access a saved parameter inside any evaluation expression use

```
getparam(idx)
locint idx;
```

// index of the parameter as returned by `mkparam_idx()`

The above three functions work only during the creation of the trace, i.e., between the invocation of `trace_on()` and `trace_off()`.

For all following derivative computations using the drivers described in the preceding sections, the saved parameter values will be used, unless the user asks to replace them with a different vector of values.

To replace the whole parameter vector one uses the routine

# 程序代写代做 CS编程辅导

## 6.2 Differentiating external functions

53

```
set_param( const char* tape_id, size_t num_params, double* p )  
short int set_param( const char* tape_id, size_t num_params, double* p )  
    // tape identification  
    // total number of constants marked as parameters  
    // array of constants to be used as replacement
```

All following derivative drivers using the drivers described in the preceding sections will now use the new parameters.

After replacing the parameters, however, one cannot directly call a basic reverse mode driver. A basic forward mode driver must be called. However, the easy to use drivers from Section 3 do this automatically.

### 6.2 Differentiating external functions

## Assignment Project Exam Help

Ideally, AD is applied to a given computation as a whole. This is not always possible because parts of the computation may be coded in a different programming language or may be a call to an external library. In the former case one may want to differentiate the parts in question with a different AD tool or provide hand written derivatives. In practice, however, sophisticated projects usually evolve over a long period of time. Within this process, a heterogeneous code base for the project develops, which may include the incorporation of external solutions, changes in programming paradigms or even of programming languages. Equally heterogeneous, the computation of derivative values appears. Hence, different AD-tools may be combined with hand-derived codes based on the same or different programming languages. ADOL-C supports such settings by the concept of externally differentiated functions. That is, a function not differentiated by ADOL-C itself. The required derivatives have to be provided by the user.

For this purpose, it is required that the externally differentiated function (for example named *euler\_step*) has the following signature

```
int euler_step(int n, double *x, int m, double *y); .
```

Note that the formal parameters in the signature have *double* type, that is, they are not active as in the original program before the ADOL-C type change. The externally differentiated function has to be *registered*<sup>4</sup> using an ADOL-C method as follows

```
ext_diff_fct *edf = reg_ext_fct(euler_step); .
```

This returns a pointer to an *ext\_diff\_fct* instance specific to the registered function. Then, the user has to supply the function pointers for the callback methods (for example here *zos\_for\_euler\_step* and *fos\_rev\_euler\_step*) the user implemented to compute the derivatives as follows:

---

<sup>4</sup>we record the function pointer

```

edf->_euler_step;
// fu putting Zero-Order-Scalar (=zos)
// fo
edf->_euler_step;
// fu putting First-Order-Scalar (=fos)
rever

```

To facilitate the switch between active and passive versions of the parameters  $x$  and  $y$  one has to provide (allocate) both variants. I.e. if the call to `euler_step` was originally

**WeChat: cstutorcs**

then the ADOL-C typechange for the calling context will turn `sOld` and `sNew` in `adouble` pointers. To trigger the appropriate action for the derivative computation (i.e. creating an external differentiation entry on the trace) the original call to the externally differentiated function must be substituted by

`rc=call_ext_fct(edf, n, sOldPassive, sOld, m, sNewPassive, sNew); .`

**Email: tutorcs@163.com**

Here, `sOldPassive` and `sNewPassive` are the passive counterparts (`double` pointers allocated to length `n` and `m`, respectively) to the active arguments `sNew` in `adouble`. The usage of the external function facility is illustrated by the example `ext_diff_func` contained in `examples/additional_examples/ext_diff_func`. There, the external differentiated function is also a C code, but the handling as external differentiated function decreases the overall required tape size.

**<https://tutorcs.com>**

### 6.3 Advanced algorithmic differentiation of time integration processes

For many time-dependent applications, the corresponding simulations are based on ordinary or partial differential equations. Furthermore, frequently there are quantities that influence the result of the simulation and can be seen as control of the systems. To compute an approximation of the simulated process for a time interval  $[0, T]$  and evaluated the desired target function, one applies an appropriate integration scheme given by

```

some initializations yielding  $x_0$ 
for  $i = 0, \dots, N - 1$ 
 $x_{i+1} = F(x_i, u_i, t_i)$ 
evaluation of the target function

```

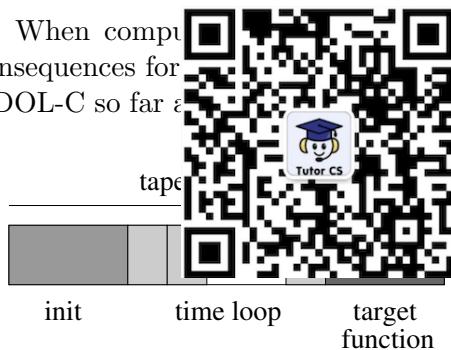
where  $x_i \in \mathbf{R}^n$  denotes the state and  $u_i \in \mathbf{R}^m$  the control at time  $t_i$  for a given time grid  $t_0, \dots, t_N$  with  $t_0 = 0$  and  $t_N = T$ . The operator  $F : \mathbf{R}^n \times \mathbf{R}^m \times \mathbf{R} \mapsto \mathbf{R}^n$  defines the time step to compute the state at time  $t_i$ . Note that we do not assume a uniform grid.

# 程序代写代做 CS编程辅导

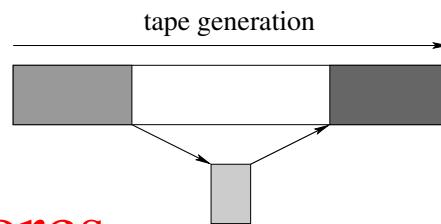
6.3 Advanced algorithmic differentiation of time integration processes

55

When computing the target function with respect to the control, the consequences for using the “basic” taping approach as implemented in ADOL-C so far are shown in the left part of Figure 4. As can be seen, the iterative process



WeChat: cstutorcs  
Basic taping process



Advanced taping process

Figure 4: Different taping approaches

## Assignment Project Exam Help

is completely unrolled due to the taping process. That is, the tape contains an internal representation of each time step. Hence, the overall tape comprises a serious amount of redundant information, as illustrated by the light grey rectangles in Figure 4.

Email: [tutors@163.com](mailto:tutors@163.com)

To overcome the repeated storage of essentially the same information, a *nested taping* mechanism has been incorporated into ADOL-C as illustrated on the right-hand side of Figure 4. This new capability allows the encapsulation of the time-stepping procedure such that only the last time step  $x_N = F(x_{N-1}, u_{N-1})$  is taped as one representative of the time steps in addition to a function pointer to the evaluation procedure  $F$  of the time steps. The function pointer has to be stored for a possibly necessary retaping during the derivative calculation as explained below.

<https://tutorcs.com>

Instead of storing the complete tape, only a very limited number of intermediate states are kept in memory. They serve as checkpoints, such that the required information for the backward integration is generated piecewise during the adjoint calculation. For this modified adjoint computation the optimal checkpointing schedules provided by **revolve** are employed. An adapted version of the software package **revolve** is part of ADOL-C and automatically integrated in the ADOL-C library. Based on **revolve**,  $c$  checkpoints are distributed such that computational effort is minimized for the given number of checkpoints and time steps  $N$ . It is important to note that the overall tape size is drastically reduced due to the advanced taping strategy. For the implementation of this nested taping we introduced a so-called “differentiating context” that enables ADOL-C to handle different internal function representations during the taping procedure and the derivative calculation. This approach allows the generation of a new tape inside the overall tape, where the coupling of the different tapes is based on the *external differentiated function* described above.

Written under the objective of minimal user effort, the checkpointing routines of ADOL-C need only very limited information. The user must provide two routines as implementation of the time-stepping function  $F$  with the signatures

```
int ti
int ti
```



where the function names can be chosen by the user as long as the names are unique. It is possible that the time step iteration overwrites the argument vector of the same time step. Iterations are required to prepare the next time step.

At first, the time step function has to be *registered* using the ADOL-C function

```
CP_Context cpc(time_step_function);
```

## WeChat: cstutorcs

This function initializes the structure `cpc`. Then, the user has to provide the remaining checkpointing information by the following commands:

## Assignment Project Exam Help

```
cpc.setDoublePct(time_step_function);
// double variante of the time step function
cpc.setNumberOfSteps(N);
// number of time steps to perform
cpc.setNumberOfCheckpoints(10);
// number of checkpoint
cpc.setDimensionXY(n);
// dimension of input/output
cpc.setInput(y);
// input vector
cpc.setOutput(y);
// output vector
cpc.setTapeNumber(tag_check);
// subtape number for checkpointing
cpc.setAlwaysRetaping(false);
// always retape or not ?
```

QQ: 749389476

<https://tutorcs.com>

Subsequently, the time loop in the function evaluation can be substituted by a call of the function

```
int cpc.checkpointing();
```

Then, ADOL-C computes derivative information using the optimal checkpointing strategy provided by `revolve` internally, i.e., completely hidden from the user.

The presented driver is prototyped in the header file `<adolc/checkpointing.h>`. This header is included by the global header file `<adolc/adolc.h>` automatically. An example program `checkpointing.cpp` illustrates the checkpointing facilities. It can be found in the directory `examples/additional_examples/checkpointing`.

# 程序代写代做 CS编程辅导

6.4 Advanced algorithmic differentiation of fixed point iterations

57

## 6.4 Advanced



### differentiation of fixed point iterations

Quite often, the state vector  $x \in \mathbb{R}^n$  of a system denoted by some design parameters denoted by  $u \in \mathbb{R}^m$  is determined by a flow equation. An example for this setting forms the flow over an aircraft wing. Here, the state  $x$  is defined by the design vector  $u$  determines the flow field  $x$ . The state  $x_*$  fulfills the fixed point equation

$$x_* = F(x_*, u) \quad (11)$$

for a given continuously differentiable function  $F : \mathbb{R}^n \times \mathbb{R}^m \rightarrow \mathbb{R}^n$ . A fixed point property of this kind is also exploited by many other applications.

WeChat: cstutorcs

Assume that one can apply the iteration

$$x_{k+1} = F(x_k, u) \quad (12)$$

to obtain a linear converging sequence  $\{x_k\}$  generated for any given control  $u \in \mathbb{R}^m$ . Then the limit point  $x_* \in \mathbb{R}^n$  fulfills the fixed point equation (11). Moreover, suppose that  $\|\frac{dF}{dx}(x_*, u)\| < 1$  holds for any pair  $(x_*, u)$  satisfying equation (11). Hence, there exists a differentiable function  $\phi : \mathbb{R}^n \rightarrow \mathbb{R}^n$ , such that  $\phi(u) = F(\phi(u), u)$ , where the state  $\phi(u)$  is a fixed point of  $F$  according to a control  $u$ . To optimize the system described by the state vector  $x = \phi(u)$  with respect to the design vector  $u$ , derivatives of  $\phi$  with respect to  $u$  are of particular interest.

QQ: 749389476

To exploit the advanced algorithmic differentiation of such fixed point iterations ADOL-C provides the special functions `fp_iteration(...)`. It has the following interface:

```
int fp_iteration(sub_tape_num, double *adouble_F, adouble_F_norm, norm_deriv, eps, eps_deriv,
                N_max, N_max_deriv, x_0, u, x_fix, dim_x, dim_u)

short int sub_tape_num;           // tape identification for sub_tape
int *double_F;                  // pointer to a function that compute for x and u
                                 // the value y = F(x, u) for double arguments
int *adouble_F;                 // pointer to a function that compute for x and u
                                 // the value y = F(x, u) for double arguments
int *norm;                      // pointer to a function that computes
                                 // the norm of a vector
int *norm_deriv;                // pointer to a function that computes
                                 // the norm of a vector
double eps;                     // termination criterion for fixed point iteration
double eps_deriv;               // termination criterion for adjoint fixed point iteration
N_max;                          // maximal number of iterations for state computation
N_max_deriv;                    // maximal number of iterations for adjoint computation
adouble *x_0;                   // initial state of fixed point iteration
adouble *u;                      // value of u
adouble *x_fix;                 // final state of fixed point iteration
```

# 程序代写代做 CS编程辅导

58

6 ADVANCED ALGORITHMIC DIFFERENTIATION IN ADOL-C

```
int dim_x;  
int dim_u;
```



dimension of  $x$

dimension of  $u$

Here `sub_tape_n` is an identifier for the subtape that should be used for the fixed point iteration. `double_F` and `adouble_F` are pointers to functions, that compute for  $x$  and  $u$  a single value  $y = F(x, u)$  and  $\bar{y} = F'(x, u)$ . Thereby `double_F` uses `double` arguments and `adouble_F` uses ADOL-C `adouble` arguments. The parameters `norm` and `norm_deriv` are pointers to functions computing the norm of a vector. The latter functions together with `eps`, `eps_deriv`, `N_max`, and `N_max_deriv` control the iterations. Thus the following loops are performed:

WeChat: cstutorcs

```
do  
  k = k + 1  
  x = y  
  y = F(x, u)  
  while ||y - x|| ≥ ε and k ≤ Nmax
```

```
do  
  k = k + 1  
  ξ = ζ  
  (ξT, ȳT) = ζTF'(x*, u) + (ȳT, 0T)  
  while ||ξ - ζ||deriv ≥ εderiv  
    and k ≤ Nmax_deriv
```

**Assignment Project Exam Help**  
**Email: tutorcs@163.com**

The vector for the initial iterate and the control is stored in `x_0` and `u` respectively. The vector in which the fixed point is stored is `x_fix`. Finally `dim_x` and `dim_u` represent the dimensions  $n$  and  $m$  of the corresponding vectors.

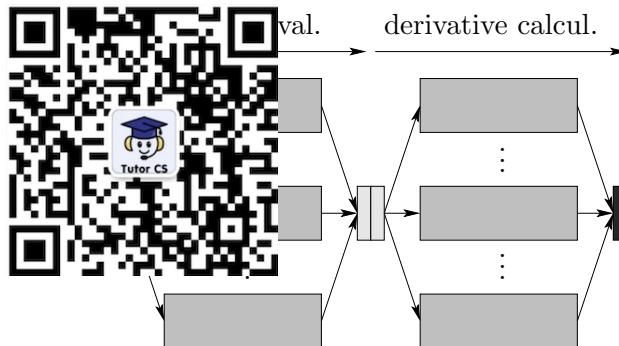
**QQ: 749389476**

The presented driver is prototyped in the header file `<adolc/fixpoint.h>`. This header is included by the global header file `<adolc/adolc.h>` automatically. An example code that shows also the expected signature of the function pointers is contained in the directory `examples/additional/examples/fixpoint_exam`.

## 6.5 Advanced algorithmic differentiation of OpenMP parallel programs

ADOL-C allows to compute derivatives in parallel for functions containing OpenMP parallel loops. This implies that an explicit loop-handling approach is applied. A typical situation is shown in Figure 5, where the OpenMP-parallel loop is preceded by a serial startup calculation and followed by a serial finalization phase.

Initialization of the OpenMP-parallel regions for ADOL-C is only a matter of adding a macro to the outermost OpenMP statement. Two macros are available that only differ in the way the global tape information is handled. Using `ADOLC_OPENMP`, this information, including the values of the augmented variables, is always transferred from the serial to the parallel region using `firstprivate` directives for initialization. For the special case of iterative codes where parallel regions, working on the same data structures, are called repeatedly the `ADOLC_OPENMP_NC` macro can be used. Then, the information transfer is performed only once within the iterative process upon encounter of the first parallel region through use



WeChat: cstutorcs

Figure 5: Basic layout of mixed function and the corresponding derivation process

of the *threadprivate* feature of OpenMP that makes use of thread-local storage, i.e., global memory local to a thread. Due to the inserted macro, the OpenMP statement has the following structure:

```
#pragma omp ... ADOLC_OPENMP          or
#pragma omp ... ADOLC_OPENMP_P_N
```

Email: tutorcs@163.com  
QQ: 749389476

Inside the parallel region, separate tapes may then be created. Each single thread works in its own dedicated AD-environment, and all serial facilities of ADOL-C are applicable as usual. The global derivatives can be computed using the tapes created in the serial and parallel parts of the function evaluation, where user interaction is required for the correct derivative concatenation of the various tapes.

For the usage of the parallel facilities, the `configure` command has to be used with the option `--with-openmp-flag=FLAG`, where FLAG stands for the system dependent OpenMP flag. The parallel differentiation of a parallel program is illustrated by the example program `openmp_exam.cpp` contained in `examples/additional_examples/openmp_exam`.

## 7 Traceless forward differentiation in ADOL-C

Up to version 1.9.0, the development of the ADOL-C software package was based on the decision to store all data necessary for derivative computation on traces (tapes), where large applications require the traces to be written out to corresponding files. In almost all cases this means a considerable drawback in terms of run time due to the excessive memory accesses. Using these traces enables ADOL-C to offer multiple functions. However, it is not necessary for all tasks of derivative computation to do that.

Starting with version 1.10.0, ADOL-C now features a traceless forward mode for computing first order derivatives in scalar mode, i.e.,  $\dot{y} = F'(x)\dot{x}$ , and in vector mode, i.e.,  $\dot{Y} = F'(x)\dot{X}$ . This traceless variant coexists with the more universal trace based mode in

the package. The QR code describes the source code modifications required to use the traceless mode.



## 7.1 Modifying the Source Code

Let us consider the transformation from Cartesian to spherical polar coordinates given by the function  $y = F(x)$ , with

$$y_1 = \sqrt{x_1^2 + x_2^2 + x_3^2}, \quad y_2 = \arctan\left(\sqrt{x_1^2 + x_2^2}/x_3\right), \quad y_3 = \arctan(x_2/x_1),$$

as an example. The corresponding source code is shown in Figure 6. Changes to the source

#include <iostream>  
using namespace std;

int main() {  
 double x[3], y[3];

for (int i=0; i<3; ++i) // Initialize  $x_i$

...  
**QQ: 749389476**

y[0] = sqrt(x[0]\*x[0]+x[1]\*x[1]+x[2]\*x[2]);

y[1] = atan(sqrt(x[0]\*x[0]+x[1]\*x[1])/x[2]);

y[2] = atan(x[1]/x[0]);

**https://tutorcs.com**

cout << "y1=" << y[0] << " , y2=" << y[1] << " , y3=" << y[2] << endl;

return 0;

}

Figure 6: Example for traceless forward mode

code that are necessary for applying the traceless forward mode of ADOL-C are described in the following two subsections, where the vector mode version is described as extension of the scalar mode.

### The scalar mode

To use the traceless forward mode, one has to include only the header file `adtl.h` since it does not include the trace based functions defined in other header files.

# 程序代写代做 CS编程辅导

## 7.1 Modifying the Source Code

61

As in the trace-based version of ADOL-C all derivative calculations are introduced by call-back functions. Therefore, similar to the trace-based version all independent, dependent variables must be declared with type `adouble`. The whole trace-based code provided by `adt1.h` was written as complete inline functions, where the real portion of inlined code can be influenced by switches, where the real portion of inlined code can be influenced by switches. Likely, the whole derivative code is inlined by default. Our experiments have produced complete inlined code by using standard switches (optimization) for GNU and Intel C++ compiler.

To avoid name conflicts resulting from the inlining the traceless version has its own namespace `adt1`. As a result four possibilities of using the `adouble` type are available for the traceless version:

- Defining a new type

**Assignment Project Exam Help**

```
typedef adtl::adouble adouble;
```

...

```
adouble tmp;
```

**Email: tutorcs@163.com**

This is the preferred way. Remember, you can not write an own `adouble` type/class with different meaning after doing the `typedef`.

- Declaring with namespace prefix

**QQ: 749389476**

```
adtl::adouble tmp;
```

Not the most handsome and efficient way with respect to coding but without any doubt one of the safest ways. The identifier `adouble` is still available for user types/classes.

- Trusting macros

```
#define adouble adtl::adouble
```

...

```
adouble tmp;
```

This approach should be used with care, since standard defines are text replacements.

- Using the complete namespace

```
using namespace adtl;
```

...

```
adouble tmp;
```

A very clear approach with the disadvantage of uncovering all the hidden secrets. Name conflicts may arise!

# 程序代写代做 CS 编程辅导

*TRACELESS FORWARD DIFFERENTIATION IN ADOL-C*

After defining the values of the variables, the traceless forward differentiation can be used without assigning the value to the variable. To set the **ad**-values of the independent variables



things are left to do. First one needs to initialize the **ad**-values for the function evaluation. This can be done by setting the **ad**-value to zero in this case. Additionally, ADOL-C offers a function named **setValue** for setting the **ad**-value. To set the **ad**-values of the independent variables

- Using the constructor

WeChat: cstutorcs

```
double seedOne = 1., seedZero = 0.;
adouble x1(2,&seedOne), x2(4,&seedZero), y;
```

This would create three **adoubles**  $x_1$ ,  $x_2$  and  $y$ . Obviously, the latter remain uninitialized. In terms of function evaluation,  $x_1$  holds the value 2 and  $x_2$  the value 4 whereas the derivative values are initialized to  $\dot{x}_1 = 1$  and  $\dot{x}_2 = 0$ .

- Setting point values directly

Email: tutorcs@163.com

QQ: 749389476  
 double seedOne = 1., seedZero = 0.;
 adouble x1=2, x2=4, y;
 ...
 x1.setADValue(&seedOne);
 x2.setADValue(&seedZero);

<https://tutorcs.com>

The same example as above but now using **setADValue**-method for initializing the derivative values. It is important to note that in both cases the derivative values have to be set by passing an address of a variable or a pointer of the **double** type (instead of passing the value directly). The reason for doing this will become clear in the next subsection.

The derivatives can be obtained at any time during the evaluation process by calling the **getADValue**-method which returns a pointer (instead of a value), and therefore, it can be assigned to another pointer or it can be dereferenced immediately in order to read the value:

```
adouble y;
...
cout << *(y.getADValue());
```

Figure 7 shows the resulting source code incorporating all required changes for the example given above.

# 程序代写代做 CS编程辅导

## 7.1 Modifying the Source Code

63



<https://tutorcs.com>

## The vector mode

In scalar mode only one direction element has to be stored per `adouble` whereas a field of  $p$  elements is needed in the vector mode to cover the computations for the given  $p$  directions. The resulting changes to the source code are described in this section.

In order to use the vector mode one has to call `adtl::setNumDir(N)`. This function takes the maximal number of directions to be used within the resulting vector mode. Changing the number of directions in the middle of executing code containing active variables is undesirable and can lead to segmentation violations, hence such an operation will result in an error. `adtl::setNumDir(N)` must therefore be called before declaring any `adtl::adouble` objects.

Setting and getting the derivative values is done in the same manner as in the scalar case, by passing and retrieving the pointers, as illustrated in the following example:



```

NumDir(10);
x, y;
ptr1 = new double[10];
ptr2;
...Value(ptr1);
ptr2=y.getADValue();

```

## WeChat: estutorcs

Now using the pointers in the methods `setADValue` and `getADValue` makes more sense since the vector mode requires the field of  $p$  elements per `adouble` which holds the derivative values. Additionally, the traceless vector forward mode of ADOL-C offers two new methods for setting/getting the derivative values. Setting a derivative value of the desired vector element is possible by passing a `double` value and its position in the vector (`integer`). Getting a derivative value of the desired vector element requires as argument only its position in the vector. One can note that the pointers are not used in these cases, as illustrated in the following example:

**Email: tutorcs@163.com**

```

adtl::setNumDir(10);
...
adouble x, y;
...
x1=2;
x1.setADValue(5,1);           // set the 6th point value of x to 1.0
...
cout << y.getADValue(3) << endl; // print the 4th derivative value of y

```

**QQ: 749389476**

**<https://tutorcs.com>**

The resulting source code containing all changes that are required is shown in Figure 8

## 7.2 Compiling and Linking the Source Code

After incorporating the required changes, one has to compile the source code and link the object files to get the executable. As long as the ADOL-C header files are not included in the absolute path the compile sequence should be similar to the following example:

```
g++ -I/home/username/adolc_base/include -c traceless_scalar.cpp
```

The `-I` option tells the compiler where to search for the ADOL-C header files. This option can be omitted when the headers are included with absolute path or if ADOL-C is installed in a “global” directory.

# 程序代写代做 CS编程辅导

7.2 Compiling and Linking the Source Code

65

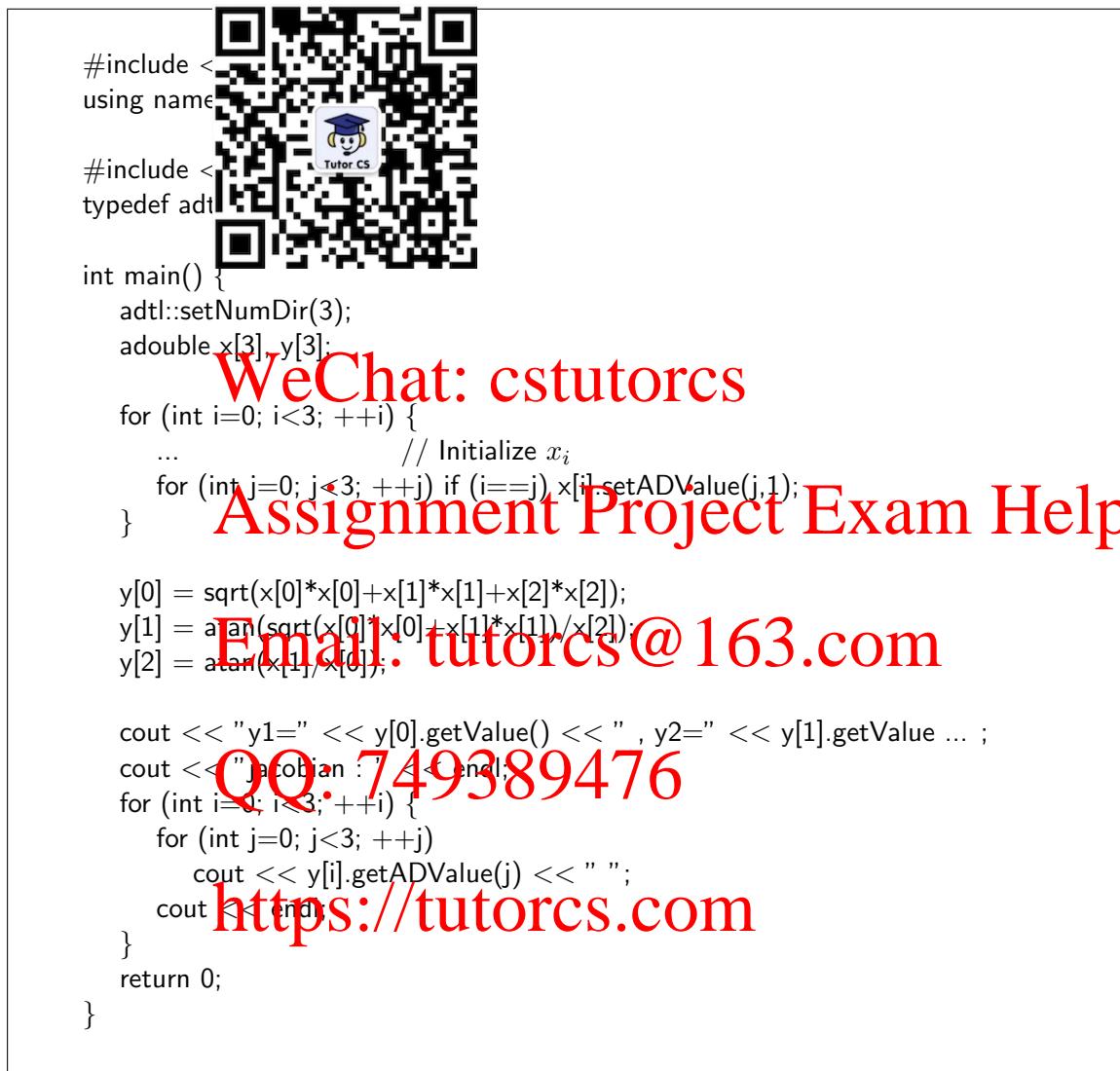


Figure 8: Example for traceless vector forward mode

Although originally designed to be completely inline, certain global variables need to be initialised in the compiled part of the library and therefore the ADOL-C library must be linked with the compiled code. The example started above could be finished with the following command:

```
g++ -o traceless_scalar traceless_scalar.o  
-WI,--rpath -WI,/home/username/adolc_base/lib -L/home/username/adolc_base/lib -ladolc
```

The mentioned source codes `traceless_scalar.c` and `traceless_vector.c` illustrating the use of the for traceless scalar and vector mode can be found in the directory `examples`.

# 程序代写代做 CS编程辅导

*7 TRACELESS FORWARD DIFFERENTIATION IN ADOL-C*

### 7.3 Conclusion of the Traceless Forward Mode Variant

As many other ADOL-C modes, the traceless forward mode provided by the ADOL-C package has its own strengths and weaknesses. Please read the following section carefully to become familiar with the pros and cons of this mode.

- Advantages:

- Code speed

Increasing computation speed was one of the main aspects in writing the traceless code. In many cases higher performance can be expected this way. However this is not guaranteed for repeated evaluations.

- Smaller overall memory requirements

Traceless ADOL-C does not write traces anymore, as the name implies. Loop "unrolling" can be avoided this way. Considering main memory plus disk space as overall memory requirements the traceless version can be executed in a more efficient way.

- Drawbacks:

**Email: tutorcs@163.com**

- Main memory limitations

The ability to compute derivatives to a given function is bounded by the main memory plus swap size when using traceless ADOL-C. Computation from swap should be avoided anyway as far as possible since it slows down the computing time drastically. Therefore, if the program execution is terminated without error message insufficient memory size can be the reason among other things. The memory requirements  $M$  can be determined roughly as followed:

\* Scalar mode:  $M = (\text{number of adoubles}) * 2 + M_p$

\* Vector mode:  $M = (\text{number of adoubles}) * (\text{NUMBER\_DIRECTIONS} + 1) + M_p$

where the storage size of all non `adouble` based variables is described by  $M_p$ .

- Compile time

As discussed in the previous sections, the traceless forward mode of the ADOL-C package is implemented as inline intended version. Using this approach results in a higher source code size, since every operation involving at least one `adouble` stands for the operation itself as well as for the corresponding derivative code after the inlining process. Therefore, the compilation time needed for the traceless version may be higher than that of the trace based code.

- Code Size

A second drawback and result of the code inlining is the increase of code sizes for the binaries. The increase factor compared to the corresponding trace based program is difficult to quantify as it is task dependent. Practical results have shown that values between 1.0 and 2.0 can be expected. Factors higher than 2.0 are possible too and even values below 1.0 have been observed.

## 8 Traceless Differentiation in ADOL-C using Cuda

One major draw-back of the tracefull version of ADOL-C is the fact that several function evaluations are needed for calculating the Jacobian at  $M$  points. More precisely, to calculate the Jacobian of  $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$  in  $M$  points,  $M$  function evaluations are needed for the tracefull scalar mode and even  $M * n$  for the traceless scalar mode. Depending on the number of parallel threads this can result in a long runtime. To achieve a better performance one can use parallelisation techniques as the same operations are performed during a function evaluation. One possibility is to use GPUs since they are optimized for data parallel computation. Starting with version 2.3.0 ADOL-C now features a traceless forward mode for computing first order derivatives in scalar mode on GPUs using the general purpose parallel computing architecture Cuda.

The idea is to include parallel code that executes in many GPU threads across processing elements. This can be done by using kernels, that is functions which are executed on GPU as an array of threads in parallel. In general all threads execute the same code. They are grouped into blocks which are then grouped into grids. A kernel is executed as a grid of blocks of threads. For more details see, e.g. the NVIDIA CUDA C Programming Guide which can be downloaded from the web page [www.nvidia.com](http://www.nvidia.com). To solve the problem of calculating the Jacobian of  $F$  at  $M$  points it is possible to let each thread perform a function evaluation and thus the computation of derivatives for one direction at one point. The advantage is that the function is evaluated in different points in parallel which can result in a faster wallclock runtime.

The following subsection describes the source code modifications required to use the traceless forward mode of ADOL-C with Cuda.

**<https://tutorcs.com>**

### 8.1 Modifying the source code

Let us again consider the coordinate transformation from Cartesian to spherical polar coordinates given by the function  $F : \mathbb{R}^3 \rightarrow \mathbb{R}^3$ ,  $y = F(x)$ , with

$$y_1 = \sqrt{x_1^2 + x_2^2 + x_3^2}, \quad y_2 = \arctan\left(\sqrt{x_1^2 + x_2^2}/x_3\right), \quad y_3 = \arctan(x_2/x_1),$$

as an example. We now calculate the Jacobian at  $M = 1024$  different points. The source code for one point is shown in Figure 6. This example has no real application but can still be used to show the combination of the traceless version of ADOL-C and Cuda.

For the use of this mode a Cuda toolkit, which is suitable for the grafic card used, has to be installed. Furthermore, it is important to check that the graphic card used supports double precision (for details see e.g. NVIDIA CUDA C Programming Guide). Otherwise the data type employed inside of the `adouble` class has to be adapted to `float`. To use the traceless forward mode with Cuda, one has to include the header files `adoublecuda.h`

# 程序代写代做 CS编程辅导

68

8 TRACELESS FORWARD DIFFERENTIATION IN ADOL-C USING CUDA

and `cuda.h`. The operators for this are introduced by the definition of the class `adouble` and the overloaded operators. As in the other versions all derivative calculations are needed for the use of Cuda.

One possibility above is the following. First of all three `double` arrays are needed for the values of independent variables, `y` for the dependent variables and `deriv` for the values of derivatives. The independent variables have to be initialised, therefore the points at which the function should be evaluated are saved in a row in the same array of length  $3 * M$ . For the computation on GPUs one also has to allocate memory on this device. Using the syntax in Cuda one can allocate an array of length  $3 * M$  (number of independent variables times number of points) for the independent variables as follows:

```
double * devx;  
cudaMalloc((void**)&devx, 3*M*sizeof(double));
```

## WeChat: cstutorcs Assignment Project Exam Help

The arrays for the dependent variables and the values of the Jacobian matrices are allocated in the same way. Then the values of the independent variables have to be copied to the GPU using the following command:

```
cudaMemcpy(devx, x, sizeof(double)*3*M, cudaMemcpyHostToDevice);
```

QQ: 749389476

The argument `cudaMemcpyHostToDevice` indicates that the values are copied from the host to the GPU. In this case the values stored in `x` are copied to `devx`.

Now all required information has been transferred to the GPU. The changes in the source code made so far are summarized in Figure 9.

<https://tutores.com>

In the next step the user has to specify how many blocks and threads per block will be needed for the function evaluations. In the present example this is done by the call of the function `kernellaunch`, see Figure 10. In this case the blocks are two dimensional: the x-dimension is determined by the number of points  $M$  at which the Jacobian matrix has to be calculated while the y-dimension is given by the number of independent variables, i.e., 3. Since a block cannot contain more than 1024 threads, the x-dimension in the example is 64 instead of  $M = 1024$ . Therefore  $1024/64 = 16$  blocks are needed. The described division into blocks is reasonable as each thread has to perform a function evaluation for one point and one direction, hence  $M * 3$  threads are needed where 3 denotes the number of independent variables corresponding to the number of directions needed for the computation of the Jacobian in one point.

We can now perform the function evaluations together with the calculation of the Jacobians. The corresponding code is illustrated in Figure 11. Since the function evaluations should be performed on a GPU a kernel function is needed which is defined by using the `__ global__` declaration specifier (see Figure 11). Then each thread executes the operations that are defined in the kernel.

# 程序代写代做 CS编程辅导

## 8.1 Modifying the source code

69

```
#include <iost  
#include <cu  
#include <ado  
using namespace std;
```



```
int main() {  
    int M=1024;  
    double* deriv = new double[9*M];  
    double* y = new double[3*M];  
    double* x = new double[3*M];  
  
    // Initialize  $x_i$   
    for (int k=0; k<M; ++k){  
        for (int i=0, i<3, ++i)  
            x[k*3+i] = i + 1/(k+1);}
```

// Allocate array for independent and dependent variables  
// and Jacobian matrices on GPU

double \* devx;  
cudaMalloc((void\*\*)&devx, 3\*M\*sizeof(double));  
double \* devy;  
cudaMalloc((void\*\*)&devy, 3\*M\*sizeof(double));  
double \* devderiv;  
cudaMalloc((void\*\*)&devderiv, 3\*3\*M\*sizeof(double));

// Copy values of independent variables from host to GPU  
cudaMemcpy(devx, x, sizeof(double)\*3\*M, cudaMemcpyHostToDevice);  
  
// Call function to specify amount of blocks and threads to be used  
kernellaunch(devx, devy, devderiv, M);

// Copy values of dependent variables and Jacobian matrices from GPU to host  
cudaMemcpy(y, devy, sizeof(double)\*3\*M, cudaMemcpyDeviceToHost);  
cudaMemcpy(deriv, devderiv, sizeof(double)\*3\*3\*M, cudaMemcpyDeviceToHost);

}

Figure 9: Example for traceless scalar forward mode with Cuda

In this example each thread is assigned the task of calculating the derivatives in one point with respect to one independent variable. Therefore some indices are needed for the

```

cudaError_t tracelessForward( double* inx, double* outy, double* outeriv, int M) {
    // Create
    int Blocks;
    // Two d
    dim3 threadsPerBlock((M/Blocks)×3 blocks
    dim3 threadsPerBlock(64,3);

    // Call kernel function with 16 blocks with (M/Blocks)×3 threads per block
    kernel <<< Blocks, threadsPerBlock >>>(inx, outy, outeriv);
    cudaError_t cudaErr = cudaGetLastError();
    return cudaErr;
}

```

**WeChat: cstutorcs**

## Assignment Project Exam Help

Figure 10: Example for traceless scalar forward mode with Cuda

implementation. Each thread has a unique thread ID in a block consisting of an x and an y-dimension. The blocks have an ID as well. In the example the following indices are used.

- **index = threadIdx.x** denotes the x-dimension of a thread (ranges from 0 to 63 in the example)
- **index1 = threadIdx.y** denotes the y-dimension of a thread (ranges from 0 to 2 in the example)
- **index2 = blockIdx.x** denotes the block index (ranges from 0 to 15 in the example)
- **index3 = blockDim.x** denotes the x-dimension of a block (always 64 in the example)
- **index4 = blockDim.x\*blockDim.y** denotes the size of a block (always  $64 * 3 = 192$  in the example)

For the calculation of derivatives the function evaluation has to be performed with **adoubles**. Therefore the dependent and the independent variables have to be declared as **adoubles** as in the other versions of ADOL-C (see, e.g., Section 7). Similar to the traceless version without Cuda the namespace **adl1c** is used. Now each thread has to read out the point at which the function evaluation is then performed. This is determined by the **blockIndex** and the x-dimension of the thread, therefore the independent variables in a thread have the values

$$x[i] = \text{inx}[\text{index2} * \text{index4} + \text{index3} * 3 + i];$$

# 程序代写代做 CS编程辅导

8.1 Modifying the source code

71

```
--global__x, double* outy, double* outderiv) {  
    const int index = i * blockDim.x + threadIdx.x;  
    const int index1 = index * blockDim.y + blockIdx.y;  
    const int index2 = index1 * blockDim.z + threadIdx.z;  
    const int index3 = index2 * blockDim.y + blockIdx.y;  
  
    // Declare dependent and independent variables as adoubles  
    adtlc::adouble y[3], x[3];  
    // Read out point for function evaluation  
    for(int i=0; i < 3; i++)  
        x[i]=inx[index2*index4+index*3+i];  
    // Set direction for calculation of derivatives  
    x[index1].setADValue(1);  
  
    // Function evaluation  
    y[0] = sqrt(x[0]*x[0]+x[1]*x[1]+x[2]*x[2]);  
    y[1] = atan(sqrt(x[0]*x[0]+x[1]*x[1])/x[2]);  
    y[2] = atan(x[1]/x[0]);  
  
    for(int i=0; i < 3; i++)  
        outy[(index2*index3+index)*3+i]=y[i].getValue();  
    for(int i=0; i < 3; i++)  
        outderiv[(index2*index4+index*3+index1)*3+i]=y[i].getADValue();  
}
```

WeChat: cstutorcs

Assignment Project Exam Help

Email: tutorcs@163.com

QQ: 749389476

<https://tutorcs.com>

Figure 11: Example for traceless scalar forward mode with Cuda

for  $i=0,1,2$  where  $\text{inx}$  corresponds to the vector  $\text{devx}$ . The direction for the derivatives is given by  $\text{index1}$ .

```
x[index1].setADValue(1);
```

The functions for setting and getting the value and the derivative value of an **adouble** are the same as in the traceless version of ADOL-C for first order derivatives (see Section 7). The function evaluation is then performed with **adouble**  $x$  on GPU and the results are saved in **adouble**  $y$ . The function evaluation itself remains unchanged (see Figure 11). Then we store the values of the function evaluations in each point in a row in one array:

```
outy[(index2*index3+index)*3+i]=y[i].getValue();
```

# 程序代写代做 CS编程辅导

72

for  $i=0,1,2$ . The  are stored column by column in a row:



```
-index*3+index1)*3+i]=y[i].getADValue();
```

for  $i=0,1,2$ .

Now there is  The values calculated on the GPU have to be copied back to host. For the dependent variables in the example this can be done by the following call:

```
cudaMemcpy(y, dev_y, sizeof(double)*S*M, cudaMemcpyDeviceToHost);
```

see Figure 9. The argument `cudaMemcpyDeviceToHost` determines that the values are copied from GPU to host.

# Assignment Project Exam Help

## 8.2 Compiling and Linking the Source Code

After incorporating the required changes, one has to compile the source code and link the object files to get the executable. For the compilation of a Cuda file the nvcc compiler is needed. The compile sequence should be similar to the following example:

nvcc -arch=sm\_20 -o traceless\_cuda traceless\_cuda.cu

The compiler option `-arch=sm_20` specifies the compute capability that is assumed, in this case one that supports double precision.

The mentioned source code `traceless_cuda.cu` illustrating the use of the forward traceless scalar mode with Cuda and a further example `liborgpu.cu` can be found in the directory examples. The second example is an adaption of the OpenMP example to the traceless version with Cuda.

## 9 Installing and Using ADQL-C

## 9.1 Generating the ADOL-C Library

The currently built system is best summarized by the ubiquitous gnu install triplet:

`configure = make = make install`

Executing this three steps from the package base directory </SOME PATH/ado1c-2.7.0-trunk> will compile the static and the dynamic ADQL-C

# 程序代写代做 CS编程辅导

library with default installation directory `include` will contain the subdirectory `include` which will contain the header files that may be included by the user program, the 32-bit compiled library and the subdirectory `lib64` which will contain the 64-bit compiled library. Depending on the compiler only one of `lib` or `lib64` may be created.

Before doing this it is necessary to modify the header file `usrparms.h` in order to tailor the ADOL-C package to the needs in the particular system environment as discussed in Section 2.2. The configure procedure which creates the necessary `Makefiles` can be customized by use of some switches. Available options and their meaning can be obtained by executing `./configure --help` from the package base directory.

All object files and other intermediately generated files can be removed by the call `make clean`. Uninstalling ADOL-C by executing `make uninstall` is only reasonable after a previous called `make install` and will remove all installed package files but will leave the created directories behind.

The sparse drivers are included in the ADOL-C libraries if the `./configure` command is executed with the option `--enable-sparse`. The ColPack library available at <http://cscapes.cs.purdue.edu/coloringpage/software.htm> is required to compute the sparse structures, and is searched for in all the default locations. In case the library and its headers are installed in a nonstandard path this may be specified with the `--with-colpack=PATH` option. It is assumed that the library and its header files have the following directory structure: `PATH/include` contains all the header files, `PATH/lib` contains the 32-bit compiled library and `PATH/lib64` contains the 64-bit compiled library. Depending on the compiler used to compile ADOL-C one of these libraries will be used for linking.

Email: [tutorcs@163.com](mailto:tutorcs@163.com)

QQ: 749389476

<https://tutorcs.com>

The option `--disable-stdczero` turns off the initialization in the `adouble` default constructor. This will improve efficiency but requires that there be no implicit array initialization in the code, see Section 2.3.

Support for MPI and the AdjoinableMPI API (see <http://www.mcs.anl.gov/~utke/AdjoinableMPI/AdjoinableMPIIndex.html>) may be enabled using the option `--enable-ampi`. This requires the presence of MPI compiler wrappers `mpicc` and `mpicxx` in the `$PATH` and the AdjoinableMPI libraries in the standard locations. If MPI is installed in a nonstandard path one may specify this using the `--with-mpi=PATH` option. Similarly if the AdjoinableMPI libraries are in a nonstandard path this may be specified using the `--with-ampi=PATH` option. When MPI and AdjoinableMPI support is compiled into ADOL-C the generated library will be unsuitable for linking with non-MPI programs and is called `libadolc_ampi`. Therefore the user programs should use the flag `-ladolc_ampi` instead of `-adolc` in this case. An advanced user may in fact specify whatever name the resulting ADOL-C library should have using the `--with-soname=SONAME` option.

## 9.2 Compiling the Example Programs

The installation in Section 9.1 also provides the Makefiles to compile the example programs in the directory `<adolc-2.7.0-trunk>/ADOL-C/examples` and the additional examples in `<adolc-2.7.0-trunk>/ADOL-C/examples/additional_examples`. However, one has to run the `./configure` command with appropriate options for the compilation of examples. Available options are:

- `--enable-docexamples` build all examples discussed in this manual  
(compare Section 10)
- `--enable-addexamples` build all additional examples  
(See file README in the various subdirectories)

Just calling `make` from the packages base directory generates all configured examples and the library if necessary. Compiling from subdirectory `examples` or one of its subfolders is possible too. At least one kind of the ADOL-C library (static or shared) must have been built previously in that case. Hence, building the library is always the first step.

For Compiling the library and the documented examples on Windows using Visual Studio please refer to the `<Readme_VC++.txt>` files in the `<windows/>`, `<ThirdParty/ColPack/>` and `<ADOL-C/examples/>` subdirectories.

## 9.3 Description of Important Header Files

The application of the facilities of ADOL-C requires the user source code (program or module) to include appropriate header files where the desired data types and routines are prototyped. The new hierarchy of header files enables the user to take one of two possible ways to access the right interfaces. The first and easy way is recommended to beginners: As indicated in Table 5 the provided *global* header file `<adolc/adolc.h>` can be included by any user code to support all capabilities of ADOL-C depending on the particular programming language of the source.

<code>&lt;adolc/adolc.h&gt;</code>	<ul style="list-style-type: none"> <li>→ global header file available for easy use of ADOL-C;</li> <li>• includes all ADOL-C header files depending on whether the users source is C++ or C code.</li> </ul>
<code>&lt;adolc/internal/usrparms.h&gt;</code>	<ul style="list-style-type: none"> <li>→ user customization of ADOL-C package (see Section 2.2);</li> <li>• after a change of user options the ADOL-C library <code>libadolc.*</code> has to be rebuilt (see Section 9.1);</li> <li>• is included by all ADOL-C header files and thus by all user programs.</li> </ul>

Table 5: Global header files

The second way is meant for the more advanced ADOL-C user: Some source code includes only those interfaces used by the particular application. The respectively needed

# 程序代写代做 CS编程辅导

header files are i  
dences between t  
in order to main  
Table 6 and Tab



the manual. Existing application determined depen  
routines are realized by automatic includes of headers  
header files important to the user are described in the

<adolc/adouble.h>	des the interface to the basic active scalar data type of L-C: class <code>adouble</code> (see Section 1);
<adolc/taputil.h>	les functions to start/stop the tracing of active sections (see Section 1.3) as well as utilities to obtain tape statistics (see Section 2.1); • is included by the header < <code>adolc/adouble.h</code> >.

WeChat: cstutorcs

Table 6: Important header files: tracing/taping

## Assignment Project Exam Help

<adolc/interfaces.h>	→ provides interfaces to the forward and reverse routines as basic version of derivative evaluation (see Section 5); • comprises C++, C, and Fortran-callable versions; • includes the header < <code>adolc/sparse/sparsedrivers.h</code> >; • is included by the header < <code>adolc/drivers/odedrivers.h</code> >.
<adolc/drivers.h>	→ provides “easy to use” drivers for solving optimization prob lems and nonlinear equations (see Section 3.1); • comprises C and Fortran-callable versions.
<adolc/sparse/ sparsedrivers.h>	→ provides the “easy to use” sparse drivers to exploit the sparsity structure of Jacobians (see Section 3.3); → provides interfaces to C++-callable versions of forward and re verse routines propagating bit patterns (see Section 5.3); • is included by the header < <code>adolc/interfaces.h</code> >.
<adolc/sparse/ sparse_fo_rev.h>	→ provides interfaces to the underlying C-callable versions of for ward and reverse routines propagating bit patterns.
<adolc/drivers/ odedrivers.h>	→ provides “easy to use” drivers for numerical solution of ordi nary differential equations (see Section 3.2); • comprises C++, C, and Fortran-callable versions; • includes the header < <code>adolc/interfaces.h</code> >.
<adolc/drivers/ taylor.h>	→ provides “easy to use” drivers for evaluation of higher order derivative tensors (see Section 3.4) and inverse/implicit func tion differentiation (see Section 3.5); • comprises C++ and C-callable versions.
<adolc/adalloc.h>	→ provides C++ and C functions for allocation of vectors, ma trices and three dimensional arrays of doubles.

Table 7: Important header files: evaluation of derivatives

## 9.4 Compiling and Linking C/C++ Programs

To compile a C/C++ program or single module using ADOL-C data types and routines one has to ensure that all necessary header files according to Section 9.3 are included. All

# 程序代写代做 CS编程辅导

76

9 INSTALLING AND USING ADOL-C

modules involving `adouble` have to be compiled as C++. Modules that make use of a `pr` type to evaluate derivatives can either be programmed in ANSI-C (which is the default) or via C++ interfaces. Depending on the chosen programming language, the user has to provide the right ADOL-C prototypes. For linking the resulting object file `libadolc.*` must be used (see Section 9.1).



## 9.5 Interfacing with Python and R

For generating the SWIG based interface to Python and R programming languages the prerequisite is an installation of the SWIG tool (at least newer than version 3.0.8). Furthermore, for installing and using the R interface please refer to the documentation and software at <https://github.com/sriharikrishna/radolc>. Installation of the Python interface can be done by changing into the ADOL-C/swig subdirectory of the ADOL-C source code and calling the command

Assignment Project Exam Help

```
python setup.py install --user
```

Email: tutorcs@163.com

This will internally cause the compilation and installation of the ADOL-C library and header files using default configuration options as described in Section 9.1.

For a description on the working of these interfaces refer to the following article.

K. Kulshreshtha, S.H.K. Narayanan, J. Bessac and K. MacIntyre: Efficient computation of derivatives for solving optimization problems in R and Python using SWIG generated interfaces to ADOL-C. *Optimisation Methods and Software*, 33(4-6), pp1173-1191. 2018.

<https://tutorcs.com>

## 9.6 Adding Quadratures as Special Functions

Suppose an integral

$$f(x) = \int_0^x g(t)dt$$

is evaluated numerically by a user-supplied function

```
double myintegral(double& x);
```

Similarly, let us suppose that the integrand itself is evaluated by a user-supplied block of C code `integrand`, which computes a variable with the fixed name `val` from a variable with the fixed name `arg`. In many cases of interest, `integrand` will simply be of the form

```
{ val = expression(arg) } .
```

# 程序代写代做 CS编程辅导

77

In general, the function may be preceded by several intermediate calculations, possibly involving local or global variables of type `adouble`, but no external or static variables of that type. However, if the user provides a block of code that does not involve local or global variables of type `double` or `int`, provided they do not depend on the value of `arg`. The variables `arg` and `val` are declared automatically; a header line.



Now the function `myintegral` is overloaded for `adouble` arguments and thus included in the library of elementary functions by the following modifications:

1. At the end of the file `<adouble.cpp>`, include the full code defining  
`double myintegral(double& x)` and add the line

```
extend_quad(myintegral, integrand);
```

This macro is extended to the definition of `adouble myintegral(adouble& arg)`. Then remake the library `libadlc.*` (see Section 9.1).

2. In the definition of the class `ADOLC_DLL_EXPORT adouble` in `<adolc/adouble.h>`, add the statement

**Email: tutorcs@163.com**  
friend `adouble myintegral(adouble&)`.

In the first modification, `myintegral` represents the name of the `double` function, whereas `integrand` represents the actual block of C code.

**QQ: 749389476**

For example, in case of the inverse hyperbolic cosine, we have `myintegral = acosh`. Then `integrand` can be written as `{ val = sqrt(arg*arg-1); }` so that the line

**https://tutorcs.com**  
`extend_quad(acosh, val = sqrt(arg*arg-1));`

can be added to the file `<adouble.cpp>`. A mathematically equivalent but longer representation of `integrand` is

```
{ adouble temp = arg;  
    temp = temp*temp;  
    val = sqrt(temp-1); }
```

The code block `integrand` may call on any elementary function that has already been defined in file `<adouble.cpp>`, so that one may also introduce iterated integrals.

## 10 Example Codes

The following listings are all simplified versions of codes that are contained in the example subdirectory `<adolc-2.7.0-trunk>/ADOL-C/examples` of ADOL-C. In particular, we have left out timings, which are included in the complete codes.

# 程序代写代做 CS编程辅导

## 10.1 Speelpenning (speelpenning.cpp)

The first example shows how to calculate the gradient and the Hessian of the function



$$f(x) = \prod_{i=0}^{n-1} x_i$$

using the appropriate `gradient()` and `hessian()`.

```
#include <adolc/adouble.h>           // use of active doubles and taping
#include <adolc/drivers/drivers.h>     // use of "Easy to Use" drivers
// gradient(.) and hessian(.)
#include <adolc/taping.h>             // use of taping
...
void main() {
    int n,i,j;
    size_t tape_stats[STAT_SIZE];
    cout << "SPEELPENNINGS PRODUCT (ADOL-C Documented Example) \n";
    cout << "number of independent variables = " << n << endl;
    cin >> n;
    double* xp = new double[n];
    double  yp = 0.0;
    adouble* x = new adouble[n];
    adouble  y = 1;
    for(i=0;i<n;i++)
        xp[i] = (i+1.0)/(2.0+i); // some initialization
    trace_on(1); // tag =1, keep=0 by default
    for(i=0;i<n;i++) {
        x[i] <= xp[i]; y *= x[i];
    }
    y >>= yp;
    delete[] x;
    trace_off();
    tapestats(1,tape_stats); // reading of tape statistics
    cout << "maxlive "<< tape_stats[2] << "\n";
    ... // ..... print other tape stats
    double* g = new double[n];
    gradient(1,n,xp,g); // gradient evaluation
    double** H=(double**)malloc(n*sizeof(double*));
    for(i=0;i<n;i++)
        H[i]=(double*)malloc((i+1)*sizeof(double));
    hessian(1,n,xp,H); // H equals (n-1)g since g is
    double errg = 0; // homogeneous of degree n-1.
    double errh = 0;
```

# 程序代写代做 CS编程辅导

10.2 Power Example (powexam.cpp)

79

```
for(i=0;i<n;i+  
    errg += fabs  
for(i=0;i<n;i+  
    for(j=0;j<n;  
        if (i>j) // lower half of hessian  
            errh += xp[j]); } }  
cout << yp-1/( in function \n";  
cout << errg << " error in gradient \n";  
cout << errh << " consistency check \n";  
} // end main
```

WeChat: cstutorcs

10.2 Power Example (powexam.cpp)

The second example function calculates the  $n$ -th power of a real variable  $x$  in  $\log_2 n$  multiplications by recursive halving of the exponent. Since there is only one independent variable, the scalar derivative can be computed by using both forward and reverse, and the results are subsequently compared.

Email: tutorcs@163.com

```
#include <adolc/adolc.h> // use of ALL ADOL-C interfaces
```

```
adouble power(adouble x, int n) {  
adouble z=1; QQ: 749389476  
if (n>0) { // recursion and branches  
    int nh =n/2; // that do not depend on  
    z = power(x,nh); // adoubles are fine !!!!  
    z *= z;  
    if (2*nh != n)  
        z *= x;  
    return z; } // end if  
else {  
    if (n==0) // the local adouble z dies  
        return z; // as it goes out of scope.  
    else  
        return 1/power(x,-n); } // end else  
} // end power
```

https://tutorcs.com

The function `power` above was obtained from the original undifferentiated version by simply changing the type of all doubles including the return variable to `adoubles`. The new version can now be called from within any active section, as in the following main program.

```
#include ... // as above  
int main() {
```

# 程序代写代做 CS编程辅导

10 EXAMPE CODES

```

int i,n,tag=1;
cout <<"COMPUTER EXAMPLE (ADOL-C Documented Example)\n\n";
cout<<"monomial degree: ";
cin >> n;

double* Y[1];
*Y = new double[n+1];
double* X[1];
*X = new double[n+4];
X[0][0] = 0.5;
X[0][1] = 1.0;
// first derivative = 1. coefficient
for(i=0;i<n+2;i++)
    X[0][i+2]=0;
double* Z[1];
*Z = new double[n+1];
adouble y,x;
// declare active variables
// beginning of active section
// tag = 1 and keep = 0
x <<= X[0][0];
// only one independent variable
y = power(x,n);
// actual function call
y >>= Y[0][0];
// only one dependent adouble
trace_off();
// end global adouble has died
// end of active section
double u[1];
u[0]=1;
// for reverse call
for(i=0;i<n+2;i++)
    forward(tag,1,1,i,i+1,X,Y); // note that step = i+1 in call
    // evaluate the i-the derivative
    if (i==0)
        cout << Y[0][i] << " - " << y.value() << " = " << Y[0][i]-y.value()
        << " (should be 0)\n";
    else
        cout << Y[0][i] << " - " << Z[0][i] << " = " << Y[0][i]-Z[0][i]
        << " (should be 0)\n";
    reverse(tag,1,1,i,u,Z); // evaluate the (i+1)-st derivative
    Z[0][i+1]=Z[0][i]/(i+1); } // scale derivative to Taylorcoeff.
return 1;
} // end main

```

Since this example has only one independent and one dependent variable, forward and reverse have the same complexity and calculate the same scalar derivatives, albeit with a slightly different scaling. By replacing the function `power` with any other univariate test function, one can check that forward and reverse are at least consistent. In the following example the number of independents is much larger than the number of dependents, which makes the

WeChat: cstutores  
Assignment Project Exam Help  
Email: tutorcs@163.com  
QQ: 749389476  
<https://tutorcs.com>



# 程序代写代做 CS编程辅导

10.3 Determinant Example (*detexam.cpp*)

81

reverse mode pre



## 10.3 Determinant Example (*detexam.cpp*)

Now let us consider the computation of a determinant. This is an expensive calculation, namely, the evaluation of a determinant by summing up all terms along rows. The gradient of the determinant with respect to the matrix elements is simply the adjoint, i.e. the matrix of cofactors. Hence the correctness of the numerical result is easily checked by matrix-vector multiplication. The example illustrates the use of *adouble* arrays and pointers.

WeChat: cstutorcs

```
#include <adolc/adouble.h>           // use of active doubles and taping
#include <adolc/interfaces.h>          // use of basic forward/reverse
adouble** A;                         // interfaces of ADOL-C
int i,n;                             // A is an n x n matrix
int p;                                // k <= n is the order
adouble det(int k, int m) {            // of the sub-matrix
    if (m == 0) return 1.0;             // its column indices
    else {                            // are encoded in m
        adouble* pt = A[k-1];
        adouble t = zero;              // zero is predefined
        int s, p = 1;
        if (k%2) s = 1, else s = -1;
        for(i=0;i<n;i++) {
            int p1 = 2*p;
            if (m%p1 >= p) {
                if (m == p) t =
                    if (s>0) t += *pt; else t -= *pt;
            }
            else {
                if (s>0)
                    t += *pt*det(k-1,m-p); // recursive call to det
                else
                    t -= *pt*det(k-1,m-p); } // recursive call to det
            s = -s;
            ++pt;
            p = p1;
        }
        return t;
    }
}
```

Assignment Project Exam Help

Email: tutorcs@163.com

QQ: 749389476

<https://tutorcs.com>

As one can see, the overloading mechanism has no problem with pointers and looks exactly the same as the original undifferentiated function except for the change of type from *double* to *adouble*. If the type of the temporary *t* or the pointer *pt* had not been changed, a compile time error would have resulted. Now consider a corresponding calling program.

```
#include ... // as above
int main() {
    int i,j, m=1,t
    cout << "COMPUJANTS (ADOL-C Documented Example)\n\n";
    cout << "order"; // select matrix size
    cin >> n;
    A = new adouble[n][n];
    trace_on(tag,keep);
    double detout=0.0, diag = 1.0;
    for(i=0;i<n;i++) {
        m *=2;
        A[i] = new adouble[n];
        adouble* pt = A[i];
        for(j=0;j<n;j++)
            A[i][j] <<= 1.0+1; // makes all elements of A independent
        diag += A[i][i].value(); // value() converts to double
        A[i][i] += 1.0;
    }
    det(n,m-1) >= detout; // actual function call
    printf("\n %1.1f %1.1f (%should be 0)\n",detout,detout-diag);
    trace_off();
    double u[1];
    u[0] = 1.0;
    double* B = new double[n*n];
    reverse(tag,1,n*n,1,u,B);
    cout <<"\n first base? : ";
    for (i=0;i<n;i++) {
        adouble sum = 0;
        for (j=0;j<n;j++) // the matrix A times the first n
            sum += A[i][j]*B[j]; // components of the gradient B
        cout<<sum.value()<<" "; } // must be a Cartesian basis vector
    return 1;
} // end main
```

The variable `diag` should be mathematically equal to the determinant, because the matrix `A` is defined as a rank 1 perturbation of the identity.

## 10.4 Ordinary Differential Equation Example (`odexam.cpp`)

Here, we consider a nonlinear ordinary differential equation that is a slight modification of the Robertson test problem given in Hairer and Wanner's book on the numerical solution of ODEs [11]. The following source code computes the corresponding values of  $y' \in \mathbb{R}^3$ :

# 程序代写代做 CS编程辅导

10.4 Ordinary Differential Equation Example (*odexam.cpp*)

83

```
#include <adol/cpp/tapdir.h>           // use of active doubles and taping
#include <adol/cpp/tapdir.h>           // use of "Easy To use" ODE drivers
#include <adol/cpp/tapdir.h>           // use of ADOL-C allocation utilities

void tracerhs(double* py, double* pyprime) {           // this time we left the parameters
    adouble y[3];                                     // passive and use the vector types
    adouble yprime[3];
    trace_on(tag);
    for (int i=0; i<3; i++)
        y[i] <= py[i];                                // initialize and mark independents
    yprime[0] = -sin(y[2]) + 1e8*y[2]*(S-1)y[0];
    yprime[1] = -10*y[0] + 3e7*y[2]*(1-y[1]);
    yprime[2] = -yprime[0] - yprime[1];
    yprime >>= pyprime;                            // mark and pass dependents
    trace_off(tag);                                 // end tracerhs
}
```

The Jacobian of the right-hand side has large negative eigenvalues, which make the ODE quite stiff. We have added some numerically boring transcendents to make the differentiation more interesting. The following main program uses `forode` to calculate the Taylor series defined by the ODE at the given point  $y_0$  and `reverse` as well as `accode` to compute the Jacobians of the coefficient vectors with respect to  $x_0$ .

```
#include .....                                // as above
int main() {
int i,j,deg;
int n=3;
double py[3];
double pyp[3];
cout << "MODIFIED ROBERTSON TEST PROBLEM (ADOL-C Documented Example)\n";
cout << "degree of Taylor series =?\n";
cin >> deg;
double **X;
X=(double**)malloc(n*sizeof(double*));
for(i=0;i<n;i++)
    X[i]=(double*)malloc((deg+1)*sizeof(double));
double*** Z=new double**[n];
double*** B=new double**[n];
short** nz = new short*[n];
for(i=0;i<n;i++) {
    Z[i]=new double*[n];
    B[i]=new double*[n];
    for(j=0;j<n;j++) {
```

# 程序代写代做 CS编程辅导

REFERENCES

```

Z[i][j]=ne
B[i][j]=ne
}
for(i=0;i<n;i+
    py[i] = (i =
    X[i][0] = py
    nz[i] = new
tracerhs(1,py,pyp);
forode(1,n,deg,X);
reverse(1,n,n,deg-1,Z,nz);
accode(n,deg-1,Z,B,nz);
cout << "nonzero pattern:\n";
for(i=0;i<n;i++) {
    for(j=0;j<n;j++)
        cout << nz[i][j]<"\t";
    cout <<"\n"; }
return 1;
} // end main

```

**WeChat: cstutorcs**

**Email: tutorcs@163.com**

The pattern `nz` returned by `accode` is


**QQ: 749389476**

The original pattern `nz` returned by `reverse` is the same except that the negative entry  $-1$  was zero.

**<https://tutorcs.com>**

## Acknowledgements

Parts of the ADOL-C source were developed by Andreas Kowarz, Hristo Mitev, Sebastian Schlenkrich, and Olaf Vogel. We are also indebted to George Corliss, Tom Epperly, Bruce Christianson, David Gay, David Juedes, Brad Karp, Koichi Kubota, Bob Olson, Marcela Rosemblun, Dima Shiriaev, Jay Srinivasan, Chuck Tyner, Jean Utke, and Duane Yoder for helping in various ways with the development and documentation of ADOL-C.

## References

- [1] Christian H. Bischof, Peyvand M. Khademi, Ali Bouaricha and Alan Carle. *Efficient computation of gradients and Jacobians by dynamic exploitation of sparsity in automatic differentiation*. Optimization Methods and Software 7(1):1-39, 1996.

# 程序代写代做 CS编程辅导

## REFERENCES

85

- [2] Bruce Chris umulation and accurate rounding error estimates for Taylor serie Methods and Software 1:81–94, 1992.
- [3] Assefaw Gel anne, and Alex Pothen. What color is your Jacobian? Graph color SIAM Review 47(4):629–705, 2005.
- [4] Assefaw Ge hen, Arijit Tarafdar and Andrea Walther. Efficient Computation An Experimental Study using ADOL-C. Tech. Rep. (2006). To appear in INFORMS Journal on Computing.
- [5] Assefaw Gebremedhin, Alex Pothen, and Andrea Walther. Exploiting Sparsity in Jacobian Computation via Coloring and Automatic Differentiation: a Case Study in a Simulated Moving Bed Process. In Chr. Bischof et al., eds., Proceedings AD 2008 conference, LNCSE 64, pp. 327 – 338, Springer (2008).
- [6] Assefaw Gebremedhin, Arijit Tarafdar, Fredrik Marne, and Alex Pothen. New Cycle and Star Coloring Algorithms with Applications to Hessian Computation. SIAM Journal on Scientific Computing 29(3):1042–1072, 2007.
- [7] Andreas Griewank. On stable piecewise linearization and generalized algorithmic differentiation. Optimization Methods and Software 28(6):1139–1178, 2013.
- [8] Andreas Griewank and Andrea Walther: Evaluating Derivatives, Principles and Techniques of Algorithmic Differentiation, Second edition. SIAM, 2008.
- [9] Andreas Griewank, Jean Utke, and Andrea Walther. Evaluating higher derivative tensors by forward propagation of univariate Taylor series. Mathematics of Computation, 69:1117–1131, 2000.
- [10] Andreas Griewank and Andrea Walther. Revolve: An Implementation of Checkpointing for the Reverse or Adjoint Mode of Computational Differentiation, ACM Transaction on Mathematical Software 26:19–45, 2000.
- [11] Ernst Hairer and Gerhard Wanner. Solving Ordinary Differential Equations II. Springer-Verlag, Berlin, 1991.
- [12] Donald E. Knuth. The Art of Computer Programming. Second edition. Addison-Wesley, Reading, 1973.
- [13] Klaus Röbenack. Computation of Lie Derivatives of Tensor Fields Required for Non-linear Controller and Observer Design Employing Automatic Differentiation. PAMM, 5(1): 181-184, 2005.
- [14] Klaus Röbenack, Jan Winkler and Siqian Wang. LIEDRIVERS - A Toolbox for the Efficient Computation of Lie Derivatives Based on the Object-Oriented Algorithmic Differentiation Package ADOL-C. Proc. of the 4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools, F. E. Cellier and D. Broman

# 程序代写代做 CS编程辅导

REFERENCES

and P. Fritz  
Proceedings



itors, volume 56 of Linköping Electronic Conference  
2011.

- [15] Andrea Wal  
action on M  
rse Hessians with Automatic Differentiation. Trans-  
, 34(1), Artikel 3 (2008).

WeChat: cstutorcs

Assignment Project Exam Help

Email: tutorcs@163.com

QQ: 749389476

<https://tutorcs.com>