1 Why do we need adjoint routines?

Numerical libraries are extremely beneficial when you need to implement a model. They allow developers to concentrate on high level modelling rather than spend time on developing low level routines, such as linear equation solvers, optimizers etc. What happens if you need to differentiate a code that contains library routine calls? This is, for example, the case if you want to calibrate the parameters of your model. The algorithms (optimizers) used for calibration often require derivatives for better performance. If your library does not provide support to compute the derivatives of its routines, the only choice you have is to approximate derivatives using finite differences. Leaving aside the fact that
approximation of derivatives can have poor accuracy, there is another important problem you might encounter using this approach. Below we highlight this problem based on the small example given in Listing 2.

This example sets up data from \( n \) values of the exponential function in the interval 0 to 1.

```c
for (int i = 0; i < m; ++i) {
    x[i] = double(i)/(m-1);
    y[i] = exp(x[i]);
}
```

Routine \texttt{e01baf} is then called to compute a spline interpolant to these data. The spline is then evaluated at some point \( x_0 \) (\texttt{xarg}) using routine \texttt{e02bbf}.

```c
e01baf_ (m, x.data(), y.data(), lamda.data(), c.data(), lck, wrk.data(), lwrk, ifail);
e02bbf_ (lck, lamda.data(), c.data(), xarg, fit, ifail);
```

If we are now interested in the derivative of the spline value at the point \( x_0 \) with respect to the abscissa of the set up data, we could compute this gradient with forward finite differences as shown in the following code.

```c
for (int i = 0; i < m; ++i) {
    x[i] += h;
    e01baf_ (m, x.data(), y.data(), lamda.data(), c.data(), lck, wrk.data(), lwrk, ifail);
    e02bbf_ (lck, lamda.data(), c.data(), xarg, fith, ifail);
    x[i] -= h;
    if (m < 8) {
        printf(" %" NAG_IFMT " %13.8 f \n", i + 1, (fith-fit)/h);
    }
}
```

In order to approximate all entries of the gradient, we have to perturb the \( x \) value of each data point separately then interpolate and evaluate the spline for the resulting data set. Assuming that our data sets consists of \( m \) data points we have to compute and evaluate the spline \( m \) to obtain the desired derivatives. Obviously this approach becomes very time consuming for big \( m \). E.g., if we set \( m = 10000 \) in our example code, we will need roughly 21 seconds to compute the derivatives. This is quite a long time when a single interpolation (\texttt{e01baf}) and evaluation (\texttt{e02bbf}) takes only 0.0021 seconds.

Mark 26.2 of the NAG Library introduced adjoint versions of the Library routines. If a routine for a given input \( x \) computes \( y = F(x) \), the adjoint version of this routine computes for given inputs \( x \) and \( y(1) \)

\[
x(1) = f(1)(x, y(1)) = (f'(x))^T \cdot y(1),
\]
where $f'$ denotes the Jacobian of $f$. For more details please refer to the [NAG AD Library Introduction]. The biggest advantage of using the adjoint version of the NAG Library routines is that it allows you to compute the Jacobian row by row, while as with finite difference you compute the Jacobian column by column. Another advantage is that it allows you to compute the exact derivatives. The computation of the same derivatives using adjoint versions of routines e01baf and e02bbf is shown in Listing[1]. In our example the Jacobian has only one row, hence we need to run our adjoint routines only once to compute the Jacobian, as shown in the code below:

```c
/* E01BA_A1W_F. */
* Adjoint of interpolating function, cubic spline interpolant, one variable */
E01BA_A1W_F(ad_config, m, x.data(), y.data(), lamda.data(), c.data(), lck, wrk.data(), lwrk, ifail);

for (int j = 0; j < lck; ++j) {
    dco::a1w::global_ir->register_output_variable(lamda[j]);
    dco::a1w::global_ir->register_output_variable(c[j]);
}

/* E02BB_A1W_F. */
* Adjoint of evaluation of fitted cubic spline, function only */
E02BB_A1W_F(ad_config, lck, lamda.data(), c.data(), xarg, fit, ifail);
```

You can see we have to call the adjoint versions of e01baf (E01BA_A1W_F) and e02bbf (E02BB_A1W_F) only once. Therefore the computation of the Jacobian takes only 0.2 seconds. We will discuss the additional API calls in later sections. For an adjoint routine it does not matter how many inputs the function has, the derivatives with respect to additional input or even intermediate values are computed for “free”. E.g., the code in Listing[1] also computes the derivatives with respect to spline knots (lamda) and spline coefficients (c) all we have to do is to register these variables:

```c
for (int j = 0; j < lck; ++j) {
    dco::a1w::global_ir->register_output_variable(lamda[j]);
    dco::a1w::global_ir->register_output_variable(c[j]);
}
```
No additional calls of adjoint routines are performed to compute these derivatives. With finite difference we would need to perturb each entry of \( \text{lambda} \) and \( c \) to compute these derivatives.

```
for (int i = 0; i < lck; ++i) {
    lamdah[i] += h;
    e02bbf_(lck, lamdah.data(), ch.data(), xarg, fith, ifail);
    lamdah[i] -= h;
}
ch[i] += h;
```

The use of adjoint versions of NAG Library routines is an advantage if the Jacobian you are computing has much smaller number of rows than columns.

2 Introduction to Algorithmic Differentiation (AD)

To introduce adjoint derivatives and their advantages, we first briefly illustrate the tangent model and a commonly used implementation of this model which approximates by finite differences. For more detailed information on adjoints in general, see [3], and for algorithmic adjoints in particular, see [6] and [4]. For a continuously differentiable function (the primal)

\[
y := f(x) \text{ with } f : \mathbb{R}^n \rightarrow \mathbb{R}^m,
\]

and assuming distinct inputs and outputs, the tangent model is defined as

\[
y^{(1)} := f^{(1)}(x, x^{(1)}) = \nabla_x f(x) \cdot x^{(1)},
\]

with tangents \( x^{(1)} \in \mathbb{R}^n \) and \( y^{(1)} \in \mathbb{R}^m \), and the tangent function \( f^{(1)} : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}^m \). The symbol \( \nabla_x \) stands for the full derivative tensor of \( f \) with respect to \( x \), i.e., the Jacobian. The tangent model calculates a weighted sum of the columns of the Jacobian matrix, which corresponds to a directional derivative of \( f \) in direction \( x^{(1)} \). The tangent model can easily be approximated using finite differences by perturbing the input into a scaled direction \( x^{(1)} \) with appropriate scaling factor \( h \) and calculating the difference quotient

\[
\frac{f(x + hx^{(1)}) - f(x)}{h} \approx \nabla f(x) \cdot x^{(1)}.
\]
Using the tangent model to compute the full Jacobian matrix, we need to calculate the directional derivatives in the direction of the \( n \) Cartesian basis vectors in \( \mathbb{R}^n \). This approach delivers the Jacobian column-by-column. Since it requires the evaluation of \( f \) at the original point \( x \) in addition to the \( n \) perturbed points, it has a computational complexity of \( O(n) \cdot \text{cost}(f) \), which corresponds to \( n \) evaluations of the tangent model. The adjoint model is defined as

\[
\left( x^{(1)}, y^{(1)} \right) := f^{(1)}(x, x^{(1)}, y^{(1)}) = \left( x^{(1)} + \left[ \nabla_x f(x)^T \right] y^{(1)} \right)
\]

with adjoint variables \( x^{(1)} \in \mathbb{R}^n \) and \( y^{(1)} \in \mathbb{R}^m \) (corresponding to respective primal variables \( x \) and \( y \)), and the adjoint function \( f^{(1)} : \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n \times \mathbb{R}^m \) which calculates the product of the transposed Jacobian with \( y^{(1)} \), and which sets \( y^{(1)} \) to zero on output. The calculated weighted sum of rows of the Jacobian matrix corresponds to an adjoint directional derivative of \( f \) in the adjoint direction \( y^{(1)} \). The computation of the full Jacobian matrix requires adjoint directional derivatives in the direction of the \( m \) Cartesian basis vectors in \( \mathbb{R}^m \). This approach delivers the Jacobian row-by-row and has a computational complexity of \( O(m) \cdot \text{cost}(f) \). This is an enormous advantage over the tangent model whenever \( R \cdot m < n \), where \( R \) quantifies the implementation overhead of the adjoint. For example, in many optimization problems, a single scalar objective value \((m = 1)\) is computed from a large number of input model arguments \((n \text{ large})\). There is no finite difference approximation for the adjoint model since computing the adjoints efficiently requires a data flow reversal of the primal. Adjoint code development is a non-trivial task; however, having an adjoint library available can be extremely powerful. As an alternative to handwriting adjoint code, algorithmic differentiation (AD) is a widespread technique also used to generate parts of the NAG AD Library routines. For this purpose, overloading is implemented in C++ by the AD tool dco/c++. See url ??.

### 3 C++ Interface to the NAG AD Library

#### 3.1 General Remarks

The subset of the library containing the adjoint versions of the NAG Library routines is called the NAG AD Library. The content of the NAG AD Library is shown in (url ??). In later releases of the NAG Library it is planned to extend the content of the NAG AD Library by adding more adjoint versions of NAG Library routines, introducing tangent versions of the NAG Library routines and potentially higher order routines.

As shown in the previous examples the interface of the NAG AD Library routines is very similar to that of the original (primal) Fortran Library routine. There are only two differences between these interfaces

1. all floating point data types are replaced with special data types (called active data types).
2. the first argument is always `ad_handle`. Used to configure the routine.

The active data types are used to reference the primal and adjoint variable. The respective components are accessible via functions. Use of active data types instead of intrinsic floating point types makes it possible to keep the number of arguments of the routine small. Otherwise we would have to add a separate variable for each adjoint value. Another advantage of using the active data types is that they allow easy integration of adjoint library routines into NAG’s AD tool, dco. The active data types used by the NAG AD Library are binary compatible to the data types used by dco. So you have the choice to either use the NAG AD Library functions to work with active data types or use the similar functions from dco. Although the NAG AD Library is designed to be compatible with NAG’s AD tool, dco, the use of dco is not essential, meaning that the NAG AD Library can be used on its own or in conjunction with any other AD solution. However, using the NAG AD Library in conjunction with dco provides the best user experience.

NAG AD Library routines provide different computational modes. In the current release (Mark 26.2) only the following two modes are available

1. algorithmic
2. symbolic

In later releases we plan to add more computation modes. Algorithmic mode is available for all routines of the NAG AD Library.

Algorithmic mode means that the derivatives are computed with the help of NAG’s AD tool, dco. Specifically, the routine code was overloaded with the active data types from dco. While in algorithmic mode the adjoint code is automatically generated by dco, in symbolic mode the adjoint code has to be written by hand. The benefit of symbolic adjoints is that it can exploit mathematical properties of a Library routine resulting in a more efficient code. The possible efficiency gains through symbolic adjoints depend on the specific routine. Routines providing huge benefits are the linear and the nonlinear solvers. More about symbolic adjoints of (non) linear solvers can be found in [5] and [7].

As symbolic adjoint code has to be written by hand, symbolic mode is available only for selected routines. Please refer to the routine documentation to see whether a symbolic mode is available for the adjoint routine you require.

### 3.2 Routines Without User-supplied Subroutines or Functions

#### 3.2.1 Algorithmic mode

To compute the adjoints using the NAG AD Library you need to perform a number of steps; some of which are similar to those used for dco.

1. Initialize `ad_handle`
2. Set the computation mode
3. Allocate global memory for internal data (IR)
4. Copy routine data into active data types
5. Register input variables to IR
6. Call Library routine(s)
7. Set (increment) adjoints of the outputs $y_{(1)}$
8. Interpret the IR to calculate the adjoints
9. Get the adjoints of inputs $x_{(1)}$
10. Free the internal memory
11. Free ad_handle

All of these steps must be performed when using adjoint versions of Library routines. In the following we will go through these steps in turn and discuss them in detail in the context of the example presented in the first section.

1. Initialize ad_handle: Initialization is performed by the following function call

   ```c
   void *ad_config = dco::aiw::create_config();
   ```

   This function call creates an object that can be used to configure a NAG AD Library routine. For more details please see the documentation for `x10aa_aiw_f`.

2. Set the computation mode: the computation mode is set by the following function call

   ```c
   dco::aiw::adjoint_mode(ad_config) = nagad_algorithmic;
   ```

3. Allocate global memory for IR: allocation is performed by the following function call

   ```c
   dco::aiw::global_ir = dco::aiw::ir_t::create();
   ```

   There are two reasons why the NAG AD Library uses IR. Firstly, computation of an adjoint of a function that consists of two or more subsequent routine calls (e.g., see the example in the first section) requires that the order of the routine calls be reversed. The IR tracks the order of adjoint routine calls automatically, so you do not need to take care of this. Secondly, the NAG AD Library routines can be easily integrated with dco/c++. Similar to the active data type the IR used by the NAG AD Library is binary compatible to the IR used by dco, hence for dco/c++ there is no difference between natively overloaded code and the NAG AD Library routine.

4. Copy routine data into active data types: If you do not use dco/c++ then the rest of your code is probably using standard floating point data types (e.g., `double`). In order to call the NAG AD Library routine you must copy this information into the active data types. In C++ you have two options to set the value component of the active data type
(i) through overloaded assignment operation (this is the way used in the example)

```cpp
27  for (int i = 0; i < m; ++i){
28     x[i] = double(i)/(m-1);
29     y[i] = exp(x[i]);
30  }
```

(ii) through `dco::value` function

```cpp
1  for (int i = 0; i < m; ++i){
2     dco::value(x[i]) = double(i)/(m-1);
3     dco::value(y[i]) = exp(x[i]);
4  }
```

dco::value returns a reference to a value component of the active data type. So it can be used for both reading and writing.

For more details please see the dco/c++ User Guide.

5. Register input variables to IR: before you call the NAG AD Library routines you must register input variables to IR. This is done through

```cpp
59  for (int i = 0; i < m; ++i)
60     dco::aiw::global_ir->register_variable(x[i]);
```

You should register only those inputs for which you want adjoints. In our example we are not interested in adjoints w.r.t. ordinate of the set up data points. That is why we don’t register the entries of array y. Registering of additional variables does not influence the correctness of the result but may increase the memory requirements and computational time.

6. Call library routine(s): After performing the previous steps the NAG AD Library routines can be called in a similar way to any other NAG Library routine.

```cpp
66  /* E01BA_A1W_F.
67   * Adjoint of interpolating function, cubic spline
68   * interpolant, one
69   * variable
70  */
71  E01BA_A1W_F(ad_config,m, x.data(), y.data(), lamda.
72    data(), c.data(), lck, wrk.data(), lwrk, ifail);
73  for(int j=0;j<lck;++j) {
74     dco::aiw::global_ir->register_output_variable(
75        lamda[j]);
76     dco::aiw::global_ir->register_output_variable(c[j]
77        [j]);
78  }
```
/* E02BB_A1W_F. 
* Adjoint of evaluation of fitted cubic spline, 
* function only 
*/
E02BB_A1W_F(ad_config, lck, lamda.data(), c.data(), 
xarg, fit, ifail);

Moreover, after running the NAG AD Library routines the value component of the outputs contain the same values as corresponding outputs of the primal routine.

7. Set (increment) adjoints of the outputs \( y(1) \): You should set the adjoint of the outputs so that it computes the desired adjoint. In our example we want to compute the derivative of the spline value with respect to the abscissa of the setup data points. Therefore we set the adjoint component of \( \text{fit} \) to 1.

\[
dco::\text{derivative}(\text{fit}) = 1.0;
\]

Similar to \( dco::\text{value} \), \( dco::\text{derivative} \) returns a reference to the adjoint component of the active data type, so it can be used for both, reading and writing. For more details please refer to the \( dco/c++ \) User Guide.

8. Interpret the IR to calculate the adjoints: The interpretation is performed by calling

\[
dco::\text{a1w::global_ir} -> \text{interpret_adjoint}();
\]

9. Get the adjoints of inputs \( x(1) \): As previously mentioned, the adjoints (derivative component of the active data types) can be accessed by using \( dco::\text{derivative} \) function. In our example we are interested in adjoints stored in array \( x \), therefore

\[
\text{for (int j = 0; j < m; ++j)} \ \\
\text{printf("} \
\text{"} NAG_IFMT " \%13.8f \n", j + 1, \
\text{dco::\text{derivative}(x[j])});
\]

prints the desired adjoint information to stdout.

10. Free internal memory:

\[
dco::\text{a1w::remove_config(ad_config)};
\]

11. Free \( \text{ad_handle} \):

\[
dco::\text{a1w::ir_t::remove(dco::\text{a1w::global_ir})};
\]

### 3.2.2 Symbolic mode

For routines without user-supplied subroutines or functions, the symbolic mode is called in exactly the same way as algorithmic mode. The only difference is that mode stored \( \text{ad_handle} \) must be set to \( \text{nagad_symbolic} \) by the corresponding function call.

\[
dco::\text{a1w::adjoint_mode(ad_handle)} = \text{nagad_symbolic};
\]
Neither E01BA_A1W_F nor E02BB_A1W_F are available in symbolic mode, so we cannot modify our example to use symbolic adjoints. Instead, we will use F07CA_A1W_F to demonstrate the use of symbolic adjoints. You will find the example code in Listing 3. Please refer to the documentation for routine f07ca_a1w_f, for more details.

3.3 Routines With User-supplied Subroutines or Functions

Here we discuss the use of NAG AD Library routines with user-supplied subroutines or functions based on the example for routine e04gbf. This routine is an unconstrained least squares optimizer (for more details please see the NAG Library Manual).

We will use the following example to demonstrate the derivative computation. Let’s assume that you try to find least squares estimates of \(x_1\), \(x_2\) and \(x_3\) in the model

\[
y = x_1 + \frac{t_1}{x_2 t_2 + x_3 t_3}
\]

using the 15 sets of data given in Table 1. This problem can be solved by calling e04gbf from the NAG Library as demonstrated in the code example (Listing 4). In addition to the solution of an optimization problem, this code approximates the sensitivities of the obtained solution with respect to the observed values \(y\) using the finite differences approach. As already mentioned, when using finite differences you have to solve the perturbed optimization problem for each column of the Jacobian you are interested in. In our example, the Jacobian we are trying to compute has only 15 columns, so we need to solve 15 perturbed optimization problems.

<table>
<thead>
<tr>
<th>(y)</th>
<th>(t_1)</th>
<th>(t_2)</th>
<th>(t_3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.14</td>
<td>1.0</td>
<td>15.0</td>
<td>1.0</td>
</tr>
<tr>
<td>0.18</td>
<td>2.0</td>
<td>14.0</td>
<td>2.0</td>
</tr>
<tr>
<td>0.22</td>
<td>3.0</td>
<td>13.0</td>
<td>3.0</td>
</tr>
<tr>
<td>0.25</td>
<td>4.0</td>
<td>12.0</td>
<td>4.0</td>
</tr>
<tr>
<td>0.29</td>
<td>5.0</td>
<td>11.0</td>
<td>5.0</td>
</tr>
<tr>
<td>0.32</td>
<td>6.0</td>
<td>10.0</td>
<td>6.0</td>
</tr>
<tr>
<td>0.35</td>
<td>7.0</td>
<td>9.0</td>
<td>7.0</td>
</tr>
<tr>
<td>0.39</td>
<td>8.0</td>
<td>8.0</td>
<td>8.0</td>
</tr>
<tr>
<td>0.37</td>
<td>9.0</td>
<td>7.0</td>
<td>7.0</td>
</tr>
<tr>
<td>0.58</td>
<td>10.0</td>
<td>6.0</td>
<td>6.0</td>
</tr>
<tr>
<td>0.73</td>
<td>11.0</td>
<td>5.0</td>
<td>5.0</td>
</tr>
<tr>
<td>0.96</td>
<td>12.0</td>
<td>4.0</td>
<td>4.0</td>
</tr>
<tr>
<td>1.34</td>
<td>13.0</td>
<td>3.0</td>
<td>3.0</td>
</tr>
<tr>
<td>2.10</td>
<td>14.0</td>
<td>2.0</td>
<td>2.0</td>
</tr>
<tr>
<td>4.39</td>
<td>15.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Table 1: Table 1
Next, we will see how the same derivatives can be computed with the adjoint version of this routine. E04GB_A1W_F implements the adjoint of e04gbf. It provides both algorithmic and symbolic computation modes.

3.3.1 Algorithmic mode

Use of adjoint routines with user-supplied-functions is more complicated compared to usage of routines without any user-supplied code. The reason for this is that you must provide an adjoint version of the user-supplied-functions. An example, using NAG AD Library routine E04GB_A1W_F is presented in Listing 5.

All the steps described in Section 3.2.1 are performed. So let’s take a look user-supplied-functions. The interface of the user-supplied-function is changed in a similar way to the interface of the main routine. It means that ad_handle is the first argument and all floating point arguments are now of active data type. We now need to ensure that the user-supplied functions also compute the required adjoints.

The lsqmon function is used only to monitor the progress of the optimizer so we don’t need to change this routine.

The lsqfun function computes the function value and the corresponding derivative, hence we need to provide a corresponding adjoint version of this routine.

As explained above in algorithmic mode the operator overloading tool dco is used to compute the adjoints of the routines. The adjoints are computed in two steps:

i record all computations needed to compute the primal value to the IR;

ii compute the adjoints by interpreting the IR.

So the adjoints must be made available during the IR interpretation. This is achieved by moving the adjoint computation to a separate routine that is called by the IR interpreter. In the documentation this user-supplied routine is referred to as the companion callback. The user-supplied callback must only compute the primal values, tell the IR interpreter where the companion callback is located and provide the companion callback with the information needed to compute the adjoint. The basic procedure for this is as follows:

1. Create callback data object

2. Write input arguments to callback data object

3. Calculate primal values

4. Register output variables

5. Write registered variables to callback data object

6. Insert the companion callback and the callback data object in IR.
Discussing each of these steps for our example.

1. Create callback data object: This object is used to transport the information to the companion callback.

   ```c
   dco::a1w::external_adjoint_object_t *D = dco::a1w::
   global_ir->create_callback_object<dco::a1w::
   external_adjoint_object_t>();
   
   2. Write input arguments to callback data object: These values are needed to compute the adjoints in the companion callback.

   ```c
   // store inputs
   D->write_data(iflag);
   D->write_data(m);
   D->write_data(n);
   D->write_data(ldfjac);
   D->write_data(liw);
   D->write_data(lw);
   
   for (int i =0; i<n; i++)
     D->write_data(xc[i]);
   
   for (int i =0; i<lw; i++)
     D->write_data(w[i]);
   ```

3. Calculate primal values: We use the implementation of `lsqfun` from the finite differences example to compute the function values.

   ```c
   lsqfun(ad_handle, iflag, m, n, dco::value(xc), dco::value(fvec), dco::value(fjac), ldfjac, iw, liw, dco::value(w), lw);
   ```

4. Register output variables and write registered variables to callback data object: The adjoints of the outputs of the user-supplied function are needed as input to our adjoint computation.

   ```c
   //register outputs
   for (int i = 0; i<m; i++) {
     dco::a1w::global_ir->register_variable(fvec[i]);
     D->write_data(fvec[i]);
   }
   
   for (int i = 0; i<ldfjac*n; i++) {
     dco::a1w::global_ir->register_variable(fjac[i]);
     D->write_data(fjac[i]);
   }
   ```

5. Insert companion callback and the callback object in IR:

   ```c
   dco::a1w::global_ir->insert_callback(&
   lsqfun_a1w_tape_callback, D);
   ```
Now that we have described how to implement the user-supplied function, let us now see how to implement the companion callback correctly.

The companion callback always has the following interface no matter how the user-supplied routine looks:

```cpp
void lsqfun_a1w_tape_callback(dco::a1w::external_adjoint_object_t *D) {
    // For the callback companion we also have a basic procedure to follow:
    1. Read data from callback data object
    2. Get adjoints of outputs
    3. Calculate the adjoint increments of inputs
    4. Increment the adjoint of inputs

    Discussing these steps for our example.
    1. Read data from callback data object:
       ```
       int const & iflag = D->read_data<int>();
       int const & m = D->read_data<int>();
       int const & n = D->read_data<int>();
       int const & ldfjac = D->read_data<int>();
       int const & liw = D->read_data<int>();
       int const & lw = D->read_data<int>();

       for (int i = 0; i < n; i++)
           xc[i] = D->read_data<dco::a1w::type>();

       for (int i = 0; i < lw; i++)
           w[i] = D->read_data<dco::a1w::type>();

       for (int i = 0; i < m; i++) {
           fvec[i] = D->read_data<dco::a1w::type>();
           fveca[i] = dco::derivative(fvec[i]);
       }

       for (int i = 0; i < ldfjac * n; i++) {
           fjac[i] = D->read_data<dco::a1w::type>();
           fjaca[i] = dco::derivative(fjac[i]);
       }
       ```
    2. Get adjoints of outputs:
       ```
       for (int i = 0; i < m; i++) {
           fvec[i] = D->read_data<dco::a1w::type>();
       }
       ```
```

The data is read in the same order it was stored in the callback data object. You must specify the correct data type during the read operation.

2. Get adjoints of outputs:
   ```
   for (int i = 0; i < m; i++) {
       fvec[i] = D->read_data<dco::a1w::type>();
   }
   ```
```
fveca[i] = dco::derivative(fvec[i]);
}

for (int i = 0; i<ldfjac*n; i++) {
    fjac[i] = D->read_data<dco::aiw::type>();
    fjaca[i] = dco::derivative(fjac[i]);
}

3. Calculate the adjoint increments of inputs: In our example we must compute the adjoint of $x$ and $st.y$.

// adjoint of lsqfun
for (int i = m-1; i>=0; i--)
    denom = dco::value(xc[1]) * dco::value(st.t[i][1]) + dco::value(xc[2]) * dco::value(st.t[i][2]);
    denoma = 0.0;
    if (iflag !=0) {
        dummy = -1/(denom*denom);
        dummya = 0.0;
        sta.t[i][0] += FJACA(i,2)*dco::value(st.t[i][2])*dummy;
        sta.t[i][2] += FJACA(i,2)*dco::value(st.t[i][0])*dummy;
        dummya += FJACA(i,2)*dco::value(st.t[i][0])*dco::value(st.t[i][2]);
        sta.t[i][0] += FJACA(i,1)*dco::value(st.t[i][1])*dummy;
        sta.t[i][1] += FJACA(i,1)*dco::value(st.t[i][0])*dummy;
        dummya += FJACA(i,1)*dco::value(st.t[i][0])*dco::value(st.t[i][1]);
        denoma += dummya*2.0/(denom*denom*denom);
    }
    if (iflag != 1) {
        xca[0] += fveca[i];
        sta.t[i][0] += fveca[i]/denom;
        denoma += -fveca[i]*dco::value(st.t[i][0])/(denom*denom);
        sta.y[i] += -fveca[i];
    }
    xca[i] += denoma*dco::value(st.t[i][1]);
The adjoint increments are stored in arrays \( \text{xca} \) and \( \text{sta.y} \).

4. Increment the adjoints of inputs:

\[
\begin{align*}
\text{for (int } i \text{ = 0; } i < n; i++) & \\
\text{dco::derivative(xc[i]) += xca[i];} & \\
\text{for (int } i \text{ = 0; } i < m; i++) & \\
\text{dco::derivative(st.y[i]) += sta.y[i];}
\end{align*}
\]

Once we get our \texttt{lsqfun} and its companion callback correct, we are able to compute the adjoints.

### 3.3.2 Symbolic mode

The NAG AD Library contains a number of hand-written symbolic adjoints. Symbolic adjoints may exploit mathematical properties of a Library routine yielding a more efficient adjoint calculation but their use can also have restrictions. The typical restrictions are: derivatives with respect to some arguments of the function are not computed; and the primal algorithm is assumed to converge.

For the symbolic adjoint of routine \texttt{e04gbf} both restrictions mentioned above apply. It assumes that the primal optimizer has converged and it allows to compute the derivatives only with respect to the parameters of the residual function (see [4]). Specifically it does not allow to compute the derivative with respect to starting point of the optimization. In Listing [9] we implemented the same example using symbolic adjoints. Below we describe this code in more detail.

The user-supplied function of routine \texttt{e04gbf} evaluates

\[
(z, g) = (f(x, p), \nabla_x f(x, p)).
\]

To use the symbolic adjoint of routine \texttt{e04gbf} the companion callback must support computation of the following for modes:

1. Function evaluation only, i.e.,

\[
(f(x, p), \nabla_x f(x, p)).
\]

2. Function evaluation and adjoint computation w.r.t. state \( x \), i.e.,

\[
x_{(1)}^{(1)} = [\nabla_x f(x, p)]^T z_{(1)} + [\nabla^2_x f(x, p)]^T g_{(1)}.
\]
3. Function evaluation and adjoint computation w.r.t. parameter $p$, i.e.,

$$p(1) + = \left[ \nabla_p f(x, p) \right]^T z(1) + \left[ \nabla^2_p f(x, p) \right]^T g(1).$$

4. Function evaluation and computation of all adjoints.

The required mode is stored in the callback data object

```cpp
D->write_data(callmode);
```

and then passed over to the companion callback

```cpp
int const & callmode = D->read_data<int>();
```

The companion callback increments the adjoints of the state or parameter only if it is requested by the corresponding mode.

```cpp
if (callmode == nagad_dx || callmode == nagad_full) {
    for (int i = 0; i<n; i++)
        dco::derivative(xc[i]) += xca[i];
}
if (callmode == nagad_druser || callmode == nagad_full) {
    for (int i = 0; i<m; i++)
        dco::derivative(st.y[i]) += sta.y[i];
```

4 Interfacing the NAG AD Library with dco/c++

dco/c++ is an AD tool based on operator overloading developed by STCE group from RWTH-Aachen University in cooperation with NAG. For more details please see ([2]).

The C++ interface described here is the same interface as used by dco/c++. As previously mentioned the active data types and the IR are binary compatible with that of dco/c++, hence all examples presented here will work with dco/c++ out of the box.

Usage of dco/c++ allows you to call the NAG AD Library with user-supplied subroutines or functions in a much more convenient way.

4.1 Algorithmic Mode

With dco/c++ you can use NAG AD Library routines with user-supplied functions in algorithmic mode in the same way as routines without user callbacks as shown in Listing 7. With dco/c++ you don’t need to change the user-supplied routine at all and you also avoid writing the companion callback. The most important thing is that you don’t have to write the adjoint code for the user-supplied routine which is a very error prone job. Instead the adjoint code is automatically generated by dco/c++. 

16
4.2 Symbolic Mode

In symbolic mode dco/c++ can also help to reduce development time. Although in this mode we still need to modify the user-supplied routine and provide companion callbacks that provides the desired modes (adjoints w.r.t. state or parameter), we can rely on dco/c++ to compute the adjoints instead of writing them by hand. The corresponding code is shown in Listing 8. We don’t write the adjoint code by hand, we only insure that if only adjoints w.r.t. state are required we do not increment the adjoints of the parameters

```cpp
else if (callmode == nagad_dx)
    lsqfun_dx(ad_handle, iflag, m, n, xc, fvec, fjac,
             ldfjac, iw, liw, w, lw);
```

This is done by using only the value component of the arrays st.y and st.t.

```cpp
for (int i = 0; i < m; ++i) {
    denom = xc[1] * dco::value(st.t[i][1]) + xc[2] * dco::value(st.t[i][2]);
    if (iflag != 1)
        fvec[i] = xc[0] + dco::value(st.t[i][0]) / denom - dco::value(st.y[i]);
    if (iflag != 0) {
        FJAC(i, 0) = 1.0;
        dummy = -1.0 / (denom * denom);
        FJAC(i, 1) = dco::value(st.t[i][1]) * dummy;
        FJAC(i, 2) = dco::value(st.t[i][2]) * dummy;
    }
}
```

Similarly, if only the adjoints w.r.t. to parameters are required we use only value components of the array xc

```cpp
else if (callmode == nagad_druser)
    lsqfun_druser(ad_handle, iflag, m, n, xc, fvec,
                  fjac, ldfjac, iw, liw, w, lw);

for (int i = 0; i < m; ++i) {
    denom = dco::value(xc[1]) * st.t[i][1] + dco::value(xc[2]) * st.t[i][2];
    if (iflag != 1)
        fvec[i] = dco::value(xc[0]) + st.t[i][0] / denom - st.y[i];
    if (iflag != 0) {
        FJAC(i, 0) = 1.0;
    }
```

17
dummy = -1.0 / (denom * denom);
FJAC(i, 1) = st.t[i][0] * st.t[i][1] * dummy;
FJAC(i, 2) = st.t[i][0] * st.t[i][2] * dummy;
}

In case all adjoints are required (nagad_full) the original overloaded function is executed
else if (callmode==nagad_full)
    lsqfun(ad_handle, iflag, m, n, xc, fvec, fjac, ldfjac, iw, liw, w, lw);

As we can see the development overhead for using symbolic adjoint with dco/c++ is quite small.

5 Conclusion

Bullet points that can be used in presentations

Why should you use NAG AD Library

• adjoints allow fast computation of derivatives
• exact derivatives instead of approximations when finite differences is used
• NAG Library users who apply AD can now use high quality adjoint routines from NAG
  – no need to write adjoint versions of these routine or
  – search for proper replacement
• NAG AD Library can be used with any AD tool
• Easy switch between symbolic and algorithmic adjoints
  – same interface for symbolic and algorithmic adjoints
• a single AD solution - dco/c++ and NAG AD Library
• use of IR to reverse the routine call tree
• specify arguments you are not interested in derivatives w.r.t.
• more routines available on request
  – additional algorithmic adjoints that haven’t made it into the release
  – consultancy for adjoint routines (symbolic, hybrid, robust)
Why do you want dco/c++ and NAG AD Library

- NAG AD Library routines can be used as intrinsics in dco/c++ (no difference between sin and a NAG AD Library routine)
- Algorithmic adjoints of routines with user-supplied functions can be used without additional development time
- smooth transition from non dco/c++ solution to solution with dco/c++
- no need to copy variables when used with dco/c++ (binary compatible data types)
- smaller development overhead with dco/c++

6 Example Codes

Listing 1: "Spline adjoint"

```cpp
1 #include <vector>
2 #include <dco.hpp>
3 #include <nagad.hpp>

4 #define MMAX 7

5 double getTime(void){
6     struct timespec tv;
7     if(clock_gettime(CLOCK_REALTIME, &tv) != 0)
9         return 0;
11     return (((double) tv.tv_sec) + (double) (tv.
13         tv_nsec / 1000000000.0));
15 }

16 template<typename T>
17 T initialize(int &m, int &lck, int &lwrk, std::vector
19     <T> &x, std::vector<T> &y, std::vector<T> &lamda, std::vector<T> &c, std
20     :vector<T> &wrk){
22     lck = m + 4;
23     lwrk = 6*m + 16;
25     x.resize(m);
26     y.resize(m);
27     lamda.resize(lck);
28     c.resize(lck);
```
wrk.resize(lwrk);
for (int i = 0; i < m; ++i){
    x[i] = double(i)/(m-1);
    y[i] = exp(x[i]);
}
return (x[m/2] + x[m/2+1])/2.0;
}

int main(int argc, char **argv)
{
    Integer exit_status = 0, m = MMAX, lck=MMAX+4, lwrk =0;
    Integer ifail=-1;
    std::vector<dco::aiw::w_rtype> lamda, c, wrk, x, y;
    dco::aiw::w_rtype fit, xarg;
    if (argc > 1){
        m = atoi(argv[1]);
    }
    /* Initialize spline */
    printf("nag_1d_spline_interpolant (e01bac) Example Program Results\n");
    void *ad_config=dco::aiw::create_config();
    dco::aiw::adjoint_mode(ad_config) = nagad_algorithmic;
    dco::aiw::global_ir = dco::aiw::ir_t::create();
    xarg = initialize(m,lck,lwrk,x,y, lamda,c,wrk);
    for (int i = 0; i < m; ++i)
        dco::aiw::global_ir->register_variable(x[i]);
    double start = getTime();
    /* E01BA_AIW_F.
    * Adjoint of interpolating function, cubic spline

interpolant, one

* variable

E01BA_A1W_F(ad_config,m, x.data(), y.data(), lamda.
data(), c.data(), lck, wrk.data(), lwrk, ifail);

for(int j=0;j<lck;++j) {
    dco::a1w::global_ir->register_output_variable(
lamda[j]);
    dco::a1w::global_ir->register_output_variable(c[j]);
}

/* E02BB_A1W_F.
* Adjoint of evaluation of fitted cubic spline, function only
*/
E02BB_A1W_F(ad_config, lck, lamda.data(), c.data(),
xarg, fit, ifail);

if (m < 8) {
    printf(\nNumber of distinct knots = %" NAG_IFMT
        \n\n", m - 2);
    printf("Distinct knots located at \n\n");
    for (int j = 3; j < m + 1; j++)
        printf("%8.4f\n", dco::value(lamda[j]), (j - 3)
            % 5 == 4
            || j == m ? "\n" : "");
    printf("\n\n\n J B-spline coeff c\n\n");
    for (int j = 0; j < m; ++j)
        printf(" %" NAG_IFMT " %13.4f\n", j + 1,
            dco::value(c[j]));
}

printf("\n J Abscissa Ordinate Spline\n\n");

if (m < 8) {
    printf(" %13.4f %13.4f \n", 
        dco::value(xarg), dco::value(fit));
}

dco::a1w::global_ir->register_output_variable(fit);

dco::derivative(fit) = 1.0;

dco::a1w::global_ir->interpret_adjoint();

double end = getTime();
if (m < 8) {
    printf("\n\n J x_a1s \n\n");
    for (int j = 0; j < m; ++j)
        printf(" %" NAG_IFMT " %13.8f \n", j + 1, 
dco::derivative(x[j]));

    printf("\n\n J lamda_a1s c_a1s \n\n");
    for (int j = 0; j < lck; ++j)
        printf(" %" NAG_IFMT " %13.8f %13.8f\n", j + 1, 
dco::derivative(lamda[j]), dco::derivative(c[j]));
}
std::cout << "time " << end - start << std::endl;
dco::aiw::remove_config(ad_config);
dco::aiw::ir_t::remove(dco::aiw::global_ir);
/* Free memory allocated */
return exit_status;
}

Listing 2: "Spline finite differences"

#include <vector>
#include <iostream>
#include <nagmk26.h>
#include <cmath>
#define MMAX 7
double getTime(void){
    struct timespec tv;
    if(clock_gettime(CLOCK_REALTIME, &tv) != 0)
        return 0;
    return (((double) tv.tv_sec) + (double) (tv.
    tv_nsec / 1000000000.0));
}
template<type name T>
T initialize(int &m, int &lck, int & lwrk, std::vector<T> &x, std::vector<T> &y,
    std::vector<T> &lamda, std::vector<T> &c, std::vector<T> & wrk){
    lck = m + 4;
    lwrk = 6*m + 16;
    x.resize(m);
    y.resize(m);
    lamda.resize(lck);
    c.resize(lck);
    wrk.resize(lwrk);
    for (int i = 0; i < m; ++i){
        x[i] = double(i)/(m - 1);
        y[i] = exp(x[i]);
    }
    return (x[m/2] + x[m/2+1])/2.0;
}

int main(int argc, char **argv)
{
    Integer exit_status = 0, m = MMAX, lck=MMAX+4, lwrk =0;
    Integer ifail=-1;
    std::vector<double> x, y, lamda, c, wrk;
    double fith, fith1;
    double fit, xarg;
    double h = 1e-8;
    if (argc > 1){
        m = atoi(argv[1]);
    }
    /* Initialize spline */
    printf("nag_1d_spline_interpolant (e01bac) Example Program Results\n");
    xarg = initialize(m,lck,lwrk,x,y, lamda,c,wrk);
    double start = getTime();
if (m < 8) {
    printf("\nNumber of distinct knots = %" NAG_IFMT "\n\n", m - 2);
    printf("Distinct knots located at \n\n");
    for (int j = 3; j < m + 1; j++)
        printf("%8.4f%s", lamda[j], (j - 3) % 5 == 4 || j == m ? "\n" : "");
    printf("\n\n J B-spline coeff c\n\n");
    for (int j = 0; j < m; ++j)
        printf(" %" NAG_IFMT " %13.4f\n", j + 1, c[j]);
}

printf("\n J Abscissa Ordinate Spline\n\n");

if (m < 8) {
    printf(" %13.4f %13.4f\n", xarg, fit);
}

/* Store lamda and c as they destroyed inside the next loop */
std::vector<double> lamdah(lck), ch(lck);
for (int i = 0; i < lck; ++i) {
    lamdah[i] = lamda[i];
    ch[i] = c[i];
}

if (m < 8) {
    printf("\n\n J x_a1s\n\n");
}
for (int i = 0; i < m; ++i) {
    x[i] += h;
    e01baf_(m, x.data(), y.data(), lamda.data(), c.data(), lck, wrk.data(), lwrk, ifail);
    e02bbf_(lck, lamda.data(), c.data(), xarg, fit, ifail);
    x[i] -= h;
    if (m < 8) {
        printf(" %" NAG_IFMT " %13.8f \n", i + 1, (
Listing 3: "f07ca_a1w_f symbolic adjoint"

```c
#include <dco.hpp>
#include <nagad.hpp>

int main() {
    int n = 2, nrhs = 1, ldb = 2, ifail;
    dco::aiw::w_rtype dl[1] = {0}, du[1] = {0}, d[2] = {1, 2};
    // declare in addition a pure input variable
    dco::aiw::w_rtype b_in[2] = {1, 1}, b[2];
    dco::aiw::global_ir = dco::aiw::ir_t::create();
    dco::aiw::adjoint_mode(ad_handle) = nagad_symbolic;
```
// register pure input here
dco::a1w::global_ir->register_variable(b_in, 2);

// copy into in/out variable
b[0] = b_in[0];
b[1] = b_in[1];

void *config = dco::a1w::create_config();

// b gets overwritten
F07CA_A1W_F(config, n, nrhs, dl, d, du, b, ldb, ifail);

// set adjoint of output
dco::derivative(b[0]) += 1.0;
dco::derivative(b[1]) += 1.0;
dco::a1w::global_ir->interpret_adjoint();

// read adjoint of input
std::cout << "db/db = (" << dco::derivative(b_in[0]) << " " << dco::derivative(b_in[1]) << " ) " << std::endl;
dco::a1w::remove_config(config);
dco::a1w::ir_t::remove(dco::a1w::global_ir);
return 0;
}

Listing 4: "e04gb finite differences"

#include <dco.hpp>
#include <nagad.hpp>

#define MMAX 15
#define TMAX 3

/* Define a user structure template to store data in lsqfun. */
struct user
{
  dco::a1w::type y[MMAX];
  dco::a1w::type t[MMAX][TMAX];
};
static user st;
void lsqfun(void* config,
    int& iflag,
    const int& m,
    const int& n,
    const dco::a1w::type xc[],
    dco::a1w::type fvec[],
    dco::a1w::type fjac[],
    const int& ldfjac,
    int iw[],
    const int& liw,
    dco::a1w::type w[],
    const int& lw)
{

    #define FJAC(I, J) fjac[(J) * ldfjac + (I)]

    Integer i;
    dco::a1w::type denom, dummy;

    for (i = 0; i < m; ++i) {
        denom = xc[1] * st.t[i][1] + xc[2] * st.t[i][2];
        if (iflag != 1)
            fvec[i] = xc[0] + st.t[i][0] / denom - st.y[i];

        if (iflag != 0) {
            FJAC(i, 0) = 1.0;
            dummy = -1.0 / (denom * denom);
            FJAC(i, 1) = st.t[i][0] * st.t[i][1] * dummy;
            FJAC(i, 2) = st.t[i][0] * st.t[i][2] * dummy;
        }
    }

    /* lsqfun */

    void lsqmon(void* config,
        const int&,
        const dco::a1w::type[],
        const dco::a1w::type[],
        const dco::a1w::type[],
        const int&,
        const dco::a1w::type[],
        const int&,
        const int&,
const int &,
int [],
const int &,
dco::aiw::type[],
const int &) {}
x[1] = 1.0;
x[2] = 1.5;

dco::aiw::global_ir = dco::aiw::ir_t::create();

// for (int i = 0; i<m; i++)
//  dco::aiw::global_ir->register_variable(st.y[i]);

int ifail = -1;
void * config = dco::aiw::create_config();

E04GB_A1W_F(config, m, n,
    E04HEV, lsqfun, lsqmon,
    iprint, maxcal, eta, xtol, stepmx, x,
    fsumsq, fvec, fjac, ldfjac,
    s, v, ldv, niter, nf, &iw, liw, w, lw, ifail) ;

// dco::derivative(fsumsq) += 1.0;
// dco::aiw::global_ir->interpret_adjoint();

std::cout << "Solution" << std::endl;
for (int i = 0; i<n; i++)
    std::cout << "x[" << i << "]= " << dco::value(x[i]) << std::endl;

std::cout << "\n\nNorm of residual = " << dco::value(fsumsq) << std::endl;

std::cout << "\n\nResiduals" << std::endl;
for (int i = 0; i<m; i++)
    std::cout << "fvec[" << i << "]= " << dco::value(fvec[i]) << std::endl;

std::cout << "\n\n dFsumsq/dY FD " << std::endl;
for (int i = 0; i<m; i++) {
    x[0] = 0.5;
    x[1] = 1.0;
    x[2] = 1.5;
    st.y[i] += h;
    E04GB_A1W_F(config, m, n,
        E04HEV, lsqfun, lsqmon,
        iprint, maxcal, eta, xtol, stepmx, x,
fsumsqh, fvec, fjac, ldfjac,
s, v, ldv, niter, nf, &iw, liw, w, lw, ifail);

std::cout << "dFsumsq/dY[" << i << "]= " << dco::value(fsumsqh - fsumsq)/h << std::endl;
    st.y[i] -= h;
}
delete [] fjac;
delete [] fvec;
delete [] x;
delete [] v;
delete [] s;
delete [] w;
}

Listing 5: "e04gb adjoint algorithmic"

#include <vector>
#include <dco.hpp>
#include <nagad.hpp>

#define MMAX 15
#define TMAX 3

/* Define a user structure template to store data in lsqfun. */
template <typename T>
struct user
{
    T y[MMAX];
    T t[MMAX][TMAX];
};

static user<dco::a1w::type> st;

template <typename T>
void lsqfun(void* ad_handle,
    int& iflag,
    const int& m,
    const int& n,
    const T xc[],
    T fvec[],
    T fjac[],
    const int& ldfjac,
    int iw[],
const int & liw,
    T w[],
    const int & lw) {

#define FJAC(I, J) fjac[(J) * ldfjac + (I)]

    Integer i;
    T denom, dummy;

    for (i = 0; i < m; ++i) {
        denom = xc[1] * st.t[i][1] + xc[2] * st.t[i][2];
        if (iflag != 1)
            fvec[i] = xc[0] + st.t[i][0] / denom - st.y[i];
        if (iflag != 0) {
            FJAC(i, 0) = 1.0;
            dummy = -1.0 / (denom * denom);
            FJAC(i, 1) = st.t[i][0] * st.t[i][1] * dummy;
            FJAC(i, 2) = st.t[i][0] * st.t[i][2] * dummy;
        }
    }
}

/* lsqfun */

void lsqfun_a1w_tape_callback(dco::a1w::
    external_adjoint_object_t *D) {
    // read checkpointed data
    int const & iflag = D->read_data<int>();
    int const & m = D->read_data<int>();
    int const & n = D->read_data<int>();
    int const & ldfjac = D->read_data<int>();
    int const & liw = D->read_data<int>();
    int const & lw = D->read_data<int>();

#define FJACA(I, J) fjaca[(J) * ldfjacac + (I)]

    int iflag1 = iflag;
    std::vector<dco::a1w::type> xc(n), fvec(m), fjac(ldfjac*n), w(lw);
    std::vector<double> fveca(m), fjaca(ldfjacac*n);

    for (int i=0; i<n; i++)
        xc[i] = D->read_data<dco::a1w::type>();

31
for (int i = 0; i < lw; i++)
    w[i] = D->read_data<dco::a1w::type>();
for (int i = 0; i < m; i++) {
    fvec[i] = D->read_data<dco::a1w::type>();
    fvec[i] = dco::derivative(fvec[i]);
}
for (int i = 0; i < ldfjac*n; i++) {
    fjac[i] = D->read_data<dco::a1w::type>();
    fjaca[i] = dco::derivative(fjac[i]);
}
std::vector<double> xca(n);
double denoma = 0, dummya = 0, denom, dummy;
user<double> sta;
for (int i = 0; i < n; i++) xca[i] = 0.0;
for (int i = 0; i < m; i++) sta.y[i] = 0.0;
for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++)
        sta.t[i][j] = 0.0;
}
// adjoint of lsqfun
for (int i = m - 1; i >= 0; i--){
    denom = dco::value(xc[1]) * dco::value(sta.t[i][1]) + dco::value(xc[2]) * dco::value(sta.t[i][2]);
    denoma = 0.0;
    if (iflag != 0) {
        dummy = -1/(denom*denom);
        dummya = 0.0;
        sta.t[i][0] += FJACA(i,2)*dco::value(sta.t[i][2])*dummy;
        sta.t[i][2] += FJACA(i,2)*dco::value(sta.t[i][0])*dummy;
        dummya += FJACA(i,2)*dco::value(sta.t[i][0])*dco::value(sta.t[i][2]);
        sta.t[i][0] += FJACA(i,1)*dco::value(sta.t[i][1])*dummy;
        sta.t[i][1] += FJACA(i,1)*dco::value(sta.t[i][1])*dummy;
    }
dummy += FJACA(i,1)*dco::value(st.t[i][0])*dco::value(st.t[i][1]);

denoma += dummya*2.0/(denom*denom*denom);

}  
if (iflag != 1) {  
  xca[0] += fveca[i];
  sta.t[i][0] += fveca[i]/denom;
  denoma += -fveca[i]*dco::value(st.t[i][0])/(denom*denom);
  sta.y[i] += -fveca[i];
}

xca[1] += denoma*dco::value(st.t[i][1]);  
sta.t[i][1] += denoma*xca[1];

xca[2] += denoma*dco::value(st.t[i][2]);

sta.t[i][2] += denoma*xca[2];

}  
for (int i = 0; i<n; i++)  
  dco::derivative(xc[i]) += xca[i];

for (int i = 0; i<m; i++)  
  dco::derivative(st.y[i]) += sta.y[i];

for (int i = 0; i<m; i++) {
  for (int j = 0; j<n; j++)
    dco::derivative(st.t[i][j]) += sta.t[i][j];
}

} /* lsqfun_a1w_tape_callback */

template <typename T>
void lsqfun_a1w(void* ad_handle,
    int& iflag,
    const int& m,
    const int& n,
    const T xc[],
    T fvec[],
    T fjac[],
    const int& ldfjac,
    int iw[],
    const int& liw,
T w[],
   const int& lw) {

dco::alw::external_adjoint_object_t *D = dco::alw::
global_ir->create_callback_object<dco::alw::
   external_adjoint_object_t>();

   // store inputs
   D->write_data(iflag);
   D->write_data(m);
   D->write_data(n);
   D->write_data(ldfjac);
   D->write_data(liw);
   D->write_data(lw);

   for (int i =0; i<n; i++)
      D->write_data(xc[i]);

   for (int i =0; i<lw; i++)
      D->write_data(w[i]);
   // evaluate function
   lsqfun(ad_handle, iflag, m, n, dco::value(xc), dco
      ::value(fvec), dco::value(fjac), ldfjac, iw, liw
      , dco::value(w), lw);

   // register outputs
   for (int i = 0; i<m; i++) {
      dco::alw::global_ir->register_variable(fvec[i]);
      D->write_data(fvec[i]);
   }

   for (int i = 0; i<ldfjac*n; i++) {
      dco::alw::global_ir->register_variable(fjac[i]);
      D->write_data(fjac[i]);
   }

   // insert adjoint (tape) callback
   dco::alw::global_ir->insert_callback(&
      lsqfun_alw_tape_callback, D);
} /* lsqfun_alw */

void lsqmon(void* ad_handle,
   const int&,
   const int&,
   const dco::alw::type[],

const dco::aiw::type[],
const dco::aiw::type[],
const int&,
const dco::aiw::type[],
const int&,
const int&,
const int&,
int[],
const int&,
dco::aiw::type[],
const int&) {}
/* Set up the starting point */
x[0] = 0.5;
x[1] = 1.0;
x[2] = 1.5;
dco::alw::global_ir = dco::alw::ir_t::create();
for (int i = 0; i<m; i++)
    dco::alw::global_ir->register_variable(st.y[i]);

int ifail = -1;
void *ad_handle = dco::alw::create_config();

E04GB_A1W_F(ad_handle, m, n,
    E04HEV, lsqfun_alw, lsqmon,
    iprint, maxcal, eta, xtol, stepmx, x.data(),
    fsumsq, fvec.data(), fjac.data(), ldfjac,
    s.data(), v.data(), ldv, niter, nf, &iw, liw,
    w.data(), lw, ifail);

dco::derivative(fsumsq) += 1.0;
dco::alw::global_ir->interpret_adjoint();

std::cout << "Solution" << std::endl;
for (int i = 0; i<n; i++)
    std::cout << "x[" << i << "]= " << dco::value(x[i]) << std::endl;

std::cout << "\n\nNorm of residual = " << dco::value(fsumsq) << std::endl;

std::cout << "\n\nResiduals"<< std::endl;
for (int i = 0; i<m; i++)
    std::cout << "fvec[" << i << "]= " << dco::value(fvec[i]) << std::endl;

std::cout << "\n\n dFsumsq/dY NODCO ALG"<< std::endl;
for (int i = 0; i<m; i++)
    std::cout << "dFsumsq/dY[" << i << "]= " << dco::
        derivative(st.y[i]) << std::endl;
Listing 6: "e04gb adjoint symbolic"

```cpp
#include <vector>
#include <dco.hpp>
#include <nagad.hpp>

#define MMAX 15
#define TMAX 3

/* Define a user structure template to store data in lsqfun. */
template <typename T>
struct user
{
    T y[MMAX];
    T t[MMAX][TMAX];
};

static user<dco::aiw::type> st;

template <typename T>
void lsqfun(void* ad_handle,
            int& iflag,
            const int& m,
            const int& n,
            const T xc[],
            T fvec[],
            T fjac[],
            const int& ldfjac,
            int iw[],
            const int& liw,
            T w[],
            const int& lw) {

#define FJAC(I, J) fjac[(J) *ldfjac + (I)]

    Integer i;
    double denom, dummy;
```
for (i = 0; i < m; ++i) {
    denom = dco::value(xc[1]) * dco::value(st.t[i][1]) + dco::value(xc[2]) * dco::value(st.t[i][2]);
    if (iflag != 1)
        dco::value(fvec[i]) = dco::value(xc[0]) + dco::value(st.t[i][0]) / denom - dco::value(st.y[i]);
    else
        dco::value(FJAC(i, 0)) = 1.0; 
        dummy = -1.0 / (denom * denom);
        dco::value(FJAC(i, 1)) = dco::value(st.t[i][0]) * dco::value(st.t[i][1]) * dummy;
        dco::value(FJAC(i, 2)) = dco::value(st.t[i][0]) * dco::value(st.t[i][2]) * dummy;
}
/* lsqfun */

void lsqfun_a1w_tape_callback(dco::a1w::external_adjoint_object_t *D) {
    // read checkpointed data
    int const & callmode = D->read_data<int>();
    int const & iflag = D->read_data<int>();
    int const & m = D->read_data<int>();
    int const & n = D->read_data<int>();
    int const & ldjac = D->read_data<int>();
    int const & liw = D->read_data<int>();
    int const & lw = D->read_data<int>();
    #define FJACA(I, J) fjaca[(J) * ldjac + (I)]

    int iflag1 = iflag;
    std::vector<dco::a1w::type> xc(n), fvec(m), fjac(ldjac*n), w(liw);
    std::vector<double> fveca(m), fjaca(ldjac*n);
    for (int i = 0; i < n; i++)
        xc[i] = D->read_data<dco::a1w::type>();
    for (int i = 0; i < liw; i++)
        w[i] = D->read_data<dco::a1w::type>();
for (int i = 0; i < m; i++) {
    fvec[i] = D->read_data<dco::a1w::type>();
    fvec[i] = dco::derivative(fvec[i]);
}

for (int i = 0; i < ldfjac*n; i++) {
    fjac[i] = D->read_data<dco::a1w::type>();
    fjaca[i] = dco::derivative(fjac[i]);
}

std::vector<double> xca(n);
for (int i = 0; i < n; i++) xca[i] = 0.0;
for (int i = 0; i < m; i++) sta.y[i] = 0.0;
for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++)
        sta.t[i][j] = 0.0;
}

// adjoint of lsqfun
for (int i = m-1; i >= 0; i--){
    denom = dco::value(xc[1]) * dco::value(sta.t[i][1]) + dco::value(xc[2]) * dco::value(sta.t[i][2]);
    denoma = 0.0;
    if (iflag != 0) {
        dummy = -1/(denom*denom);
        dummya = 0.0;
        sta.t[i][0] += FJACA(i,2)*dco::value(sta.t[i][0])*dummy;
        sta.t[i][2] += FJACA(i,2)*dco::value(sta.t[i][0])*dummy;
        dummya += FJACA(i,2)*dco::value(sta.t[i][0])*dco::value(sta.t[i][2]);
        sta.t[i][0] += FJACA(i,1)*dco::value(sta.t[i][0])*dco::value(sta.t[i][1]);
        sta.t[i][1] += FJACA(i,1)*dco::value(sta.t[i][0])*dco::value(sta.t[i][1]);
        dummya += FJACA(i,1)*dco::value(sta.t[i][0])*dco::value(sta.t[i][1]);
    }
}
denoma += dummya*2.0/(denom*denom*denom);
}
if (iflag != 1) {
    xca[0] += fveca[i];
    sta.t[i][0] += fveca[i]/denom;
    denoma += -fveca[i]*dco::value(sta.t[i][0])/(denom*denom);
    sta.y[i] += -fveca[i];
}

    xca[1] += denoma*dco::value(sta.t[i][1]);
    sta.t[i][1] += denoma*xca[1];
    xca[2] += denoma*dco::value(sta.t[i][2]);
    sta.t[i][2] += denoma*xca[2];
}
if (callmode == nagad_dx || callmode == nagad_full) {
    for (int i = 0; i<n; i++)
        dco::derivative(xc[i]) += xca[i];
}
if (callmode == nagad_druser || callmode == nagad_full) {
    for (int i = 0; i<m; i++)
        dco::derivative(sta.y[i]) += sta.y[i];
    for (int i = 0; i<m; i++) {
        for (int j = 0; j<n; j++)
            dco::derivative(sta.t[i][j]) += sta.t[i][j];
    }
} /* lsqfun_a1w_tape_callback */

template <typename T>
void lsqfun_a1w(void* ad_handle,
    int& iflag,
    const int& m,
    const int& n,
    const T xc[],
    T fvec[],
    T fjac[],
    const int& ldfjac,
    int iw[],
    int jw[],
    T fjacw[])

```cpp
const int & liw, 
  T w[], 
const int & lw) {

dco::alw::external_adjoint_object_t *D = dco::alw::
global_ir->create_callback_object<dco::alw::
  external_adjoint_object_t>();
int callmode = dco::alw::callback_mode(ad_handle);

if(callmode != nagad_evalonly) {
  // store inputs
  D->write_data(callmode);
  D->write_data(iflag);
  D->write_data(m);
  D->write_data(n);
  D->write_data(ldfjac);
  D->write_data(liw);
  D->write_data(lw);

  for (int i = 0; i<n; i++)
    D->write_data(xc[i]);

  for (int i = 0; i<lw; i++)
    D->write_data(w[i]);
}

// evaluate function without recording the ir
lsqfun(ad_handle, iflag, m, n, xc, fvec, fjac, 
      ldfjac, iw, liw, w, lw);

if(callmode != nagad_evalonly) {
  // register outputs
  for (int i = 0; i<m; i++) {
    dco::alw::global_ir->register_variable(fvec[i])
       ;
    D->write_data(fvec[i]);
  }

  for (int i = 0; i<ldfjac*n; i++) {
    dco::alw::global_ir->register_variable(fjac[i])
       ;
    D->write_data(fjac[i]);
  }

  // insert adjoint (tape) callback
  dco::alw::global_ir->insert_callback(&
```
lsqfun_aiw_tape_callback, D); }
} /* lsqfun_aiw */

void lsqmon(void* ad_handle, 
    const int&, 
    const int&, 
    const dco::aiw::type[], 
    const dco::aiw::type[], 
    const dco::aiw::type[], 
    const int&, 
    const dco::aiw::type[], 
    const int&, 
    const int&, 
    const int&, 
    const int&, 
    int[], 
    const int&, 
    dco::aiw::type[], 
    const int&) {} 

int main() {
    dco::aiw::type fsumsq, eta, xtol, stepmx;
    std::vector<dco::aiw::type> fvec, fjac, s, v, w, x;
    Integer m, n, niter, nf, iw, iprint=-1;
    Integer maxcal, ldv, ldfjac, lw, liw;

    m = MMAX;
    n = TMAX;
    maxcal = 10000;
    eta = 0.01;
    xtol = 10e-6;
    stepmx = 100000.0;
    ldv = n;
    ldfjac = m;
    liw = 1;
    lw = 7*n + m*n + 2*m + n*n;
    fjac.resize(m*ldfjac);
    fvec.resize(m);
    x.resize(n);
    v.resize(ldv*n);
s.resize(n);
w.resize(lw);

scanf("%*[\n"]"); /* Skip heading in data file */
for (int i = 0; i < m; ++i) {
    scanf("%lf", &dco::value(st.y[i]));
    for (int j = 0; j < n; ++j)
        scanf("%lf", &dco::value(st.t[i][j]));
}

/* Set up the starting point */

x[0] = 0.5;
x[1] = 1.0;
x[2] = 1.5;

dco::aiw::global_ir = dco::aiw::ir_t::create();
for (int i = 0; i<m; i++)
    dco::aiw::global_ir->register_variable(st.y[i]);

int ifail = -1;
void *ad_handle = dco::aiw::create_config();
dco::aiw::adjoint_mode(ad_handle) = nagad_symbolic;

E04GB_A1W_F(ad_handle, m, n,
    E04HEV, lsqfun_aiw, lsqmon,
    iprint, maxcal, eta, xtol, stepmx, x.data(),
    fsumsq, fvec.data(), fjac.data(), ldfjac,
    s.data(), v.data(), ldv, niter, nf, &iw, liw,
    w.data(), lw, ifail);

dco::derivative(fsumsq) += 1.0;
dco::aiw::global_ir->interpret_adjoint();

std::cout << "Solution" << std::endl;
for (int i = 0; i<n; i++)
    std::cout << "x[" << i << "]= " << dco::value(x[i]) << std::endl;

std::cout << "\n\nNorm of residual = " << dco::value(fsumsq) << std::endl;
std::cout << "\n\nResiduals"<< std::endl;
for (int i = 0; i<m; i++)
    std::cout << "fvec[" << i << "]= " << dco::value(fvec[i]) << std::endl;

std::cout << 
    std::endl;
for (int i = 0; i<m; i++)
    std::cout << "dFsumsq/dY[" << i << "]= " << dco::
        derivative(st.y[i]) << std::endl;

dco::aiw::ir_t::remove(dco::aiw::global_ir);
dco::aiw::remove_config(ad_handle);
}

Listing 7: "e04gb adjoint algorithmic with dco/c++"
#include <vector>
#include <dco.hpp>
#include <nagad.hpp>

#define MMAX 15
#define TMAX 3

/* Define a user structure template to store data in lsqfun. */
template <typename T>
struct user
{
    T y[MMAX];
    T t[MMAX][TMAX];
};

static user<dco::aiw::type> st;

template <typename T>
void lsqfun(void* ad_handle,
    int& iflag,
    const int& m,
    const int& n,
    const T xc[],
    T fvec[],
    T fjac[],
    const int& ldjac,
    int iw[],
const int & liw,  
T w[],  
const int & lw) {

#define FJAC(I, J) fjac[(J) * ldfjac + (I)]

Integer i;  
T denom, dummy;

for (i = 0; i < m; ++i) {
    denom = xc[1] * st.t[i][1] + xc[2] * st.t[i][2];
    if (iflag != 1)
        fvec[i] = xc[0] + st.t[i][0] / denom - st.y[i];
    if (iflag != 0) {
        FJAC(i, 0) = 1.0;
        dummy = -1.0 / (denom * denom);
        FJAC(i, 1) = st.t[i][0] * st.t[i][1] * dummy;
        FJAC(i, 2) = st.t[i][0] * st.t[i][2] * dummy;
    }
}
} /* lsqfun */

void lsqmon(void* ad_handle,  
    const int &,  
    const int &,  
    const dco::aiw::type [],  
    const dco::aiw::type [],  
    const dco::aiw::type [],  
    const int &,  
    const dco::aiw::type [],  
    const int &,  
    const dco::aiw::type [],  
    const int &,  
    const int &,  
    const int &,  
    int [],  
    const int &,  
    dco::aiw::type [],  
    const int&) {}  

int main() {
    dco::aiw::type fsmsq, eta, xtol, stepmx;
std::vector<dco::a1w::type> fvec, fjac, s, v, w, x;
Integer m, n, niter, nf, iw, iprint=-1;
Integer maxcal, ldv, ldfjac, lw, liw;

m = MMMAX;
n = TMAX;
maxcal = 10000;
etau = 0.01;
xtol = 10e-6;
stepmx = 10000.0;
ldv = n;
ldfjac = m;
liw = 1;
lw = 7*n + m*n + 2*m + n*n;
fjac.resize(m*ldfjac);
fvec.resize(m);
x.resize(n);
v.resize(ldv*n);
s.resize(n);
w.resize(lw);

scanf(" %*[\n]"); /* Skip heading in data file */
for (int i = 0; i < m; ++i) {
    scanf("%lf", &dco::value(st.y[i]));
    for (int j = 0; j < n; ++j)
        scanf("%lf", &dco::value(st.t[i][j]));
}

/* Set up the starting point */
x[0] = 0.5;
x[1] = 1.0;
x[2] = 1.5;
dco::a1w::global_ir = dco::a1w::ir_t::create();
for (int i = 0; i<m; i++)
    dco::a1w::global_ir->register_variable(st.y[i]);
int ifail = -1;
void *ad_handle = dco::a1w::create_config();
Listing 8: "e04gb adjoint symbolic with dco/c++"

```cpp
#include <vector>
#include <dco.hpp>
#include <nagad.hpp>

#define MMAX 15
#define TMAX 3

/* Define a user structure template to store data in lsqfun. */
```
template <typename T>
struct user
{
    T y[MMAX];
    T t[MMAX][TMAX];
};

static user<dco::a1w::type> st;

template <typename T>
void lsqfun(void* ad_handle,
    int& iflag,
    const int& m,
    const int& n,
    const T xc[],
    T fvec[],
    T fjac[],
    const int& ldfjac,
    int iw[],
    const int& liw,
    T w[],
    const int& lw)
{
    #define FJAC(I, J) fjac[(J) * ldfjac + (I)]

    Integer i;
    T denom, dummy;

    for (i = 0; i < m; ++i) {
        denom = xc[1] * st.t[i][1] + xc[2] * st.t[i][2];
        if (iflag != 1)
            fvec[i] = xc[0] + st.t[i][0] / denom - st.y[i];

        if (iflag != 0) {
            FJAC(i, 0) = 1.0;
            dummy = -1.0 / (denom * denom);
            FJAC(i, 1) = st.t[i][0] * st.t[i][1] * dummy;
            FJAC(i, 2) = st.t[i][0] * st.t[i][2] * dummy;
        }
    }
    /* lsqfun */
}

template <typename T>
void lsqfun_dx(void* ad_handle,
int& iflag,
const int& m,
const int& n,
const T xc[],
T fvec[],
T fjac[],
const int& ldfjac,
int iw[],
const int& liw,
T w[],
const int& lw) {

#define FJAC(I, J) fjac[(J) * ldfjac + (I)]

Integer i;
T denom, dummy;

for (int i = 0; i < m; ++i) {
    denom = xc[1] * dco::value(st.t[i][1]) + xc[2] *
          dco::value(st.t[i][2]);
    if (iflag != 1)
        fvec[i] = xc[0] + dco::value(st.t[i][0]) /
                   denom - dco::value(st.y[i]);
    else
        FJAC(i, 0) = 1.0;
    dummy = -1.0 / (denom * denom);
    FJAC(i, 1) = dco::value(st.t[i][0]) * dco::
                   value(st.t[i][1]) * dummy;
    FJAC(i, 2) = dco::value(st.t[i][0]) * dco::
                   value(st.t[i][2]) * dummy;
}

} /* lsqfun_dx */

template <typename T>
void lsqfun_druser(void* ad_handle,
                   int& iflag,
                   const int& m,
                   const int& n,
                   const T xc[],
                   T fvec[],
                   T fjac[],
                   const int& ldfjac,
                   const int& liw,
                   T w[],
                   T fvec[],
                   T fjac[],
                   T fvec[],
                   T fjac[],
                   const int& ldfjac,
int iw[],
const int& liw,
T w[],
const int& lw) {

#define FJAC(I, J) fjac[(J) * ldfjac + (I)]

Integer i;
T denom, dummy;

for (int i = 0; i < m; ++i) {
    denom = dco::value(xc[1]) * st.t[i][1] + dco::value(xc[2]) * st.t[i][2];
    if (iflag != 1)
        fvec[i] = dco::value(xc[0]) + st.t[i][0] / denom - st.y[i];
    if (iflag != 0) {
        FJAC(i, 0) = 1.0;
        dummy = -1.0 / (denom * denom);
        FJAC(i, 1) = st.t[i][0] * st.t[i][1] * dummy;
        FJAC(i, 2) = st.t[i][0] * st.t[i][2] * dummy;
    }
}

} /* lsqfun_druser */

template <typename T>
void lsqfun_a1w(void* ad_handle,
    int& iflag,
    const int& m,
    const int& n,
    const T xc[],
    T fvec[],
    T fjac[],
    const int& ldfjac,
    int iw[],
    const int& liw,
    T w[],
    const int& lw) {

    int callmode = dco::a1w::callback_mode(ad_handle);
    if (callmode == nagad_evalonly)
lsqfun(ad_handle, iflag, m, n, xc, fvec, fjac, 
ldfjac, iw, liw, w, lw);
else if (callmode == nagad_dx)
  lsqfun_dx(ad_handle, iflag, m, n, xc, fvec, fjac, 
  ldfjac, iw, liw, w, lw);
else if (callmode==nagad_druser)
  lsqfun_druser(ad_handle, iflag, m, n, xc, fvec, fjac, 
  ldfjac, iw, liw, w, lw);
else if (callmode==nagad_full)
  lsqfun(ad_handle, iflag, m, n, xc, fvec, fjac, 
  ldfjac, iw, liw, w, lw);
}
/* lsqfun_a1w */

void lsqmon(void* ad_handle,
  const int &,
  const int &,
  const dco::a1w::type [],
  const dco::a1w::type [],
  const dco::a1w::type [],
  const int &,
  const dco::a1w::type [],
  const int &,
  const int &,
  const int &,
  int [],
  const int &,
  const int &,
  dco::a1w::type [],
  const int &) {}

int main() {

dco::a1w::type fsmsq, eta, xtol, stepmx;
std::vector<dco::a1w::type> fvec, fjac, s, v, w, x;
Integer m, n, niter, nf, iw, iprint=-1;
Integer maxcal, ldv, ldfjac, lw, liw;

m = MMAX;
n = TMAX;
maxcal = 10000;
eta = 0.01;
taxol = 10e-6;
stepmx = 100000.0;
ldv = n;
ldfjac = m;
liw = 1;
lw = 7*n + m*n + 2*m + n*n;

fjac.resize(m*ldfjac);
fvec.resize(m);
x.resize(n);
v.resize(ldv*n);
s.resize(n);
w.resize(lw);

scanf("%*[\n]"); /* Skip heading in data file */
for (int i = 0; i < m; ++i) {
    scanf("%lf", &dco::value(st.y[i]));
    for (int j = 0; j < n; ++j)
        scanf("%lf", &dco::value(st.t[i][j]));
}

/* Set up the starting point */
x[0] = 0.5;
x[1] = 1.0;
x[2] = 1.5;

dco::alw::global_ir = dco::alw::ir_t::create();
for (int i = 0; i < m; i++)
    dco::alw::global_ir->register_variable(st.y[i]);

int ifail = -1;
void *ad_handle = dco::alw::create_config();
dco::alw::adjoint_mode(ad_handle) = nagad_symbolic;

E04GB_A1W_F(ad_handle, m, n,
    E04HEV, lsqfun_alw, lsqmon,
    iprint, maxcal, eta, xtol, stepmx, x.data(),
    fsumsq, fvec.data(), fjac.data(), ldfjac,
    s.data(), v.data(), ldv, niter, nf, &iw, liw,
    w.data(), lw, ifail);

dco::derivative(fsumsq) += 1.0;
dco::alw::global_ir->interpret_adjoint();

std::cout << "Solution" << std::endl;
for (int i = 0; i<n; i++)
    std::cout << "x[" << i << "]= " << dco::value(x[i]) << std::endl;
std::cout << "\n\nNorm of residual = " << dco::value(fsumsq) << std::endl;
for (int i = 0; i<m; i++)
    std::cout << "fvec[" << i << "]= " << dco::value(fvec[i]) << std::endl;
std::cout << "\n\nResiduals " << std::endl;
for (int i = 0; i<m; i++)
    std::cout << "dFsumsq/dY\[" << i << "]= " << dco::derivative(st.y[i]) << std::endl;
dco::aiw::ir_t::remove(dco::aiw::global_ir);
dco::aiw::remove_config(ad_handle);
}

References

[2] https://nag.co.uk/content/algorithmic-differentiation-software