Why do we Need Adjoint Routines

July 30, 2018

Contents

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.

```
29  for (int i = 0; i < m; ++i){
30    x[i] = double(i)/(m-1);
31    y[i] = exp(x[i]);
32  }
```

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

```
60  e01baf_(m, x.data(), y.data(), lamda.data(), c.data
       (), lck, wrk.data(), lwrk, ifail);
61  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.

```
90  for (int i = 0; i < m; ++i) {
91    x[i] += h;
92    e01baf_(m, x.data(), y.data(), lamda.data(), c.
         data(), lck, wrk.data(), lwrk, ifail);
93    e02bbf_(lck, lamda.data(), c.data(), xarg, fith,
         ifail);
94    x[i] -= h;
95
96    if (m < 8) {
97      printf("    %" NAG_IFMT "  %13.8f \n", i + 1, (
           fith-fit)/h);
98    }
99  }
```

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 (`e01baf`) and evaluation (`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:

```
66  /* E01BA_A1W_F.
67   * Adjoint of interpolating function, cubic spline
       interpolant, one
68   * variable
69   */
70  E01BA_A1W_F(ad_config,m, x.data(), y.data(), lamda.
       data(), c.data(), lck, wrk.data(), lwrk, ifail);
71
72  for(int j=0;j<lck;++j) {
73    dco::a1w::global_ir->register_output_variable(
         lamda[j]);
74    dco::a1w::global_ir->register_output_variable(c[j
         ]);
75  }
76
77  /* E02BB_A1W_F.
78   * Adjoint of evaluation of fitted cubic spline,
       function only
79   */
80  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

```
72  for(int j=0;j<lck;++j) {
73    dco::a1w::global_ir->register_output_variable(
         lamda[j]);
74    dco::a1w::global_ir->register_output_variable(c[j
         ]);
75  }
```

No additional calls of adjoint routines are performed to compute these derivatives. With finite difference we would need to perturb each entry of `lamda` and `c` to compute these derivatives

```
102   for (int i = 0; i < lck; ++i) {
103     lamdah[i] += h;
104     e02bbf_(lck, lamdah.data(), ch.data(), xarg, fith
          , ifail);
105     lamdah[i] -= h;
106
107     ch[i] += h;
108     e02bbf_(lck, lamdah.data(), ch.data(), xarg,
          fith1, ifail);
109     ch[i] -=h;
110     if (m < 8) {
111       printf("    %" NAG_IFMT "  %13.8f   %13.8f\n",
            i + 1,
112        (fith-fit)/h,  (fith1-fit)/h);
113     }
114   }
```

Therefore 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)

$$y := f(x) \text{ with } f : \mathbb{R}^n \to \mathbb{R}^m,$$

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 $f^{(1)} : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^m$. The symbol ∇_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 $O(n)\mathbf{cost}(f)$, which corresponds to n evaluations of the tangent model. The adjoint model is defined as

$$\begin{pmatrix} x_{(1)} \\ y_{(1)} \end{pmatrix} := f_{(1)}(x, x_{(1)}, y_{(1)}) = \begin{pmatrix} x_{(1)} + \left[\nabla_x f(x)^T\right] y_{(1)} \\ 0 \end{pmatrix}$$

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 $O(m)\cdot\mathbf{cost}(f)$. 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 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 hand-writing 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

```
51  void *ad_config=dco::a1w::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_a1w_f` .

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

```
52  dco::a1w::adjoint_mode(ad_config) =
        nagad_algorithmic;
```

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

```
54  dco::a1w::global_ir = dco::a1w::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)

```
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

```
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

```
59  for (int i = 0; i < m; ++i)
60    dco::a1w::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.

```
66  /* E01BA_A1W_F.
67   * Adjoint of interpolating function, cubic spline
        interpolant, one
68   * variable
69   */
70  E01BA_A1W_F(ad_config,m, x.data(), y.data(), lamda.
       data(), c.data(), lck, wrk.data(), lwrk, ifail);
71
72  for(int j=0;j<lck;++j) {
73    dco::a1w::global_ir->register_output_variable(
         lamda[j]);
74    dco::a1w::global_ir->register_output_variable(c[j
         ]);
75  }
76
```

```
77  /* E02BB_A1W_F.
78   * Adjoint of evaluation of fitted cubic spline,
         function only
79   */
80  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 `fit` to 1

```
100  dco::derivative(fit) = 1.0;
```

Similar to `dco::value`, `dco::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

```
101  dco::a1w::global_ir->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 ::derivative` function. In our example we are interested in adjoints stored in array `x`, therefore

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

prints the desired adjoint information to stdout.

10. Free internal memory:

```
119  dco::a1w::remove_config(ad_config);
```

11. Free `ad_handle`:

```
120  dco::a1w::ir_t::remove(dco::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 `ad_handle` must be set to `nagad_symbolic` by the corresponding function call.

```
1 dco::a1w::adjoint_mode(ad_handle) = 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

y	t_1	t_2	t_3
0.14	1.0	15.0	1.0
0.18	2.0	14.0	2.0
0.22	3.0	13.0	3.0
0.25	4.0	12.0	4.0
0.29	5.0	11.0	5.0
0.32	6.0	10.0	6.0
0.35	7.0	9.0	7.0
0.39	8.0	8.0	8.0
0.37	9.0	7.0	7.0
0.58	10.0	6.0	6.0
0.73	11.0	5.0	5.0
0.96	12.0	4.0	4.0
1.34	13.0	3.0	3.0
2.10	14.0	2.0	2.0
4.39	15.0	1.0	1.0

Table 1: Table 1

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.

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.

```
160  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

```
162  //store inputs
163  D->write_data(iflag);
164  D->write_data(m);
165  D->write_data(n);
166  D->write_data(ldfjac);
167  D->write_data(liw);
168  D->write_data(lw);
169
170  for (int i=0; i<n; i++)
171    D->write_data(xc[i]);
172
173  for (int i=0; i<lw; i++)
174      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

```
176  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

```
178  //register outputs
179  for (int i = 0; i<m; i++) {
180    dco::a1w::global_ir->register_variable(fvec[i]);
181    D->write_data(fvec[i]);
182  }
183
184  for (int i = 0; i<ldfjac*n; i++) {
185    dco::a1w::global_ir->register_variable(fjac[i]);
186    D->write_data(fjac[i]);
187  }
```

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

```
190  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:

```
56 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:

```
58   int const& iflag = D->read_data<int>();
59   int const& m = D->read_data<int>();
60   int const& n = D->read_data<int>();
61   int const& ldfjac = D->read_data<int>();
62   int const& liw = D->read_data<int>();
63   int const& lw = D->read_data<int>();

71   for (int i=0; i<n; i++)
72     xc[i] = D->read_data<dco::a1w::type>();
73
74   for (int i=0; i<lw; i++)
75     w[i] = D->read_data<dco::a1w::type>();
76
77   for (int i = 0; i<m; i++) {
78     fvec[i] = D->read_data<dco::a1w::type>();
79     fveca[i] = dco::derivative(fvec[i]);
80   }
81
82   for (int i = 0; i<ldfjac*n; i++) {
83     fjac[i] = D->read_data<dco::a1w::type>();
84     fjaca[i] = dco::derivative(fjac[i]);
85   }
```

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:

```
77   for (int i = 0; i<m; i++) {
78     fvec[i] = D->read_data<dco::a1w::type>();
```

```
79      fveca[i] = dco::derivative(fvec[i]);
80    }
81
82    for (int i = 0; i<ldfjac*n; i++) {
83      fjac[i] = D->read_data<dco::a1w::type>();
84      fjaca[i] = dco::derivative(fjac[i]);
85    }
```

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

```
99    // adjoint of lsqfun
100   for (int i = m-1; i>=0; i--){
101     denom = dco::value(xc[1]) * dco::value(st.t[i
          ][1]) + dco::value(xc[2]) * dco::value(st.t[i
          ][2]);
102     denoma = 0.0;
103
104     if (iflag !=0) {
105       dummy = -1/(denom*denom);
106       dummya = 0.0;
107
108       sta.t[i][0] += FJACA(i,2)*dco::value(st.t[i
            ][2])*dummy;
109       sta.t[i][2] += FJACA(i,2)*dco::value(st.t[i
            ][0])*dummy;
110       dummya      += FJACA(i,2)*dco::value(st.t[i
            ][0])*dco::value(st.t[i][2]);
111
112       sta.t[i][0] += FJACA(i,1)*dco::value(st.t[i
            ][1])*dummy;
113       sta.t[i][1] += FJACA(i,1)*dco::value(st.t[i
            ][0])*dummy;
114       dummya      += FJACA(i,1)*dco::value(st.t[i
            ][0])*dco::value(st.t[i][1]);
115
116       denoma      += dummya*2.0/(denom*denom*denom);
117     }
118     if (iflag != 1) {
119       xca[0]      += fveca[i];
120       sta.t[i][0] += fveca[i]/denom;
121       denoma      += -fveca[i]*dco::value(st.t[i][0])
            /(denom*denom);
122       sta.y[i]    += -fveca[i];
123     }
124
125     xca[1]      += denoma*dco::value(st.t[i][1]);
```

```
126     sta.t[i][1] += denoma*xca[1];
127     xca[2]      += denoma*dco::value(st.t[i][2]);
128     sta.t[i][2] += denoma*xca[2];
129   }
```

The adjoint increments are stored in arrays `xca` and `sta.y`.

4. Increment the adjoints of inputs:

```
131   for (int i = 0; i<n; i++)
132     dco::derivative(xc[i]) += xca[i];
133
134   for (int i = 0; i<m; i++)
135     dco::derivative(st.y[i]) += sta.y[i];
```

Once we get our `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 `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 [1]). Specifically it does not allow to compute the derivative with respect to starting point of the optimization. In Listing 6 we implemented the same example using symbolic adjoints. Below we describe this code in more detail.

The user-supplied function of routine `e04gbf` evaluates

$$(z, g) = \Big(f(x, p), \nabla_x f(x, p)\Big).$$

To use the symbolic adjoint of routine `e04gbf` the companion callback must support computation of the following for modes:

1. Function evaluation only, i.e.,

$$\Big(f(x, p), \nabla_x f(x, p)\Big).$$

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

$$x_{(1)} += \big[\nabla_x f(x, p)\big]^T z_{(1)} + \big[\nabla_x^2 f(x, p)\big]^T g_{(1)}.$$

3. Function evaluation and adjoint computation w.r.t. parameter p, i.e.,

$$p_{(1)} += \left[\nabla_p f(x,p)\right]^T z_{(1)} + \left[\nabla_p^2 f(x,p)\right]^T g_{(1)}.$$

4. Function evaluation and computation of all adjoints.

The required mode is stored in the callback data object

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

and then passed over to the companion callback

```
58   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.

```
132   if (callmode == nagad_dx || callmode == nagad_full)
          {
133    for (int i = 0; i<n; i++)
134      dco::derivative(xc[i]) += xca[i];
135   }
136   if (callmode == nagad_druser || callmode ==
         nagad_full) {
137     for (int i = 0; i<m; i++)
138       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++.

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

```
144   else if (callmode==nagad_dx)
145     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.

```
76      for (int i = 0; i < m; ++i) {
77        denom = xc[1] *dco::value(st.t[i][1]) + xc[2] *
            dco::value(st.t[i][2]);
78        if (iflag != 1)
79        fvec[i] = xc[0] + dco::value(st.t[i][0]) /
            denom - dco::value(st.y[i]);
80
81        if (iflag != 0) {
82        FJAC(i, 0) = 1.0;
83        dummy = -1.0 / (denom * denom);
84        FJAC(i, 1) = dco::value(st.t[i][0]) * dco::
            value(st.t[i][1]) * dummy;
85        FJAC(i, 2) = dco::value(st.t[i][0]) * dco::
            value(st.t[i][2]) * dummy;
86        }
87      }
```

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

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

110   for (int i = 0; i < m; ++i) {
111     denom = dco::value(xc[1]) * st.t[i][1] + dco::
          value(xc[2]) * st.t[i][2];
112     if (iflag != 1)
113       fvec[i] =dco::value(xc[0]) + st.t[i][0] / denom
            - st.y[i];
114
115     if (iflag != 0) {
116       FJAC(i, 0) = 1.0;
```

```
117       dummy = -1.0 / (denom * denom);
118       FJAC(i, 1) = st.t[i][0] * st.t[i][1] * dummy;
119       FJAC(i, 2) = st.t[i][0] * st.t[i][2] * dummy;
120     }
```

In case all adjoints are required (`nagad_full`) the original overloaded function is executed

```
148   else if (callmode==nagad_full)
149     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"

```
1 #include <vector>
2 #include <dco.hpp>
3 #include <nagad.hpp>
4
5 #define MMAX 7
6
7 double getTime(void){
8     struct timespec tv;
9
10    if(clock_gettime(CLOCK_REALTIME, &tv) != 0)
          return 0;
11
12    return (((double) tv.tv_sec) + (double) (tv.
          tv_nsec / 1000000000.0));
13 }
14
15 template<typename T>
16 T initialize(int &m, int &lck, int& lwrk, std::vector
      <T> &x, std::vector<T> &y,
17          std::vector<T> &lamda, std::vector<T> &c, std
                ::vector<T> &wrk){
18   lck = m + 4;
19   lwrk = 6*m + 16;
20
21   x.resize(m);
22   y.resize(m);
23   lamda.resize(lck);
24   c.resize(lck);
```

```
25   wrk.resize(lwrk);
26
27   for (int i = 0; i < m; ++i){
28     x[i] = double(i)/(m-1);
29     y[i] = exp(x[i]);
30   }
31
32   return (x[m/2] + x[m/2+1])/2.0;
33 }
34
35 int main(int argc, char **argv)
36 {
37   Integer exit_status = 0, m = MMAX, lck=MMAX+4, lwrk
       =0;
38   Integer ifail=-1;
39   std::vector<dco::a1w::w_rtype> lamda, c, wrk, x, y;
40
41   dco::a1w::w_rtype fit, xarg;
42
43   if (argc > 1){
44     m = atoi(argv[1]);
45   }
46
47   /* Initialize spline */
48
49   printf("nag_1d_spline_interpolant (e01bac) Example
       Program Results\n");
50
51   void *ad_config=dco::a1w::create_config();
52   dco::a1w::adjoint_mode(ad_config) =
       nagad_algorithmic;
53
54   dco::a1w::global_ir = dco::a1w::ir_t::create();
55
56
57   xarg = initialize(m,lck,lwrk,x,y,lamda,c,wrk);
58
59   for (int i = 0; i < m; ++i)
60     dco::a1w::global_ir->register_variable(x[i]);
61
62
63
64   double  start = getTime();
65
66   /* E01BA_A1W_F.
67    * Adjoint of interpolating function, cubic spline
```

```
      interpolant, one
68    * variable
69    */
70    E01BA_A1W_F(ad_config,m, x.data(), y.data(), lamda.
         data(), c.data(), lck, wrk.data(), lwrk, ifail);
71
72    for(int j=0;j<lck;++j) {
73      dco::a1w::global_ir->register_output_variable(
           lamda[j]);
74      dco::a1w::global_ir->register_output_variable(c[j
           ]);
75    }
76
77    /* E02BB_A1W_F.
78     * Adjoint of evaluation of fitted cubic spline,
         function only
79     */
80    E02BB_A1W_F(ad_config, lck, lamda.data(), c.data(),
         xarg, fit, ifail);
81
82    if (m < 8) {
83      printf("\nNumber of distinct knots = %" NAG_IFMT
           "\n\n", m - 2);
84      printf("Distinct knots located at \n\n");
85      for (int j = 3; j < m + 1; j++)
86        printf("%8.4f%s", dco::value(lamda[j]), (j - 3)
             % 5 == 4
87          || j == m ? "\n" : " ");
88      printf("\n\n    J    B-spline coeff c\n\n");
89      for (int j = 0; j < m; ++j)
90        printf("    %" NAG_IFMT "  %13.4f\n", j + 1,
             dco::value(c[j]));
91    }
92
93    printf("\n    J       Abscissa            Ordinate
                 Spline\n\n");
94
95    if (m < 8) {
96    printf("    %13.4f       %13.4f\n",
97       dco::value(xarg),  dco::value(fit));
98    }
99    dco::a1w::global_ir->register_output_variable(fit);
100   dco::derivative(fit) = 1.0;
101   dco::a1w::global_ir->interpret_adjoint();
102
103   double end = getTime();
```

```
104
105
106   if (m < 8) {
107     printf("\n\n    J          x_a1s\n\n");
108     for (int j = 0; j < m; ++j)
109       printf("    %" NAG_IFMT "  %13.8f \n", j + 1,
              dco::derivative(x[j]));
110
111
112     printf("\n\n    J          lamda_a1s       c_a1s \
          n\n");
113     for (int j = 0; j < lck; ++j)
114       printf("    %" NAG_IFMT "  %13.8f   %13.8f\n",
              j + 1,
115        dco::derivative(lamda[j]), dco::derivative(c[j
             ]));
116   }
117   std::cout << "time " << end -start << std::endl;
118
119   dco::a1w::remove_config(ad_config);
120   dco::a1w::ir_t::remove(dco::a1w::global_ir);
121
122   /* Free memory allocated */
123   return exit_status;
124 }
```

Listing 2: "Spline finite differences "

```
1
2 #include <vector>
3 #include <iostream>
4 #include <nagmk26.h>
5 #include <cmath>
6
7 #define MMAX 7
8
9 double getTime(void){
10     struct timespec tv;
11
12     if(clock_gettime(CLOCK_REALTIME, &tv) != 0)
          return 0;
13
14     return (((double) tv.tv_sec) + (double) (tv.
          tv_nsec / 1000000000.0));
15 }
16
17 template<typename T>
```

```
18 T initialize(int &m, int &lck, int& lwrk, std::vector
     <T> &x, std::vector<T> &y,
19         std::vector<T> &lamda, std::vector<T> &c, std
              ::vector<T> &wrk){
20   lck = m + 4;
21   lwrk = 6*m + 16;
22
23   x.resize(m);
24   y.resize(m);
25   lamda.resize(lck);
26   c.resize(lck);
27   wrk.resize(lwrk);
28
29   for (int i = 0; i < m; ++i){
30     x[i] = double(i)/(m-1);
31     y[i] = exp(x[i]);
32   }
33
34   return (x[m/2] + x[m/2+1])/2.0;
35 }
36
37 int main(int argc, char **argv)
38 {
39   Integer exit_status = 0, m = MMAX, lck=MMAX+4, lwrk
        =0;
40   Integer ifail=-1;
41   std::vector<double> x, y, lamda, c, wrk;
42   double fith, fith1;
43   double fit, xarg;
44   double h = 1e-8;
45
46   if (argc > 1){
47     m = atoi(argv[1]);
48   }
49
50   /* Initialize spline */
51
52   printf("nag_1d_spline_interpolant (e01bac) Example
        Program Results\n");
53
54
55   xarg = initialize(m,lck,lwrk,x,y,lamda,c,wrk);
56
57
58   double  start = getTime();
59
```

```
60  e01baf_(m, x.data(), y.data(), lamda.data(), c.data
       (), lck, wrk.data(), lwrk, ifail);
61  e02bbf_(lck, lamda.data(), c.data(), xarg, fit,
       ifail);
62
63  if (m < 8) {
64    printf("\nNumber of distinct knots = %" NAG_IFMT
         "\n\n", m - 2);
65    printf("Distinct knots located at \n\n");
66    for (int j = 3; j < m + 1; j++)
67      printf("%8.4f%s", lamda[j], (j - 3) % 5 == 4
68       || j == m ? "\n" : " ");
69    printf("\n\n    J    B-spline coeff c\n\n");
70    for (int j = 0; j < m; ++j)
71      printf("    %" NAG_IFMT "  %13.4f\n", j + 1, c[
           j]);
72  }
73
74  printf("\n    J       Abscissa            Ordinate
                Spline\n\n");
75
76  if (m < 8) {
77  printf("    %13.4f       %13.4f\n",
78     xarg,  fit);
79  }
80
81  /* Store lamda and c as they destroyed inside the
       next loop */
82  std::vector<double> lamdah(lck), ch(lck);
83  for (int i = 0; i < lck; ++i) {
84    lamdah[i] = lamda[i];
85    ch[i] = c[i];
86  }
87  if (m < 8) {
88    printf("\n\n    J         x_a1s\n\n");
89  }
90  for (int i = 0; i < m; ++i) {
91    x[i] += h;
92    e01baf_(m, x.data(), y.data(), lamda.data(), c.
         data(), lck, wrk.data(), lwrk, ifail);
93    e02bbf_(lck, lamda.data(), c.data(), xarg, fith,
         ifail);
94    x[i] -= h;
95
96    if (m < 8) {
97      printf("    %" NAG_IFMT "  %13.8f \n", i + 1, (
```

```
            fith-fit)/h);
98      }
99    }
100
101   printf("\n\n    J          lamda_a1s        c_a1s \n\
          n");
102   for (int i = 0; i < lck; ++i) {
103     lamdah[i] += h;
104     e02bbf_(lck, lamdah.data(), ch.data(), xarg, fith
            , ifail);
105     lamdah[i] -= h;
106
107     ch[i] += h;
108     e02bbf_(lck, lamdah.data(), ch.data(), xarg,
            fith1, ifail);
109     ch[i] -=h;
110     if (m < 8) {
111       printf("    %" NAG_IFMT "  %13.8f   %13.8f\n",
              i + 1,
112         (fith-fit)/h,  (fith1-fit)/h);
113     }
114   }
115   double end = getTime();
116   std::cout << "time " << end -start << std::endl;
117
118   /* Free memory allocated */
119
120
121
122   return exit_status;
123 }
```

Listing 3: "f07ca_a1w_f symbolic adjoint"

```
1 #include <dco.hpp>
2 #include <nagad.hpp>
3
4 int main() {
5   int n = 2, nrhs = 1, ldb = 2, ifail;
6   dco::a1w::w_rtype dl[1] = {0}, du[1] = {0}, d[2] =
        {1, 2};
7
8   // declare in addition a pure input variable
9   dco::a1w::w_rtype b_in[2] = {1, 1}, b[2];
10
11  dco::a1w::global_ir = dco::a1w::ir_t::create();
12  dco::a1w::adjoint_mode(ad_handle) = nagad_symbolic;
```

```
13
14   // register pure input here
15   dco::a1w::global_ir->register_variable(b_in, 2);
16   // copy into in/out variable
17   b[0] = b_in[0];
18   b[1] = b_in[1];
19
20   void *config = dco::a1w::create_config();
21
22   // b gets overwritten
23   F07CA_A1W_F(config, n, nrhs, dl, d, du, b, ldb,
        ifail);
24
25   // set adjoint of output
26   dco::derivative(b[0]) += 1.0;
27   dco::derivative(b[1]) += 1.0;
28
29   dco::a1w::global_ir->interpret_adjoint();
30
31   // read adjoint of input
32   std::cout << "db/db = (" << dco::derivative(b_in
        [0]) << " " << dco::derivative(b_in[1]) << ") "
        << std::endl;
33
34   dco::a1w::remove_config(config);
35   dco::a1w::ir_t::remove(dco::a1w::global_ir);
36   return 0;
37 }
```

Listing 4: "e04gb finite differences"

```
1 #include <dco.hpp>
2 #include <nagad.hpp>
3
4
5 #define MMAX 15
6 #define TMAX 3
7
8 /* Define a user structure template to store data in
      lsqfun. */
9 struct user
10 {
11   dco::a1w::type y[MMAX];
12   dco::a1w::type t[MMAX][TMAX];
13 };
14
15 static user st;
```

```
16
17 void lsqfun(void* config,
18       int& iflag,
19       const int& m,
20       const int& n,
21       const dco::a1w::type xc[],
22       dco::a1w::type fvec[],
23       dco::a1w::type fjac[],
24       const int& ldfjac,
25       int iw[],
26       const int& liw,
27       dco::a1w::type w[],
28       const int& lw) {
29
30
31 #define FJAC(I, J) fjac[(J) *ldfjac + (I)]
32
33   Integer i;
34   dco::a1w::type denom, dummy;
35
36
37   for (i = 0; i < m; ++i) {
38     denom = xc[1] * st.t[i][1] + xc[2] * st.t[i][2];
39     if (iflag != 1)
40       fvec[i] = xc[0] + st.t[i][0] / denom - st.y[i];
41
42     if (iflag != 0) {
43       FJAC(i, 0) = 1.0;
44       dummy = -1.0 / (denom * denom);
45       FJAC(i, 1) = st.t[i][0] * st.t[i][1] * dummy;
46       FJAC(i, 2) = st.t[i][0] * st.t[i][2] * dummy;
47     }
48   }
49 } /* lsqfun */
50
51
52 void lsqmon(void* config,
53       const int&,
54       const int&,
55       const dco::a1w::type[],
56       const dco::a1w::type[],
57       const dco::a1w::type[],
58       const int&,
59       const dco::a1w::type[],
60       const int&,
61       const int&,
```

```
62        const int&,
63        int[],
64        const int&,
65        dco::a1w::type[],
66        const int&) {}
67
68
69  int main() {
70
71    dco::a1w::type fsumsq, *fvec = 0, *fjac = 0, *s =
          0, *v = 0, *w = 0, *x = 0;
72    Integer m, n, niter, nf, iw, iprint=-1;
73    Integer maxcal, ldv, ldfjac, lw, liw;
74    dco::a1w::type eta, xtol, stepmx;
75    dco::a1w::type fsumsqh;
76    double h = 1e-8;
77
78    m = 15;
79    n = 3;
80    maxcal = 10000;
81    eta = 0.01;
82    xtol = 10e-6;
83    stepmx = 100000.0;
84    ldv = n;
85    ldfjac = m;
86    liw = 1;
87    lw = 7*n + m*n + 2*m + n*n;
88
89    fjac = new dco::a1w::type[m*ldfjac];
90    fvec = new dco::a1w::type[m];
91    x = new dco::a1w::type[n];
92    v = new dco::a1w::type[ldv*n];
93    s = new dco::a1w::type[n];
94    w = new dco::a1w::type[lw];
95
96
97    scanf(" %*[^\n]"); /* Skip heading in data file */
98    for (int i = 0; i < m; ++i) {
99      scanf("%lf", &dco::value(st.y[i]));
100     for (int j = 0; j < n; ++j)
101       scanf("%lf", &dco::value(st.t[i][j]));
102   }
103
104
105   /* Set up the starting point */
106   x[0] = 0.5;
```

```
107   x[1] = 1.0;
108   x[2] = 1.5;
109
110   dco::a1w::global_ir = dco::a1w::ir_t::create();
111
112   // for (int i = 0; i<m; i++)
113   //   dco::a1w::global_ir->register_variable(st.y[i
          ]);
114
115
116   int ifail = -1;
117   void *config = dco::a1w::create_config();
118
119   E04GB_A1W_F(config, m, n,
120         E04HEV, lsqfun, lsqmon,
121         iprint, maxcal, eta, xtol, stepmx, x,
122         fsumsq, fvec, fjac, ldfjac,
123         s, v, ldv, niter, nf, &iw, liw, w, lw, ifail)
              ;
124
125   // dco::derivative(fsumsq) += 1.0;
126   // dco::a1w::global_ir->interpret_adjoint();
127
128   std::cout << "Solution" << std::endl;
129   for (int i = 0; i<n; i++)
130     std::cout << "x[" << i << "]= " << dco::value(x[i
          ]) << std::endl;
131
132   std::cout << "\n\nNorm of residual = " << dco::
          value(fsumsq) << std::endl;
133
134   std::cout << "\n\nResiduals"<< std::endl;
135   for (int i = 0; i<m; i++)
136     std::cout << "fvec[" << i << "]= " << dco::value(
          fvec[i]) << std::endl;
137
138   std::cout << "\n\n dFsumsq/dY FD "<< std::endl;
139   for (int i = 0; i<m; i++) {
140     x[0] = 0.5;
141     x[1] = 1.0;
142     x[2] = 1.5;
143
144     st.y[i] += h;
145     E04GB_A1W_F(config, m, n,
146     E04HEV, lsqfun, lsqmon,
147     iprint, maxcal, eta, xtol, stepmx, x,
```

```
148     fsumsqh, fvec, fjac, ldfjac,
149     s, v, ldv, niter, nf, &iw, liw, w, lw, ifail);
150
151     std::cout << "dFsumsq/dY[" << i << "]= " << dco::
            value(fsumsqh - fsumsq)/h << std::endl;
152     st.y[i] -= h;
153   }
154   delete [] fjac;
155   delete [] fvec;
156   delete [] x;
157   delete [] v;
158   delete [] s;
159   delete [] w;
160 }
```

Listing 5: "e04gb adjoint algorithmic"

```
1 #include <vector>
2 #include <dco.hpp>
3 #include <nagad.hpp>
4
5
6
7 #define MMAX 15
8 #define TMAX 3
9
10 /* Define a user structure template to store data in
       lsqfun. */
11 template <typename T>
12 struct user
13 {
14   T y[MMAX];
15   T t[MMAX][TMAX];
16 };
17
18 static user<dco::a1w::type> st;
19
20 template <typename T>
21 void lsqfun(void* ad_handle,
22       int& iflag,
23       const int& m,
24       const int& n,
25       const T xc[],
26       T fvec[],
27       T fjac[],
28       const int& ldfjac,
29       int iw[],
```

```
30        const int& liw,
31        T w[],
32        const int& lw) {
33
34
35  #define FJAC(I, J) fjac[(J) *ldfjac + (I)]
36
37    Integer i;
38    T denom, dummy;
39
40
41    for (i = 0; i < m; ++i) {
42      denom = xc[1] * st.t[i][1] + xc[2] * st.t[i][2];
43      if (iflag != 1)
44        fvec[i] = xc[0] + st.t[i][0] / denom - st.y[i];
45
46      if (iflag != 0) {
47        FJAC(i, 0) = 1.0;
48        dummy = -1.0 / (denom * denom);
49        FJAC(i, 1) = st.t[i][0] * st.t[i][1] * dummy;
50        FJAC(i, 2) = st.t[i][0] * st.t[i][2] * dummy;
51      }
52    }
53  } /* lsqfun */
54
55
56  void lsqfun_a1w_tape_callback(dco::a1w::
       external_adjoint_object_t *D) {
57    //read checkpointed data
58    int const& iflag = D->read_data<int>();
59    int const& m = D->read_data<int>();
60    int const& n = D->read_data<int>();
61    int const& ldfjac = D->read_data<int>();
62    int const& liw = D->read_data<int>();
63    int const& lw = D->read_data<int>();
64
65  #define FJACA(I, J) fjaca[(J) *ldfjac + (I)]
66
67    int iflag1 = iflag;
68    std::vector<dco::a1w::type> xc(n), fvec(m), fjac(
        ldfjac*n), w(lw);
69    std::vector<double> fveca(m), fjaca(ldfjac*n);
70
71    for (int i=0; i<n; i++)
72      xc[i] = D->read_data<dco::a1w::type>();
73
```

```
74  for (int i=0; i<lw; i++)
75    w[i] = D->read_data<dco::a1w::type>();
76
77  for (int i = 0; i<m; i++) {
78    fvec[i] = D->read_data<dco::a1w::type>();
79    fveca[i] = dco::derivative(fvec[i]);
80  }
81
82  for (int i = 0; i<ldfjac*n; i++) {
83    fjac[i] = D->read_data<dco::a1w::type>();
84    fjaca[i] = dco::derivative(fjac[i]);
85  }
86
87
88  std::vector<double> xca(n);
89  double denoma = 0, dummya = 0, denom, dummy;
90  user<double> sta;
91
92  for (int i = 0; i<n; i++) xca[i] = 0.0;
93  for (int i = 0; i<m; i++) sta.y[i] = 0.0;
94  for (int i = 0; i<m; i++) {
95    for (int j = 0; j<n; j++)
96      sta.t[i][j] = 0.0;
97  }
98
99  // adjoint of lsqfun
100 for (int i = m-1; i>=0; i--){
101   denom = dco::value(xc[1]) * dco::value(st.t[i
        ][1]) + dco::value(xc[2]) * dco::value(st.t[i
        ][2]);
102   denoma = 0.0;
103
104   if (iflag !=0) {
105     dummy = -1/(denom*denom);
106     dummya = 0.0;
107
108     sta.t[i][0] += FJACA(i,2)*dco::value(st.t[i
          ][2])*dummy;
109     sta.t[i][2] += FJACA(i,2)*dco::value(st.t[i
          ][0])*dummy;
110     dummya      += FJACA(i,2)*dco::value(st.t[i
          ][0])*dco::value(st.t[i][2]);
111
112     sta.t[i][0] += FJACA(i,1)*dco::value(st.t[i
          ][1])*dummy;
113     sta.t[i][1] += FJACA(i,1)*dco::value(st.t[i
```

```
          ][0])*dummy;
114       dummya      += FJACA(i,1)*dco::value(st.t[i
          ][0])*dco::value(st.t[i][1]);
115
116       denoma      += dummya*2.0/(denom*denom*denom);
117     }
118     if (iflag != 1) {
119       xca[0]      += fveca[i];
120       sta.t[i][0] += fveca[i]/denom;
121       denoma      += -fveca[i]*dco::value(st.t[i][0])
          /(denom*denom);
122       sta.y[i]    += -fveca[i];
123     }
124
125     xca[1]      += denoma*dco::value(st.t[i][1]);
126     sta.t[i][1] += denoma*xca[1];
127     xca[2]      += denoma*dco::value(st.t[i][2]);
128     sta.t[i][2] += denoma*xca[2];
129   }
130
131   for (int i = 0; i<n; i++)
132     dco::derivative(xc[i]) += xca[i];
133
134   for (int i = 0; i<m; i++)
135     dco::derivative(st.y[i]) += sta.y[i];
136
137   for (int i = 0; i<m; i++) {
138     for (int j = 0; j<n; j++)
139       dco::derivative(st.t[i][j]) += sta.t[i][j];
140   }
141
142 } /* lsqfun_a1w_tape_callback */
143
144
145
146 template <typename T>
147 void lsqfun_a1w(void* ad_handle,
148       int& iflag,
149       const int& m,
150       const int& n,
151       const T xc[],
152       T fvec[],
153       T fjac[],
154       const int& ldfjac,
155       int iw[],
156       const int& liw,
```

```
157       T w[],
158       const int& lw) {
159
160   dco::a1w::external_adjoint_object_t *D = dco::a1w::
        global_ir->create_callback_object<dco::a1w::
        external_adjoint_object_t>();
161
162   //store inputs
163   D->write_data(iflag);
164   D->write_data(m);
165   D->write_data(n);
166   D->write_data(ldfjac);
167   D->write_data(liw);
168   D->write_data(lw);
169
170   for (int i=0; i<n; i++)
171     D->write_data(xc[i]);
172
173   for (int i=0; i<lw; i++)
174       D->write_data(w[i]);
175   // evaluate function
176   lsqfun(ad_handle, iflag, m, n, dco::value(xc), dco
        ::value(fvec), dco::value(fjac), ldfjac, iw, liw
        , dco::value(w), lw);
177
178   //register outputs
179   for (int i = 0; i<m; i++) {
180     dco::a1w::global_ir->register_variable(fvec[i]);
181     D->write_data(fvec[i]);
182   }
183
184   for (int i = 0; i<ldfjac*n; i++) {
185     dco::a1w::global_ir->register_variable(fjac[i]);
186     D->write_data(fjac[i]);
187   }
188
189   //insert adjoint (tape) callback
190   dco::a1w::global_ir->insert_callback(&
        lsqfun_a1w_tape_callback, D);
191 } /* lsqfun_a1w */
192
193
194 void lsqmon(void* ad_handle,
195       const int&,
196       const int&,
197       const dco::a1w::type[],
```

```
198       const dco::a1w::type[],
199       const dco::a1w::type[],
200       const int&,
201       const dco::a1w::type[],
202       const int&,
203       const int&,
204       const int&,
205       int[],
206       const int&,
207       dco::a1w::type[],
208       const int&) {}
209
210
211 int main() {
212
213   dco::a1w::type fsumsq, eta, xtol, stepmx;
214   std::vector<dco::a1w::type> fvec, fjac, s, v, w, x;
215   Integer m, n, niter, nf, iw, iprint=-1;
216   Integer maxcal, ldv, ldfjac, lw, liw;
217
218
219
220   m = MMAX;
221   n = TMAX;
222   maxcal = 10000;
223   eta = 0.01;
224   xtol = 10e-6;
225   stepmx = 100000.0;
226   ldv = n;
227   ldfjac = m;
228   liw = 1;
229   lw = 7*n + m*n + 2*m + n*n;
230
231   fjac.resize(m*ldfjac);
232   fvec.resize(m);
233   x.resize(n);
234   v.resize(ldv*n);
235   s.resize(n);
236   w.resize(lw);
237
238
239   scanf(" %*[^\n]"); /* Skip heading in data file */
240   for (int i = 0; i < m; ++i) {
241     scanf("%lf", &dco::value(st.y[i]));
242     for (int j = 0; j < n; ++j)
243       scanf("%lf", &dco::value(st.t[i][j]));
```

```
244  }
245
246
247  /* Set up the starting point */
248  x[0] = 0.5;
249  x[1] = 1.0;
250  x[2] = 1.5;
251
252  dco::a1w::global_ir = dco::a1w::ir_t::create();
253
254  for (int i = 0; i<m; i++)
255    dco::a1w::global_ir->register_variable(st.y[i]);
256
257
258  int ifail = -1;
259  void *ad_handle = dco::a1w::create_config();
260
261  E04GB_A1W_F(ad_handle, m, n,
262        E04HEV, lsqfun_a1w, lsqmon,
263        iprint, maxcal, eta, xtol, stepmx, x.data(),
264        fsumsq, fvec.data(), fjac.data(), ldfjac,
265        s.data(), v.data(), ldv, niter, nf, &iw, liw,
              w.data(), lw, ifail);
266
267  dco::derivative(fsumsq) += 1.0;
268  dco::a1w::global_ir->interpret_adjoint();
269
270  std::cout << "Solution" << std::endl;
271  for (int i = 0; i<n; i++)
272    std::cout << "x[" << i << "]= " << dco::value(x[i
       ]) << std::endl;
273
274  std::cout << "\n\nNorm of residual = " << dco::
     value(fsumsq) << std::endl;
275
276  std::cout << "\n\nResiduals"<< std::endl;
277  for (int i = 0; i<m; i++)
278    std::cout << "fvec[" << i << "]= " << dco::value(
       fvec[i]) << std::endl;
279
280  std::cout << "\n\n dFsumsq/dY NODCO ALG"<< std::
     endl;
281  for (int i = 0; i<m; i++)
282    std::cout << "dFsumsq/dY[" << i << "]= " << dco::
       derivative(st.y[i]) << std::endl;
283
```

```
284   dco::a1w::ir_t::remove(dco::a1w::global_ir);
285   dco::a1w::remove_config(ad_handle);
286
287 }
```

Listing 6: "e04gb adjoint symbolic"

```
 1 #include <vector>
 2 #include <dco.hpp>
 3 #include <nagad.hpp>
 4
 5
 6
 7 #define MMAX 15
 8 #define TMAX 3
 9
10 /* Define a user structure template to store data in
       lsqfun. */
11 template <typename T>
12 struct user
13 {
14   T y[MMAX];
15   T t[MMAX][TMAX];
16 };
17
18 static user<dco::a1w::type> st;
19
20 template <typename T>
21 void lsqfun(void* ad_handle,
22       int& iflag,
23       const int& m,
24       const int& n,
25       const T xc[],
26       T fvec[],
27       T fjac[],
28       const int& ldfjac,
29       int iw[],
30       const int& liw,
31       T w[],
32       const int& lw) {
33
34
35 #define FJAC(I, J) fjac[(J) *ldfjac + (I)]
36
37   Integer i;
38   double denom, dummy;
39
```

```
40
41   for (i = 0; i < m; ++i) {
42     denom = dco::value(xc[1]) * dco::value(st.t[i
         ][1]) + dco::value(xc[2]) * dco::value(st.t[i
         ][2]);
43     if (iflag != 1)
44       dco::value(fvec[i]) = dco::value(xc[0]) + dco::
           value(st.t[i][0]) / denom - dco::value(st.y[
           i]);
45
46     if (iflag != 0) {
47       dco::value(FJAC(i, 0)) = 1.0;
48       dummy = -1.0 / (denom * denom);
49       dco::value(FJAC(i, 1)) = dco::value(st.t[i][0])
             * dco::value(st.t[i][1]) * dummy;
50       dco::value(FJAC(i, 2)) = dco::value(st.t[i][0])
             * dco::value(st.t[i][2]) * dummy;
51     }
52   }
53 } /* lsqfun */
54
55
56 void lsqfun_a1w_tape_callback(dco::a1w::
       external_adjoint_object_t *D) {
57   //read checkpointed data
58   int const& callmode = D->read_data<int>();
59   int const& iflag = D->read_data<int>();
60   int const& m = D->read_data<int>();
61   int const& n = D->read_data<int>();
62   int const& ldfjac = D->read_data<int>();
63   int const& liw = D->read_data<int>();
64   int const& lw = D->read_data<int>();
65
66 #define FJACA(I, J) fjaca[(J) *ldfjac + (I)]
67
68   int iflag1 = iflag;
69   std::vector<dco::a1w::type> xc(n), fvec(m), fjac(
       ldfjac*n), w(lw);
70   std::vector<double> fveca(m), fjaca(ldfjac*n);
71
72   for (int i=0; i<n; i++)
73     xc[i] = D->read_data<dco::a1w::type>();
74
75   for (int i=0; i<lw; i++)
76     w[i] = D->read_data<dco::a1w::type>();
77
```

```
78  for (int i = 0; i<m; i++) {
79    fvec[i] = D->read_data<dco::a1w::type>();
80    fveca[i] = dco::derivative(fvec[i]);
81  }
82
83  for (int i = 0; i<ldfjac*n; i++) {
84    fjac[i] = D->read_data<dco::a1w::type>();
85    fjaca[i] = dco::derivative(fjac[i]);
86  }
87
88
89  std::vector<double> xca(n);
90  double denoma = 0, dummya = 0, denom, dummy;
91  user<double> sta;
92
93  for (int i = 0; i<n; i++) xca[i] = 0.0;
94  for (int i = 0; i<m; i++) sta.y[i] = 0.0;
95  for (int i = 0; i<m; i++) {
96    for (int j = 0; j<n; j++)
97      sta.t[i][j] = 0.0;
98  }
99
100 // adjoint of lsqfun
101 for (int i = m-1; i>=0; i--){
102   denom = dco::value(xc[1]) * dco::value(st.t[i
        ][1]) + dco::value(xc[2]) * dco::value(st.t[i
        ][2]);
103   denoma = 0.0;
104
105   if (iflag !=0) {
106     dummy = -1/(denom*denom);
107     dummya = 0.0;
108
109     sta.t[i][0] += FJACA(i,2)*dco::value(st.t[i
          ][2])*dummy;
110     sta.t[i][2] += FJACA(i,2)*dco::value(st.t[i
          ][0])*dummy;
111     dummya      += FJACA(i,2)*dco::value(st.t[i
          ][0])*dco::value(st.t[i][2]);
112
113     sta.t[i][0] += FJACA(i,1)*dco::value(st.t[i
          ][1])*dummy;
114     sta.t[i][1] += FJACA(i,1)*dco::value(st.t[i
          ][0])*dummy;
115     dummya      += FJACA(i,1)*dco::value(st.t[i
          ][0])*dco::value(st.t[i][1]);
```

```
116
117       denoma      += dummya*2.0/(denom*denom*denom);
118     }
119     if (iflag != 1) {
120       xca[0]      += fveca[i];
121       sta.t[i][0] += fveca[i]/denom;
122       denoma      += -fveca[i]*dco::value(st.t[i][0])
              /(denom*denom);
123       sta.y[i]    += -fveca[i];
124     }
125
126     xca[1]      += denoma*dco::value(st.t[i][1]);
127     sta.t[i][1] += denoma*xca[1];
128     xca[2]      += denoma*dco::value(st.t[i][2]);
129     sta.t[i][2] += denoma*xca[2];
130   }
131
132   if (callmode == nagad_dx || callmode == nagad_full)
        {
133    for (int i = 0; i<n; i++)
134      dco::derivative(xc[i]) += xca[i];
135   }
136   if (callmode == nagad_druser || callmode ==
        nagad_full) {
137     for (int i = 0; i<m; i++)
138       dco::derivative(st.y[i]) += sta.y[i];
139
140     for (int i = 0; i<m; i++) {
141       for (int j = 0; j<n; j++)
142     dco::derivative(st.t[i][j]) += sta.t[i][j];
143     }
144   }
145 } /* lsqfun_a1w_tape_callback */
146
147
148
149 template <typename T>
150 void lsqfun_a1w(void* ad_handle,
151     int& iflag,
152     const int& m,
153     const int& n,
154     const T xc[],
155     T fvec[],
156     T fjac[],
157     const int& ldfjac,
158     int iw[],
```

```
159     const int& liw,
160     T w[],
161     const int& lw) {
162
163   dco::a1w::external_adjoint_object_t *D = dco::a1w::
        global_ir->create_callback_object<dco::a1w::
        external_adjoint_object_t>();
164   int callmode = dco::a1w::callback_mode(ad_handle);
165
166   if(callmode != nagad_evalonly) {
167     //store inputs
168     D->write_data(callmode);
169     D->write_data(iflag);
170     D->write_data(m);
171     D->write_data(n);
172     D->write_data(ldfjac);
173     D->write_data(liw);
174     D->write_data(lw);
175
176     for (int i=0; i<n; i++)
177       D->write_data(xc[i]);
178
179     for (int i=0; i<lw; i++)
180       D->write_data(w[i]);
181   }
182
183   // evaluate function without recording the ir
184   lsqfun(ad_handle, iflag, m, n, xc, fvec, fjac,
        ldfjac, iw, liw, w, lw);
185
186   if(callmode != nagad_evalonly) {
187     //register outputs
188     for (int i = 0; i<m; i++) {
189       dco::a1w::global_ir->register_variable(fvec[i])
            ;
190       D->write_data(fvec[i]);
191     }
192
193     for (int i = 0; i<ldfjac*n; i++) {
194       dco::a1w::global_ir->register_variable(fjac[i])
            ;
195       D->write_data(fjac[i]);
196     }
197
198     //insert adjoint (tape) callback
199     dco::a1w::global_ir->insert_callback(&
```

```
          lsqfun_a1w_tape_callback, D);
200   }
201 } /* lsqfun_a1w */
202
203
204 void lsqmon(void* ad_handle,
205       const int&,
206       const int&,
207       const dco::a1w::type[],
208       const dco::a1w::type[],
209       const dco::a1w::type[],
210       const int&,
211       const dco::a1w::type[],
212       const int&,
213       const int&,
214       const int&,
215       int[],
216       const int&,
217       dco::a1w::type[],
218       const int&) {}
219
220
221 int main() {
222
223   dco::a1w::type fsumsq, eta, xtol, stepmx;
224   std::vector<dco::a1w::type> fvec, fjac, s, v, w, x;
225   Integer m, n, niter, nf, iw, iprint=-1;
226   Integer maxcal, ldv, ldfjac, lw, liw;
227
228
229
230   m = MMAX;
231   n = TMAX;
232   maxcal = 10000;
233   eta = 0.01;
234   xtol = 10e-6;
235   stepmx = 100000.0;
236   ldv = n;
237   ldfjac = m;
238   liw = 1;
239   lw = 7*n + m*n + 2*m + n*n;
240
241   fjac.resize(m*ldfjac);
242   fvec.resize(m);
243   x.resize(n);
244   v.resize(ldv*n);
```

```
245   s.resize(n);
246   w.resize(lw);
247
248
249   scanf(" %*[^\n]"); /* Skip heading in data file */
250   for (int i = 0; i < m; ++i) {
251     scanf("%lf", &dco::value(st.y[i]));
252     for (int j = 0; j < n; ++j)
253       scanf("%lf", &dco::value(st.t[i][j]));
254   }
255
256
257   /* Set up the starting point */
258   x[0] = 0.5;
259   x[1] = 1.0;
260   x[2] = 1.5;
261
262   dco::a1w::global_ir = dco::a1w::ir_t::create();
263
264   for (int i = 0; i<m; i++)
265     dco::a1w::global_ir->register_variable(st.y[i]);
266
267
268   int ifail = -1;
269   void *ad_handle = dco::a1w::create_config();
270   dco::a1w::adjoint_mode(ad_handle) = nagad_symbolic;
271
272   E04GB_A1W_F(ad_handle, m, n,
273         E04HEV, lsqfun_a1w, lsqmon,
274         iprint, maxcal, eta, xtol, stepmx, x.data(),
275         fsumsq, fvec.data(), fjac.data(), ldfjac,
276         s.data(), v.data(), ldv, niter, nf, &iw, liw,
              w.data(), lw, ifail);
277
278   dco::derivative(fsumsq) += 1.0;
279   dco::a1w::global_ir->interpret_adjoint();
280
281   std::cout << "Solution" << std::endl;
282   for (int i = 0; i<n; i++)
283     std::cout << "x[" << i << "]= " << dco::value(x[i
        ]) << std::endl;
284
285   std::cout << "\n\nNorm of residual = " << dco::
      value(fsumsq) << std::endl;
286
287   std::cout << "\n\nResiduals"<< std::endl;
```

```
288   for (int i = 0; i<m; i++)
289     std::cout << "fvec[" << i << "]= " << dco::value(
          fvec[i]) << std::endl;
290
291   std::cout << "\n\n dFsumsq/dY NODCO SYM"<< std::
        endl;
292   for (int i = 0; i<m; i++)
293     std::cout << "dFsumsq/dY[" << i << "]= " << dco::
          derivative(st.y[i]) << std::endl;
294
295   dco::a1w::ir_t::remove(dco::a1w::global_ir);
296   dco::a1w::remove_config(ad_handle);
297
298 }
```

Listing 7: "e04gb adjoint algorithmic with dco/c++"

```
1 #include <vector>
2 #include <dco.hpp>
3 #include <nagad.hpp>
4
5
6
7 #define MMAX 15
8 #define TMAX 3
9
10 /* Define a user structure template to store data in
       lsqfun. */
11 template <typename T>
12 struct user
13 {
14   T y[MMAX];
15   T t[MMAX][TMAX];
16 };
17
18 static user<dco::a1w::type> st;
19
20 template <typename T>
21 void lsqfun(void* ad_handle,
22       int& iflag,
23       const int& m,
24       const int& n,
25       const T xc[],
26       T fvec[],
27       T fjac[],
28       const int& ldfjac,
29       int iw[],
```

```
30        const int& liw,
31        T w[],
32        const int& lw) {
33
34
35  #define FJAC(I, J) fjac[(J) *ldfjac + (I)]
36
37    Integer i;
38    T denom, dummy;
39
40
41    for (i = 0; i < m; ++i) {
42      denom = xc[1] * st.t[i][1] + xc[2] * st.t[i][2];
43      if (iflag != 1)
44        fvec[i] = xc[0] + st.t[i][0] / denom - st.y[i];
45
46      if (iflag != 0) {
47        FJAC(i, 0) = 1.0;
48        dummy = -1.0 / (denom * denom);
49        FJAC(i, 1) = st.t[i][0] * st.t[i][1] * dummy;
50        FJAC(i, 2) = st.t[i][0] * st.t[i][2] * dummy;
51      }
52    }
53  } /* lsqfun */
54
55
56  void lsqmon(void* ad_handle,
57        const int&,
58        const int&,
59        const dco::a1w::type[],
60        const dco::a1w::type[],
61        const dco::a1w::type[],
62        const int&,
63        const dco::a1w::type[],
64        const int&,
65        const int&,
66        const int&,
67        int[],
68        const int&,
69        dco::a1w::type[],
70        const int&) {}
71
72
73  int main() {
74
75    dco::a1w::type fsumsq, eta, xtol, stepmx;
```

```
76  std::vector<dco::a1w::type> fvec, fjac, s, v, w, x;
77  Integer m, n, niter, nf, iw, iprint=-1;
78  Integer maxcal, ldv, ldfjac, lw, liw;
79
80
81
82  m = MMAX;
83  n = TMAX;
84  maxcal = 10000;
85  eta = 0.01;
86  xtol = 10e-6;
87  stepmx = 100000.0;
88  ldv = n;
89  ldfjac = m;
90  liw = 1;
91  lw = 7*n + m*n + 2*m + n*n;
92
93  fjac.resize(m*ldfjac);
94  fvec.resize(m);
95  x.resize(n);
96  v.resize(ldv*n);
97  s.resize(n);
98  w.resize(lw);
99
100
101 scanf(" %*[^\n]"); /* Skip heading in data file */
102 for (int i = 0; i < m; ++i) {
103   scanf("%lf", &dco::value(st.y[i]));
104   for (int j = 0; j < n; ++j)
105     scanf("%lf", &dco::value(st.t[i][j]));
106 }
107
108
109 /* Set up the starting point */
110 x[0] = 0.5;
111 x[1] = 1.0;
112 x[2] = 1.5;
113
114 dco::a1w::global_ir = dco::a1w::ir_t::create();
115
116 for (int i = 0; i<m; i++)
117   dco::a1w::global_ir->register_variable(st.y[i]);
118
119
120 int ifail = -1;
121 void *ad_handle = dco::a1w::create_config();
```

```
122
123   E04GB_A1W_F(ad_handle, m, n,
124         E04HEV, lsqfun, lsqmon,
125         iprint, maxcal, eta, xtol, stepmx, x.data(),
126         fsumsq, fvec.data(), fjac.data(), ldfjac,
127         s.data(), v.data(), ldv, niter, nf, &iw, liw,
                w.data(), lw, ifail);
128
129   dco::derivative(fsumsq) += 1.0;
130   dco::a1w::global_ir->interpret_adjoint();
131
132   std::cout << "Solution" << std::endl;
133   for (int i = 0; i<n; i++)
134     std::cout << "x[" << i << "]= " << dco::value(x[i
          ]) << std::endl;
135
136   std::cout << "\n\nNorm of residual = " << dco::
         value(fsumsq) << std::endl;
137
138   std::cout << "\n\nResiduals"<< std::endl;
139   for (int i = 0; i<m; i++)
140     std::cout << "fvec[" << i << "]= " << dco::value(
          fvec[i]) << std::endl;
141
142   std::cout << "\n\n dFsumsq/dY DCO ALG"<< std::endl;
143   for (int i = 0; i<m; i++)
144     std::cout << "dFsumsq/dY[" << i << "]= " << dco::
          derivative(st.y[i]) << std::endl;
145
146   dco::a1w::ir_t::remove(dco::a1w::global_ir);
147   dco::a1w::remove_config(ad_handle);
148
149 }
```

Listing 8: "e04gb adjoint symbolic with dco/c++"

```
1 #include <vector>
2 #include <dco.hpp>
3 #include <nagad.hpp>
4
5
6
7 #define MMAX 15
8 #define TMAX 3
9
10 /* Define a user structure template to store data in
     lsqfun. */
```

```
11 template <typename T>
12 struct user
13 {
14   T y[MMAX];
15   T t[MMAX][TMAX];
16 };
17
18 static user<dco::a1w::type> st;
19
20 template <typename T>
21 void lsqfun(void* ad_handle,
22       int& iflag,
23       const int& m,
24       const int& n,
25       const T xc[],
26       T fvec[],
27       T fjac[],
28       const int& ldfjac,
29       int iw[],
30       const int& liw,
31       T w[],
32       const int& lw) {
33
34
35 #define FJAC(I, J) fjac[(J) *ldfjac + (I)]
36
37   Integer i;
38   T denom, dummy;
39
40
41   for (i = 0; i < m; ++i) {
42     denom = xc[1] * st.t[i][1] + xc[2] * st.t[i][2];
43     if (iflag != 1)
44       fvec[i] = xc[0] + st.t[i][0] / denom - st.y[i];
45
46     if (iflag != 0) {
47       FJAC(i, 0) = 1.0;
48       dummy = -1.0 / (denom * denom);
49       FJAC(i, 1) = st.t[i][0] * st.t[i][1] * dummy;
50       FJAC(i, 2) = st.t[i][0] * st.t[i][2] * dummy;
51     }
52   }
53 } /* lsqfun */
54
55 template <typename T>
56 void lsqfun_dx(void* ad_handle,
```

```
57        int& iflag,
58        const int& m,
59        const int& n,
60        const T xc[],
61        T fvec[],
62        T fjac[],
63        const int& ldfjac,
64        int iw[],
65        const int& liw,
66        T w[],
67        const int& lw) {
68
69
70  #define FJAC(I, J) fjac[(J) *ldfjac + (I)]
71
72    Integer i;
73    T denom, dummy;
74
75
76      for (int i = 0; i < m; ++i) {
77        denom = xc[1] *dco::value(st.t[i][1]) + xc[2] *
              dco::value(st.t[i][2]);
78        if (iflag != 1)
79        fvec[i] = xc[0] + dco::value(st.t[i][0]) /
             denom - dco::value(st.y[i]);
80
81        if (iflag != 0) {
82        FJAC(i, 0) = 1.0;
83        dummy = -1.0 / (denom * denom);
84        FJAC(i, 1) = dco::value(st.t[i][0]) * dco::
             value(st.t[i][1]) * dummy;
85        FJAC(i, 2) = dco::value(st.t[i][0]) * dco::
             value(st.t[i][2]) * dummy;
86        }
87      }
88  } /* lsqfun_dx */
89
90  template <typename T>
91  void lsqfun_druser(void* ad_handle,
92        int& iflag,
93        const int& m,
94        const int& n,
95        const T xc[],
96        T fvec[],
97        T fjac[],
98        const int& ldfjac,
```

```
99        int iw[],
100       const int& liw,
101       T w[],
102       const int& lw) {
103
104
105 #define FJAC(I, J) fjac[(J) *ldfjac + (I)]
106
107   Integer i;
108   T denom, dummy;
109
110   for (int i = 0; i < m; ++i) {
111     denom = dco::value(xc[1]) * st.t[i][1] + dco::
          value(xc[2]) * st.t[i][2];
112     if (iflag != 1)
113       fvec[i] =dco::value(xc[0]) + st.t[i][0] / denom
              - st.y[i];
114
115     if (iflag != 0) {
116       FJAC(i, 0) = 1.0;
117       dummy = -1.0 / (denom * denom);
118       FJAC(i, 1) = st.t[i][0] * st.t[i][1] * dummy;
119       FJAC(i, 2) = st.t[i][0] * st.t[i][2] * dummy;
120     }
121   }
122 } /* lsqfun_druser */
123
124
125
126 template <typename T>
127 void lsqfun_a1w(void* ad_handle,
128       int& iflag,
129       const int& m,
130       const int& n,
131       const T xc[],
132       T fvec[],
133       T fjac[],
134       const int& ldfjac,
135       int iw[],
136       const int& liw,
137       T w[],
138       const int& lw) {
139
140   int callmode = dco::a1w::callback_mode(ad_handle);
141
142   if (callmode == nagad_evalonly)
```

```
143     lsqfun(ad_handle, iflag, m, n, xc, fvec, fjac,
            ldfjac, iw, liw, w, lw);
144   else if (callmode==nagad_dx)
145     lsqfun_dx(ad_handle, iflag, m, n, xc, fvec, fjac,
             ldfjac, iw, liw, w, lw);
146   else if (callmode==nagad_druser)
147     lsqfun_druser(ad_handle, iflag, m, n, xc, fvec,
            fjac, ldfjac, iw, liw, w, lw);
148   else if (callmode==nagad_full)
149     lsqfun(ad_handle, iflag, m, n, xc, fvec, fjac,
            ldfjac, iw, liw, w, lw);
150 } /* lsqfun_a1w */
151
152 void lsqmon(void* ad_handle,
153       const int&,
154       const int&,
155       const dco::a1w::type[],
156       const dco::a1w::type[],
157       const dco::a1w::type[],
158       const int&,
159       const dco::a1w::type[],
160       const int&,
161       const int&,
162       const int&,
163       int[],
164       const int&,
165       dco::a1w::type[],
166       const int&) {}
167
168
169 int main() {
170
171   dco::a1w::type fsumsq, eta, xtol, stepmx;
172   std::vector<dco::a1w::type> fvec, fjac, s, v, w, x;
173   Integer m, n, niter, nf, iw, iprint=-1;
174   Integer maxcal, ldv, ldfjac, lw, liw;
175
176
177
178   m = MMAX;
179   n = TMAX;
180   maxcal = 10000;
181   eta = 0.01;
182   xtol = 10e-6;
183   stepmx = 100000.0;
184   ldv = n;
```

```
185  ldfjac = m;
186  liw = 1;
187  lw = 7*n + m*n + 2*m + n*n;
188
189  fjac.resize(m*ldfjac);
190  fvec.resize(m);
191  x.resize(n);
192  v.resize(ldv*n);
193  s.resize(n);
194  w.resize(lw);
195
196
197  scanf(" %*[^\n]"); /* Skip heading in data file */
198  for (int i = 0; i < m; ++i) {
199    scanf("%lf", &dco::value(st.y[i]));
200    for (int j = 0; j < n; ++j)
201      scanf("%lf", &dco::value(st.t[i][j]));
202  }
203
204
205  /* Set up the starting point */
206  x[0] = 0.5;
207  x[1] = 1.0;
208  x[2] = 1.5;
209
210  dco::a1w::global_ir = dco::a1w::ir_t::create();
211
212  for (int i = 0; i<m; i++)
213    dco::a1w::global_ir->register_variable(st.y[i]);
214
215
216  int ifail = -1;
217  void *ad_handle = dco::a1w::create_config();
218  dco::a1w::adjoint_mode(ad_handle) = nagad_symbolic;
219
220  E04GB_A1W_F(ad_handle, m, n,
221        E04HEV, lsqfun_a1w, lsqmon,
222        iprint, maxcal, eta, xtol, stepmx, x.data(),
223        fsumsq, fvec.data(), fjac.data(), ldfjac,
224        s.data(), v.data(), ldv, niter, nf, &iw, liw,
              w.data(), lw, ifail);
225
226  dco::derivative(fsumsq) += 1.0;
227  dco::a1w::global_ir->interpret_adjoint();
228
229  std::cout << "Solution" << std::endl;
```

```
230   for (int i = 0; i<n; i++)
231     std::cout << "x[" << i << "]= " << dco::value(x[i
          ]) << std::endl;
232
233   std::cout << "\n\nNorm of residual = " << dco::
        value(fsumsq) << std::endl;
234
235   std::cout << "\n\nResiduals"<< std::endl;
236   for (int i = 0; i<m; i++)
237     std::cout << "fvec[" << i << "]= " << dco::value(
          fvec[i]) << std::endl;
238
239   std::cout << "\n\n dFsumsq/dY DCO SYM"<< std::endl;
240   for (int i = 0; i<m; i++)
241     std::cout << "dFsumsq/dY[" << i << "]= " << dco::
          derivative(st.y[i]) << std::endl;
242
243   dco::a1w::ir_t::remove(dco::a1w::global_ir);
244   dco::a1w::remove_config(ad_handle);
245
246 }
```

References

[1] http://nash.nag.co.uk/nagdoc/nagdoc_fl26.2/adhtml/e04/e04gb_ad_f.html

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

[3] Dunford N, Schwartz J T, Bade W G and Bartle R G (1971) Linear Operators Wiley Interscience, New York

[4] Naumann U (2012) The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation SIAM

[5] Naumann U, Lotz J, Leppkes K, Towara M (2015) Algorithmic Differentiation of Numerical Methods: Tangent and Adjoint Solvers for Parameterized Systems of Nonlinear Equations ACM Trans. Math. Softw.

[6] Griewank A and Walther A (2008) Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiations (2nd Edition) SIAM

[7] Griewank A and Walther A (2008) Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiations (2nd Edition) SIAM