User guide for GADfit version 1.3

October 2019

Contents

1 Introduction

GADfit is a Fortran implementation of **g**lobal nonlinear curve fitting based on **a**utomatic **d**ifferentiation (AD). Global fitting refers to fitting many data sets simultaneously with some parameters shared among the data sets (not necessarily

finding the global minimum in parameter space). The optimization procedure is based on a modified Levenberg-Marquardt algorithm, which is well-suited for difficult large scale problems with a strong interdependence of parameters. The Jacobian and other quantities requiring differentiation are calculated using AD instead of finite differences, which ensures that the derivatives are always calculated with the same precision as function evaluation. The cost of calculating the derivatives is *independent* of the number of fitting parameters and is a small constant of function evaluation, resulting in very efficient code. The method can be used for fitting functions of high complexity which, in the present implementation, includes nonlinear combinations of elementary and special functions, single or double integrals, and any control flow statements allowed by the programming language.

GADfit is currently hosted by GitHub.

1.1 Contributors

Raul Laasner raullaasner@gmail.com

1.2 Terms of use

2 Method

GADfit is based on the Levenberg-Marquardt algorithm [1], which is a standard technique for solving nonlinear least squares problems. A least squares problem refers to minimizing the sum of the squares of the differences between the data and a curve constructed by a parameterized function. If the model curve (the fitting function) has a nonlinear dependence on the fitting parameters, it is referred to as a nonlinear least squares problem.

Given a set of n data points (x_i, y_i), where x_i are the independent variables, and a model function $f(\boldsymbol{x}, \beta_1, \ldots, \beta_k)$, which depends on the x_i and k additional parameters (the fitting parameters), the sum of squares is defined as

$$\chi^2 = \sum_{i=1}^{n} \left[\frac{y_i - f(x_i, \boldsymbol{\beta})}{\sigma_i} \right]^2, \tag{1}$$

where σ_i are the experimental uncertainties. As with all nonlinear optimization algorithms, Levenberg-Marquardt is an iterative procedure. One starts with an initial guess for the parameter vector $\boldsymbol{\beta}$ and finds an increment $\boldsymbol{\delta}$ such that $\boldsymbol{\beta} \to \boldsymbol{\beta} + \boldsymbol{\delta}$ leads to the lowering of χ^2. This is repeated until some convergence criterion is satisfied. Following the original LM method, the increment vector (the step) is explicitly given by

$$\left[\boldsymbol{J}^T \boldsymbol{J} + \lambda \operatorname{diag}\left(\boldsymbol{J}^T \boldsymbol{J}\right)\right] \boldsymbol{\delta} = \boldsymbol{J}^T \left[\boldsymbol{y} - \boldsymbol{f}(\boldsymbol{\beta})\right], \tag{2}$$

where[1]

$$J_{ij} = \frac{1}{\sigma_i} \frac{\partial f(x_i, \boldsymbol{\beta})}{\partial \beta_j} \tag{3}$$

defines the Jacobian matrix and where the damping parameter λ allows the nature of the algorithm to adaptively change between the steepest descent and the Gauss-Newton method. If χ^2 is far from its minimum, λ should have a large value, bringing the algorithm closer to steepest descent. On the other hand, close to the minimum, Gauss-Newton is the better choice. A simple strategy for updating λ is as follows: if for the current iteration χ^2 is smaller than for the previous iteration, accept the step and decrease λ by a factor of 10. Otherwise increase λ, solve Eq. (2), and recalculate χ^2 with the new step. If χ^2 still increases, stop the procedure, otherwise proceed with the next iteration. If the procedure stops after the first iteration, start again with a larger initial value for λ. A sufficiently large λ always exists that leads to a decrease of χ^2 [1].

2.1 Modified Levenberg-Marquardt

What was described above is the classical LM algorithm, which is close to the default regime of GADfit unless the user modifies some of the default input parameters. In this section, we discuss several interesting modifications to the algorithm (see [2, 3, 4] and references therein), which aim to improve the convergence speed and the sensitivity to the initial guess.

The damping matrix. Originally, the rationale for choosing the damping matrix as $\boldsymbol{D}^T \boldsymbol{D} \equiv \operatorname{diag}(\boldsymbol{J}^T \boldsymbol{J})$ in Eq. (2) was to make the algorithm invariant to the rescaling of parameters. This is important when the parameters have very different magnitudes and the algorithm has to follow a narrow canyon in the parameter space in search of the minimum of χ^2. However, this also poses a problem when the algorithm is in a region of space where the model is insensitive to the parameters. When this happens, the diagonal entries of both $\boldsymbol{J}^T \boldsymbol{J}$ and the damping matrix $\boldsymbol{D}^T \boldsymbol{D}$ become small for some of the parameters, which leads to uncontrollably large steps. There is a chance that some parameters are incorrectly pushed to infinite values, a phenomenon known as parameter evaporation. A compromise for both retaining the invariance to rescaling and guarding against parameter evaporation is to set the diagonal entries of $\boldsymbol{D}^T \boldsymbol{D}$ to

[1] Usually the error factors σ_i are not included in the definition.

the largest diagonal entries of $\boldsymbol{J}^T\boldsymbol{J}$ yet encountered. This approach is efficient provided that the initial guess does not lie on a plateau of the χ^2 surface in the parameter space. If this condition is not satisfied then, as a simple and robust solution, the user can specify the minimum values for the diagonal of $\boldsymbol{D}^T\boldsymbol{D}$.

Optimal path towards the minimum. Following a geometric interpretation of nonlinear optimization, it has been suggested that a geodesic in the parameter space is the most natural path for the algorithm to follow. When the algorithm is navigating a narrow curved canyon then, instead of taking simple steps along the gradient of χ^2, it would be more efficient to take the curvature of the canyon into account and move along parabolic trajectories. This is especially beneficial for functions like the Rosenbrock function. To second order, the step is then given by

$$\begin{aligned} \boldsymbol{\delta} =& \boldsymbol{\delta}_1 + \boldsymbol{\delta}_2, \\ \boldsymbol{\delta}_1 =& \left[\boldsymbol{J}^T\boldsymbol{J} + \lambda\boldsymbol{D}^T\boldsymbol{D}\right]^{-1}\boldsymbol{J}^T\left[\boldsymbol{y} - \boldsymbol{f}(\boldsymbol{\beta})\right], \\ \boldsymbol{\delta}_2 =& -\frac{1}{2}\left[\boldsymbol{J}^T\boldsymbol{J} + \lambda\boldsymbol{D}^T\boldsymbol{D}\right]^{-1}\boldsymbol{J}^T\boldsymbol{\Omega}, \end{aligned} \tag{4}$$

where (Einstein summation implied)

$$\Omega_i = \frac{\partial^2 f(x_i,\beta)}{\partial\beta_j\partial\beta_k}\delta_1^j\delta_1^k \tag{5}$$

is the second directional derivative along $\boldsymbol{\delta}_1$. From geometric considerations, $\boldsymbol{\delta}_1$ and $\boldsymbol{\delta}_2$ are also called the velocity and the accelerations terms. The acceleration term either magnifies or shrinks the velocity term, or tilts it in a different direction. We do not delve here into the theoretical foundations leading to Eq. (4) and its comparison to alternative ways of including second order corrections. The interested reader is referred to [2, 3, 4] for an in-depth discussion. We only make a few additional remarks for practical calculations.

The geodesic acceleration is most useful in a narrow canyon. When the algorithm has not yet found a canyon and is scanning a large plateau, which is where the initial guess is likely to be, then damping is small and $\boldsymbol{\delta}_1$ is large. This causes $\boldsymbol{\delta}_2$ to be even larger and pointing in the opposite direction, which may steer the algorithm in the wrong direction. To avoid this, we require the effect of the second order term to always be smaller than the first order term,

$$\frac{\sqrt{\boldsymbol{\delta}_2\boldsymbol{D}\boldsymbol{\delta}_2}}{\sqrt{\boldsymbol{\delta}_1\boldsymbol{D}\boldsymbol{\delta}_1}} < \alpha, \quad 0 \le \alpha < 1, \tag{6}$$

where $\boldsymbol{D}$ has been included to ensure scale invariance.

Whether including the acceleration term pays off depends on the model and is ultimately for the user to test. With automatic differentiation the second directional derivative has about the same cost as the Jacobian, which is 3–4 times function evaluation.

Uphill steps. Accepting only downhill steps is the safest strategy, especially when one is far from the minimum. However, if the algorithm has to follow a narrow canyon, only very short steps are acceptable, making the search costly. It might then be beneficial to conditionally also allow uphill steps. This can be thought of as analogous to the trajectory of a bobsled racer. A useful criterion for allowing an uphill step is

$$\begin{aligned} &\beta_i = \cos(\boldsymbol{\delta}_{1i}, \boldsymbol{\delta}_{1i-1}), \\ &(1-\beta_i)^b \chi_i^2 < \min(\chi_1^2, \ldots, \chi_{i-1}^2), \end{aligned} \tag{7}$$

where β_i is the cosine of the angle between the proposed step of the current iteration and that of the previous iteration. Only the velocity and not the acceleration term is considered here. According to Eq. (7), uphill steps are allowed only for acute angles, and the probability of acceptance increases with the alignment of $\boldsymbol{\delta}_{1i}$ and $\boldsymbol{\delta}_{1i-1}$. The RHS of the inequality test usually refers to the sum of squares of the previous iteration, but we use the min function here in case any of the previous steps were uphill. Reasonable values for the exponent are 1 and 2, with $b = 2$ allowing greater uphill steps than $b = 1$.

In addition to faster convergence, this so-called "bold" acceptance criterion also exhibits a smaller probability of getting stuck at the local minima, which reside on the floor of the canyon.

The drawback of this criterion is the reduced stability of the algorithm, since allowing uphill steps makes it more susceptible to parameter evaporation. It is better suited for problems where the canyon is easily found, but navigating the canyon is the main difficulty. This is in contrast to problems that start in a difficult region of the parameter space, but once heading in the right direction, the minimum is easily found.

Updating the damping parameter. The simple scheme of raising and lowering λ by a factor of 10 as the steps are accepted or rejected is often adequate, but can be improved. Increasing λ by a small factor $\lambda_\uparrow$ when the step is uphill and decreasing it by a larger factor $\lambda_\downarrow$ when the step is downhill is expected to lead to a faster convergence (see Sec. VIII.C of [3]). The optimal values for the raising and lowering factors depend on the problem. One might start with, e.g., $\lambda_\uparrow = 3$ and $\lambda_\downarrow = 5$.

A more advanced method for updating the damping parameter makes use of the gain factor,

$$\rho = \frac{\chi^2(\boldsymbol{x}, \boldsymbol{\beta}) - \chi^2(\boldsymbol{x}, \boldsymbol{\beta} + \boldsymbol{\delta}_1)}{\chi^2(\boldsymbol{x}, \boldsymbol{\beta}) - |[\boldsymbol{y} - \boldsymbol{f}(\boldsymbol{\beta})]/\boldsymbol{\sigma} - 2(\boldsymbol{J} + \lambda \boldsymbol{D})\boldsymbol{\delta}_1|^2} = \frac{\chi^2(\boldsymbol{x}, \boldsymbol{\beta}) - \chi^2(\boldsymbol{x}, \boldsymbol{\beta} + \boldsymbol{\delta}_1)}{2\boldsymbol{\delta}_1(\boldsymbol{J}^T\boldsymbol{J} + \lambda \boldsymbol{D}^T\boldsymbol{D})\boldsymbol{\delta}_1}, \tag{8}$$

where $\boldsymbol{a}/\boldsymbol{b}$ implies element-wise division. The second term in the denominator is the linearly predicted value of χ^2 using the new parameter vector $\boldsymbol{\beta} + \boldsymbol{\delta}_1$. A downhill step always produces a positive ρ and thus the steps can be accepted or rejected based on the sign of ρ. If ρ is close to zero, λ is updated with $\lambda_\uparrow$ and $\lambda_\downarrow$. However, if ρ is much greater than zero then the step should still be

accepted but now the algorithm is stepping out of its trust-region. The next step is likely to increase χ^2, which requires the lowering of λ. The oscillating behavior of λ can be avoided by limiting the decrease of λ so that the algorithm is unable to step out of the trust-region. Following Nielsen [5], this can be achieved by multiplying λ by $\max[1/\lambda_\downarrow, 1-(2\rho-1)^3]$ for $\rho > 0$. This method tends to increase the total number of iterations, but since the steps are accepted more often, χ^2 is calculated by a lesser number of times per iteration, which can overall speed up the procedure.

Another method for updating the damping parameter is due to Umrigar and Nightingale (unpublished). The idea is to vary λ based on the history of the success of previous steps. If the current step is accepted then λ is either decreased by a factor of ξ if the change in the direction of $\boldsymbol{\delta}_1$ is less that $\pi/2$, left unchanged if the change is more than $\pi/2$, or increased if the step was uphill:

$$\lambda_i = \lambda_{i-1}/\xi, \qquad \cos(\boldsymbol{\delta}_{1i}, \boldsymbol{\delta}_{1i-1}) \geq 0, \tag{9}$$

$$\lambda_i = \lambda_{i-1}, \qquad \cos(\boldsymbol{\delta}_{1i}, \boldsymbol{\delta}_{1i-1}) < 0, \tag{10}$$

$$\lambda_i = \lambda_{i-1}\sqrt{\xi}, \qquad \chi^2_i \geq \chi^2_{i-1}. \tag{11}$$

If the step is rejected, then λ is always increased, but the increase depends on the change of $\boldsymbol{\delta}_1$:

$$\lambda_i = \lambda_{i-1}\sqrt{\xi}, \qquad \cos(\boldsymbol{\delta}_{1i}, \boldsymbol{\delta}_{1i-1}) \geq 0, \tag{12}$$

$$\lambda_i = \lambda_{i-1}\xi, \qquad \cos(\boldsymbol{\delta}_{1i}, \boldsymbol{\delta}_{1i-1}) < 0. \tag{13}$$

ξ is constrained to the range $1 \ldots 100$ and is given by

$$\xi = (1 - |2a - 1|))^{-2}, \tag{14}$$

where a is updated according to

$$a_i = a_{i-1}m + A(1-m), \tag{15}$$

where

$$A = 1 \qquad \cos(\boldsymbol{\delta}_{1i}, \boldsymbol{\delta}_{1i-1}) \geq 0 \quad \text{and} \quad \chi^2_i < \chi^2_{i-1}, \tag{16}$$

$$A = 1/2, \qquad \cos(\boldsymbol{\delta}_{1i}, \boldsymbol{\delta}_{1i-1}) < 0 \quad \text{or} \quad \chi^2_i \geq \chi^2_{i-1}, \tag{17}$$

$$A = 0 \qquad \text{if step was rejected.} \tag{18}$$

The method works well in some cases but has thus far not seen extensive testing. In GADfit, the inital values for a and m are set to 0.5 and $e^{-0.2}$.

Convergence criteria. At the solution, the residual vector $\boldsymbol{y}-\boldsymbol{f}$ is orthogonal to the range of the Jacobian, $\boldsymbol{J}\boldsymbol{\delta}_1$. A measure of convergence can then be taken as the cosine of the angle between these two quantities,

$$|\cos\phi| = \frac{|(\boldsymbol{y}-\boldsymbol{f})\boldsymbol{J}\boldsymbol{\delta}_1|}{|\boldsymbol{y}-\boldsymbol{f}||\boldsymbol{J}\boldsymbol{\delta}_1|}. \tag{19}$$

In addition to (19), most of the standard convergence criteria are available in GADfit. See `gadf_fit` in Sec. 3.4.1.

2.2 Global nonlinear optimization

When fitting many curves simultaneously with shared parameters, the data sets are stacked to form a single data set, which is then fitted with a piecewise function. The fitting function in constructed so that some parameters only have an effect on a subset of data points (local parameters) whereas others are active over the whole data range (global parameters).

As an example, imagine two data sets with sizes n and m and a fitting function $f(x, \alpha, \beta)$, which depends on two parameters α and β. α is different for the two curves whereas β is shared. For an exponential decay process, these could be different initial amplitudes but the same decay constant. The global Jacobian is then

$$\boldsymbol{J} = \begin{pmatrix} \frac{1}{\sigma_1}\frac{\partial f(x_1,\alpha,\beta)}{\partial \alpha_1} & 0 & \frac{1}{\sigma_1}\frac{\partial f(x_1,\alpha,\beta)}{\partial \beta} \\ \cdots & \cdots & \cdots \\ \frac{1}{\sigma_n}\frac{\partial f(x_n,\alpha,\beta)}{\partial \alpha_1} & 0 & \frac{1}{\sigma_n}\frac{\partial f(x_n,\alpha,\beta)}{\partial \beta} \\ 0 & \frac{1}{\sigma_{n+1}}\frac{\partial f(x_{n+1},\alpha,\beta)}{\partial \alpha_2} & \frac{1}{\sigma_{n+1}}\frac{\partial f(x_{n+1},\alpha,\beta)}{\partial \beta} \\ \cdots & \cdots & \cdots \\ 0 & \frac{1}{\sigma_{n+m}}\frac{\partial f(x_{n+m},\alpha,\beta)}{\partial \alpha_2} & \frac{1}{\sigma_{n+m}}\frac{\partial f(x_{n+m},\alpha,\beta)}{\partial \beta} \end{pmatrix}, \tag{20}$$

where the notation is $\partial f/\partial \alpha_i \equiv \partial f/\partial \alpha|_{\alpha=\alpha_i}$. Similarly, with 4 data sets, each consisting of 100 points, and with 4 local and 2 global parameters, the Jacobian would have the dimensions 400×18. Sparsity of the Jacobian is not exploited. Since no conceptual change is introduced, the fitting procedure follows the same algorithm as with a single curve.

2.3 Automatic differentiation

If the Jacobian (20) was calculated using finite differences, the maximum accuracy one can hope for is half the number of significant digits of machine precision (about 7 for double precision). This is often sufficient if the fitting function is a simple combination of elementary functions such as the Gaussian or the exponential. However, problems can arise with difficult functions such as those containing a highly nonlinear combination of elementary or special functions or those requiring numerical integration. In such situations, both accuracy and the computational cost of finite differences can become problematic. With finite differences, the evaluation of J_{ij} always takes two function calls and the evaluation of the whole gradient $(J_{i1}, \ldots, J_{in})$ takes $2n$ function calls.

In contrast, with automatic differentiation the derivatives are obtained with the same accuracy as function evaluation itself, while the computational cost is 3–4 times that of function evaluation *irrespective* of the number of parameters [6]. AD exploits the fact that any mathematical function, no matter how complicated, is executed as a sequence of elementary arithmetic operations on a computer. By applying the chain rule of calculus at each step of execution, the derivative of the function can be calculated automatically, with working accuracy, and using at most a small constant of the computer time required for

function evaluation. It should be noted that while AD is not numerical differentiation, it is also not symbolic differentiation. The chain rule is applied directly to the results of elementary operations and the symbolic form of the derivative is never saved in any way. Only the derivatives of the elementary functions need to be coded by hand (in AD literature, the term "elemental function" is more common). Afterwards, the algorithm can be applied to functions of arbitrary complexity.

In this document, we describe two flavors of AD. The forward mode of AD is well-suited for calculating the gradient of a multi-valued function depending on a single parameter (called the derivative in this case) and for calculating the directional derivative of a single-valued function depending on many parameters. The reverse mode is suitable for calculating the gradient of a single-valued function depending on many parameters. For the general case of n input and m output variables, a nontrivial mix of both modes would in principle be optimal.

2.3.1 Forward mode

The basic idea behind the forward mode is to extend all numbers to include a second component,

$$x \to \tilde{x} = x + \dot{x}d \equiv (x, \dot{x}), \tag{21}$$

resulting in so-called dual numbers. The arithmetic on dual numbers is defined by requiring $d^2 = 0$. This is in contrast to ordinary complex numbers, which are also dual numbers but where the second component is defined according to $i^2 = -1$. Elementary arithmetic between dual numbers can then be written as follows:

$$\begin{aligned}
\tilde{x} + \tilde{y} =& x + y + (\dot{x} + \dot{y})d = (x + y, \dot{x} + \dot{y}), \\
\tilde{x}\tilde{y} =& xy + x\dot{y}d + \dot{x}yd + \dot{x}\dot{y}d^2 = (xy, \dot{x}y + x\dot{y}), \\
\frac{\tilde{x}}{\tilde{y}} =& \frac{x + \dot{x}d}{y + \dot{y}d} = \frac{(x + \dot{x}d)(y - \dot{y}d)}{y^2 - \dot{y}^2 d^2} = \left(\frac{x}{y}, \frac{\dot{x}y - x\dot{y}}{y^2}\right), \\
\sin\tilde{x} =& \ldots = (\sin x, \dot{x}\cos x),
\end{aligned} \tag{22}$$

where the last equality follows after a series expansion. Any nondual number can be interpreted as $\tilde{x} = x + 0d$. It is seen from Eqs. (22) that the second component, also called the tangent, always has the form of the derivative of the operation. In fact, it can be proven that for any function $f(x)$, with a dual number as its argument,

$$f(\tilde{x}) = f(x) + f'(x)\dot{x}d = (f(x), f'(x)\dot{x}), \tag{23}$$

where $f'(x)$ is the derivative of f evaluated at x. Applying the dual number arithmetic to the composition of f and g, we get

$$f(g(\tilde{x})) = f(g(x) + g'(x)\dot{x}d) = (f(g(x)), f'(g(x))g'(x)\dot{x}), \tag{24}$$

In Eq. (24), we have actually rediscovered the chain rule. Repeatedly applying Eq. (24) yields the derivatives of arbitrarily complex functions.

Which derivative the dot symbol represents depends on the initial values of the tangents. It can be shown that in general the initial values define the direction of the derivative in the space of independent variables. For instance, if we have two independent variables x and y and require the derivative with respect to x, then the tangent vector is $(1, 0)$ in the xy-plane and the variables should be seeded with $\dot{x} = \mathrm{d}x/\,\mathrm{d}x = 1$ and $\dot{y} = \mathrm{d}y/\,\mathrm{d}x = 0$. For a directional derivative $\nabla_{x,y} f(x, y)\boldsymbol{v}$, where $\boldsymbol{v} = (\alpha, \beta)$, the seeds would be $\dot{x} = \alpha$ and $\dot{y} = \beta$.

As a simple example, we shall evaluate $\frac{\partial}{\partial x} f(x, y)$ for $f(x, y) = xy + \sin(xy)$ at (x_0, y_0). Applying the rules discussed thus far, the computational graph for this procedure, in terms of the intermediate results $\tilde{v}_i$, is

$$\begin{aligned}
\tilde{v}_1 =& (x_0, \dot{x}) = (x_0, 1), \\
\tilde{v}_2 =& (y_0, \dot{y}) = (y_0, 0), \\
\tilde{v}_3 =& \tilde{v}_1 \tilde{v}_2 = (v_1 v_2, v_1 \dot{v}_2 + \dot{v}_1 v_2) = (x_0 y_0, y_0), \\
\tilde{v}_4 =& \sin \tilde{v}_3 = (\sin v_3, \dot{v}_3 \cos v_3) = (\sin x_0, y_0 \cos(x_0 y_0)), \\
\tilde{v}_5 =& \tilde{v}_3 + \tilde{v}_4 = (x_0 y_0 + \sin x_0 y_0, y_0 + y_0 \cos(x_0 y_0)).
\end{aligned} \tag{25}$$

For such a simple case, it is easy to analytically check that the tangent of $\tilde{v}_5$ is indeed the correct result. Note that AD is as economical as the function evaluation in terms of reusing intermediate results — the derivative of xy (the tangent of $\tilde{v}_3$) is calculated only once, even though it appears in two places. Again, we remind that AD is not symbolic differentiation since there is no record of the symbolic form of the derivatives. The algorithm only needs to recognize the elemental operation at hand, whose derivative needs to be precoded, but otherwise the procedure is completely mechanical. At each step in the computational graph, the algorithm has forgotten how it got there. For a better illustration, here is a throwaway C++ implementation of the same calculation with $(x_0, y_0) = (1.5, 0.7)$.

```
class ddouble {
public:
  ddouble(double val=0, double dot=0) : val(val), dot(dot) {}
  double val, dot;
};
ddouble operator+ (ddouble x, ddouble y) {
  return ddouble(x.val+y.val, x.dot+y.dot);
}
ddouble operator* (ddouble x, ddouble y) {
  return ddouble(x.val*y.val, x.val*y.dot+x.dot*y.val);
}
ddouble sin(ddouble x) {
  return ddouble(sin(x.val), x.dot*cos(x.val));
}
int main() {
  ddouble x(1.5, 1);
  ddouble y(0.7, 0);
```

```
  ddouble f;
  f = x*y + sin(x*y);
  std::cout << f.dot; // 1.0483
}
```

If, instead of just $\frac{\partial}{\partial x}$, the whole gradient $\nabla_{x,y} f(x,y)$ is required, a second calculation needs to be performed starting with $\dot{x} = 0$, $\dot{y} = 1$. The accuracy is the same as before, but the cost is twice as much. Note that for each partial derivative, the values (first components) of the elemental operations are the same. Due to the $O(n)$ scaling, the forward mode is generally not used for gradient calculation. Instead, it is better suited for directional derivatives, where a single sweep produces the value of $\nabla_{\boldsymbol{p}} f(\boldsymbol{p})\boldsymbol{v}$, where $\boldsymbol{p}$ is a vector of independent variables.

In GADfit, the forward mode comes into play in only one place, which are the accelerations terms (4) and (5), which form a second order directional derivative. The generalization of the forward mode to second order is straightforward. The AD numbers now contain three components,

$$\tilde{x} = x + \dot{x}d + \ddot{x}d^2, \tag{26}$$

and the arithmetic is defined by requiring third and higher order terms to be zero, $d^3 = 0$. The elemental operations become somewhat more complex,

$$\begin{aligned}
\tilde{x} + \tilde{y} =& x + y + (\dot{x} + \dot{y})d + (\ddot{x} + \ddot{y})d^2, \\
\tilde{x}\tilde{y} =& xy + (\dot{x}y + x\dot{y})d + (\ddot{x}\dot{y} + 2\dot{x}\dot{y} + x\ddot{y})d^2, \\
\frac{\tilde{x}}{\tilde{y}} =& \frac{x}{y} + \frac{\dot{x}y - x\dot{y}}{y^2}d + \frac{\ddot{x}y - x\ddot{y} - 2\dot{x}\dot{y} + 2\frac{x\dot{y}^2}{y}}{y^2}d^2, \\
\sin\tilde{x} =& \sin x + \dot{x}\cos x d + (\ddot{x}\cos x - \dot{x}^2 \sin x)d^2,
\end{aligned} \tag{27}$$

but otherwise the algorithm has the same structure. The complexity is actually less than appears in Eqs. (27), since once the first component of the resulting number has been calculated, it can be immediately used in the calculation of the second component. And for the third component, both of the previous terms are available. For instance, the last two of Eqs. (27) can be rewritten in the form

$$\begin{aligned}
\tilde{v} =& \frac{\tilde{x}}{\tilde{y}} = v + \frac{\dot{x} - v\dot{y}}{y}d + \frac{\ddot{x} - v\ddot{y} - 2\dot{v}\dot{y}}{y}d^2, \\
\tilde{v} =& \sin\tilde{x} = v + \dot{x}\cos x d + (\ddot{x}\dot{v}/\dot{x} - \dot{x}^2 v)d^2.
\end{aligned} \tag{28}$$

In order to calculate the second directional derivative $\partial_i \partial_j f(\boldsymbol{p}) v^i v^j$, where $\partial_i \equiv \partial/\partial p_i$, at $\boldsymbol{P}$, the independent variables should seeded with $(P_i, v_i, 0)$. A single sweep then produces the function value and the directional first and second derivatives at $\boldsymbol{P}$.

2.3.2 Reverse mode

The reverse mode is generally considered superior to the forward mode, because it is more efficient for calculating the gradient, which is more often required than the directional derivative. The Jacobian is exactly such a quantity, where each row $(J_{i1}, \ldots, J_{in})$ is the gradient of a single-valued function $f(x_i, \boldsymbol{\beta})$. At the same time, it is also harder to implement, since it consists of two parts — the forward sweep, which performs function evaluation, and the return sweep, which processes the computational graph in the reverse order to yield the partial derivatives. Information about the elemental operations must be saved during the forward sweep, which necessitates some form of memory management. This is in contrast to the forward mode, where both the function value and the derivatives are obtained by traversing in the same direction along the computational graph and no additional memory needs to be used.

The reverse mode is best understood with an example. The sequence of elemental operations for the evaluation of $f(x, y) = xy + \sin\frac{x}{y}$ at the point $(2.3, 0.8)$, which we call the forward sweep, is

$$v_1 = x = 2.3000, \tag{29a}$$
$$v_2 = y = 0.8000, \tag{29b}$$
$$v_3 = v_1 v_2 = 1.8400, \tag{29c}$$
$$v_4 = v_1 / v_2 = 2.8750, \tag{29d}$$
$$v_5 = \sin v_4 = 0.2634, \tag{29e}$$
$$v_6 = v_3 + v_5 = 2.1034, \tag{29f}$$

where the first two operations are just variable initialization. While the order of some of these operations can be compiler specific, it does not influence the outcome of the AD method. One might suppose that in order to compute the gradient $\nabla_{x,y} f$, it is necessary to process each of Eqs. (29) twice, once for either parameter. However, the number of operations can be lowered by applying the chain rule backwards, i.e., by processing Eqs. (29) in the order $v_6 \ldots v_1$ in terms of the adjoint quantities $\bar{v}_i \equiv \partial f / \partial v_i$. This is called the return sweep. Two trivial results are then obtained from Eq. (29f):

$$\bar{v}_3 = \frac{\partial f}{\partial v_3} = \frac{\partial v_6}{\partial v_3} = 1, \qquad \bar{v}_5 = \frac{\partial f}{\partial v_5} = \frac{\partial v_6}{\partial v_5} = 1. \tag{30a}$$

Using the result for $\bar{v}_5$, Eq. (29e) then yields the adjoint of v_4,

$$\bar{v}_4 = \frac{\partial v_6}{\partial v_4} = \frac{\partial v_6}{\partial v_5}\frac{\partial v_5}{\partial v_4} = \bar{v}_5 \cos v_4 = 1 \times (-0.9647) = -0.9647. \tag{30b}$$

Next, we process the two components of Eq. (29d),

$$\bar{v}_1 = \bar{v}_4 \frac{\partial v_4}{\partial v_1} = \frac{\bar{v}_4}{v_2} = -1.2059, \quad \bar{v}_2 = \bar{v}_4 \frac{\partial v_4}{\partial v_2} = -\frac{\bar{v}_4 v_1}{v_2^2} = 3.4669. \tag{30c}$$

From Eq. (29c),

$$\bar{v}_1 = \bar{v}_1 + \bar{v}_3 \frac{\partial v_3}{\partial v_1} = -0.4059, \quad \bar{v}_2 = \bar{v}_2 + \bar{v}_3 \frac{\partial v_3}{\partial v_2} = 5.7669. \tag{30d}$$

The computation of $\bar{v}_i$ is cumulative. If some variable v_i appears several times in the forward sweep then all occurrences of v_i must be accounted for. This is seen in Eqs. (30d), which remember the previous values of $\bar{v}_1$ and $\bar{v}_2$. With the above scheme, Eqs. (29a) and (29b) would trivially yield $\bar{v}_1 = \bar{v}_1$ and $\bar{v}_2 = \bar{v}_2$ and are not processed in practice. We thus have the result

$$\nabla_{x,y} f(x, y) = (-0.4059, 5.7669). \tag{31}$$

For such a simple example finite differences would of course produce the same result. However, some benefit of the reverse mode is already seen — a single sweep produces the partial derivatives of *both* variables. This property becomes especially advantageous for optimization problems with a large number of fitting parameters.

The algorithm can now be formulated as shown in Table 1. The first n oper-

Table 1: The reverse mode of AD. $a \mathrel{+}= b$ is a shorthand for $a = a + b$.

Forward sweep	Return sweep
$v_i = x_i, \quad i = 1 \dots n$	$\partial f \partial x_i = \bar{v}_i, \quad i = 1 \dots n$
$v_i \mathrel{+}= g_i(v_{j \rightsquigarrow i}), \quad i = n+1 \dots n+m$	$\bar{v}_i \mathrel{+}= \bar{v}_{j \rightsquigarrow i} \partial v_{j \rightsquigarrow i} / \partial v_i$
$f = v_{n+m}$	$\bar{v}_{n+m} = \partial f / \partial v_{n+m} = 1$

ations of the forward sweep initialize n independent variables. The intermediate values v_i are then calculated by applying the elemental operations g_i to all v_j of which v_i directly depends on (denoted with the $\rightsquigarrow$). The last one of the m intermediate values is the function value, $f = v_{n+m}$. Information about each elemental operation during the forward sweep must be saved as it is required during the return sweep. The return sweep then moves in the opposite direction, starting with $\partial f / \partial v_{n+m} = \partial f / \partial f = 1$ and followed by the calculation of $\bar{v}_i$ as illustrated in the above example. The partial derivatives are given by the first n values of the adjoints array $\bar{\boldsymbol{v}}$.

2.3.3 Implementation

GADfit is based on operator overloading, where the basic variable is the "AD variable",

```
type advar
   real(kp) :: val, d = 0.0_kp, dd = 0.0_kp
   integer :: index = 0
end type advar
```

This variable type is used for both forward and reverse modes. Each variable of this type has fields for the value and first and second derivatives, and an index

field. The former three are double or quadruple precision floating point numbers. Index is zero for "passive" variables, for which differentiation is not required, and is nonzero otherwise. Which mode is currently in use is determined by whether the global variable `reverse_mode` is true or false. Whereas the `d` and `dd` fields are characteristic to the forward mode only, the index field is required for both modes. For the forward mode, the reason is that the index can be used to immediately identify whether the variable is passive or active and thus many calculations involving `d=0.0` and `dd=0.0` can be avoided. In the forward mode, it only matters whether the index is zero or nonzero, while in the reverse mode, each new active variable is given a unique index.

The reverse mode is complicated by the fact that during the forward sweep each elemental operation[2] must be recorded by saving the indices of the participating variables and of the operation code into an array called the execution trace. The intermediate values are saved into a separate array, and an additional array is required for those operations that produce a constant which is required during the return sweep (such as multiplication with a constant). As an example, if the variables a and b had indices 4 and 5, and the operation code for addition were 23, then for the operation $c = a+b$ that section of the trace would read $(\ldots, 4, 5, 6, 23, \ldots)$, where 6 is the index of the new variable c. For unary operations, there would be 3 entries in the trace, $(\ldots, \text{in}, \text{out}, \text{opcode}, \ldots)$, for ternary operations 5 entries and so on. At the beginning of each return sweep, the adjoints array is reinitialized to zero with the last element set equal to 1. The execution trace is then processed starting with the last element, and the adjoints are calculated according to the AD algorithm, first by identifying the relevant operation (`case select(op_id)`). To summarize, the main AD work variables in the reverse mode, along with their internal names, are

- the intermediate values (`forward_values`),
- the adjoints (`adjoints`),
- the execution trace (`trace`),
- the constants (`ad_constants`).

No work variables are required for the forward mode. The current list of elemental operations includes

- $x_1 + x_2, \quad x_1 - x_2, \quad x_1 x_2, \quad \frac{x_1}{x_2}, \quad x_1^{x_2}, \quad |x_1|, \quad \ln x_1,$
- $\sin x_1, \quad \cos x_1, \quad \tan x_1, \quad \sinh x_1, \quad \tanh x_1, \quad \operatorname{asin} x_1, \quad \operatorname{acos} x_1, \quad \operatorname{atan} x_1, \quad \operatorname{asinh} x_1, \quad \operatorname{acosh} x_1, \quad \operatorname{atanh} x_1,$
- $\operatorname{erf}(x_1) = \frac{2}{\sqrt{\pi}} \int_0^{x_1} e^{-t^2} \, \mathrm{d}t, \quad \operatorname{Li}_2(x_1) = -\int_0^{x_1} \frac{\ln(1-u)}{u} \, \mathrm{d}u,$
- $\int_{a(x_1,\ldots,x_n)}^{b(x_1,\ldots,x_n)} f(x_1, \ldots, x_n; t) \, \mathrm{d}t,$

[2]Not to be confused with the Fortran "elemental" keyword. In fact, these are all impure non-elemental functions.

where x_i is either an AD variable, a real number of single/double/quadruple precision, or an integer. Note that the list also covers $-x_1$, $\exp(x_1)$, and $\sqrt{x_1}$. Integrals can be nested up to two layers, i.e., only single and double integrals are allowed. New elemental operations can be added to the source code without too much effort. Alternatively, any requests/suggestions are most welcome at `https://github.com/raullaasner/gadfit/issues`. Unless the proposed elemental operation depends on a nonstandard external library, we will make an effort to include it in the next release.

2.3.4 AD with numerical integration

GADfit supports fitting functions containing bounded, semi-infinite, and infinite integrals. Numerical integration is performed using the adaptive Gauss-Kronrod algorithm. Default is the 15-point rule; also available are 21, 31, 41, 51, and 61 point rules. The integration interval is divided into subintervals, and on each iteration the subinterval with the largest error is processed. This is similar to `QUADPACK` except without the epsilon algorithm. During this procedure, AD is temporarily switched off by making all variables passive. When the integration procedure has converged with the desired accuracy, AD is switched on and the integral is computed once more with an optimal set of subintervals. Note that the optimal set of subintervals satisfies the error tolerances of the original integral, not necessarily the integral over the derivative of the integrand. One might suppose then that the procedure needs to be performed separately for each parameter, presumably using the forward mode of AD, yielding a unique set of subintervals for each parameter. Strictly speaking though, the derivative is required for the approximation

$$F(x) = \int_A^B f(x,t)\,\mathrm{d}t \approx \sum_j \frac{b_j - a_j}{2} \sum_i w_{i,j} f\left(x, \frac{b_j - a_j}{2}\xi_{i,j} + \frac{a_j + b_j}{2}\right), \quad (32)$$

where a_j and b_j are the bounds of each subinterval, $w_{i,j}$ are the Gauss-Kronrod weights, and $\xi_{i,j}$ are the roots of the Legendre polynomials. Since it is Eq. (32) that is the fitting function (or part of it) and not the exact integral, differentiation should be performed using the same approximate form instead of finding a more accurate approximation for the integral over the derivative of f by using a different set of subintervals.

The dependence on one or more fitting parameters can be in the integrand and/or one or both bounds. The general formula for the gradient is

$$\begin{aligned}\nabla_{\boldsymbol{p}} &\int_{a(\boldsymbol{p})}^{b(\boldsymbol{p})} f(\boldsymbol{p}, s)\,\mathrm{d}s \\ &= f(\boldsymbol{p}, b(\boldsymbol{p}))\nabla_{\boldsymbol{p}} b(\boldsymbol{p}) - f(\boldsymbol{p}, a(\boldsymbol{p}))\nabla_{\boldsymbol{p}} a(\boldsymbol{p}) + \int_{a(\boldsymbol{p})}^{b(\boldsymbol{p})} \nabla_{\boldsymbol{p}} f(\boldsymbol{p}, s)\,\mathrm{d}s, \quad (33)\end{aligned}$$

where $\boldsymbol{p}$ is the fitting parameter vector.

The general formula for the second derivative, required for the acceleration term, is

$$\begin{aligned}\nabla_{\boldsymbol{p}}^2 \int_{a(\boldsymbol{p})}^{b(\boldsymbol{p})} f(\boldsymbol{p}, s)\,\mathrm{d}s &= f(\boldsymbol{p}, b(\boldsymbol{p}))\nabla_{\boldsymbol{p}}^2 b(\boldsymbol{p}) - f(\boldsymbol{p}, a(\boldsymbol{p}))\nabla_{\boldsymbol{p}}^2 a(\boldsymbol{p}) \\ &\quad + \left[\nabla_{\boldsymbol{p}} f(\boldsymbol{p}, b(\boldsymbol{p})) + \nabla_{\boldsymbol{\chi}} f(\boldsymbol{\chi}, b(\boldsymbol{p}))|_{\boldsymbol{\chi}=\boldsymbol{p}}\right] \nabla_{\boldsymbol{p}} b(\boldsymbol{p}) \\ &\quad - \left[\nabla_{\boldsymbol{p}} f(\boldsymbol{p}, a(\boldsymbol{p})) + \nabla_{\boldsymbol{\chi}} f(\boldsymbol{\chi}, a(\boldsymbol{p}))|_{\boldsymbol{\chi}=\boldsymbol{p}}\right] \nabla_{\boldsymbol{p}} a(\boldsymbol{p}) + \int_{a(b)}^{b(\boldsymbol{p})} \nabla_{\boldsymbol{p}}^2 f(\boldsymbol{p}, s)\,\mathrm{d}s, \quad (34)\end{aligned}$$

Equation (34) might seem complicated, but most terms are actually evaluated with little effort. The only difficulty is with the terms $\nabla_{\boldsymbol{p}} f(\boldsymbol{p}, \xi(\boldsymbol{p}))$, where ξ is one of the bounds. The reason is that the second argument to f, the integration variable, is required to be an ordinary floating point number. In order to calculate the derivative with respect to this variable using AD, it would need to be passed as an AD variable. In principle, this is not an issue, but it is a major technical inconvenience. If the interface of f is changed to allow this, then a lot of the internal procedures of GADfit would need to be modified to include an intermediary dummy AD variable for passing arguments to f (Fortran does not allow passing derived types by value). Most of the time this would be an unnecessary conversion from real to a passive AD variable, and only for the above mentioned two terms would the argument need to be an active AD variable. This would incur massive overhead, resulting in an increase of the walltime by up to 20%, even if one is not interested in the acceleration term. As a solution, this is the only place where we use finite differences for evaluating the derivative. The error introduced is independent of the error of the numerical integration procedure and the effect on the final result should be negligible.

The above scheme could be generalized to any order, but it is probably not worth going deeper than double integrals since each integration level greatly reduces accuracy. Even double integrals should be avoided if possible.

3 Using GADfit

GADfit is a library, i.e., not a standalone program, written in the Fortran language, meaning that ideally the user had at least basic knowledge of Fortran. Specifically, the fitting functions and the driver code would need to be written in Fortran. The structure of the code is such that it is currently not possible to easily interface it with other languages. Also, it would be inefficient to have some sort of parser for translating user input into Fortran functions. The Fortran requirement might seem inconvenient for the user, but it allows to make full use of a powerful programming language for writing functions of arbitrary complexity, which is what AD is designed for.

Section 3.3, which contains a tutorial covering the basic usage of GADfit, is aimed at all potential users, including those not familiar with Fortran.

3.1 Compilation

GADfit is dependent on the CMake build system generator. CMake is a set of tools for configuring, building, and testing software. As opposed to the commonly used GNU Autotools, it uses a simpler syntax and generally runs much faster. The prerequisites for installing GADfit are CMake and a Fortran compiler. CMake, released under the New BSD License, can be installed by issuing

```
sudo apt-get install cmake
```

(Here and elsewhere, we demonstrate how to obtain the relevant packages specifically on Ubuntu, which must not be older than version 16.04; more advanced users can probably skip this section.) For building from source visit `http://cmake.org`. GADfit is developed with the requirement that it should work with at least the GNU Fortran compiler (GFortran). Currently, there is also support for the Intel compiler. In principle, any other Fortran 2008 conforming compiler should also work, but there is no guarantee. GFortran, released under the GPL 3+ license, can be obtained with

```
sudo apt-get install gfortran
```

or one could build from source by visiting

```
http://gcc.gnu.org/wiki/GFortran.
```

Steps for configuring and building GADfit are the following. Untar the source code and navigate into the build directory (`~build`)

```
tar xf gadfit.tar.gz
mkdir build && cd build
```

One can also build in the source directory (`~gadfit`) but it is generally a good habit to do out-of-source builds or at least create a separate build directory within the source directory.

For configuring, there is the choice of using either a graphical CMake front end, a text file containing the configuration variables, or specifying the build environment from the command line (`cmake <options> ...`) in analogy to Autotools. The latter is, however, inconvenient and will not be discussed here. The two commonly used graphical front ends are the command line based `ccmake`, obtained by installing `cmake-curses-gui`, and the Qt-based `cmake-gui`, obtained by installing `cmake-qt-gui`. When using `ccmake`, issue

```
cmake ~gadfit
ccmake .
```

from the build directory. Some CMake variables and options appear, most of which should be self-explanatory. A short help text to each variable is displayed at the bottom in a status bar. Pressing 't' reveals all options. When done editing, press 'c' to reconfigure and 'g' to generate the native build script (the unix makefile on Linux). Pay attention when `ccmake` warns you that the cache

variables have been reset. This will happen, e.g., when changing the compiler, and will necessitate the reconfiguring of some variables. After configuring, it is a good idea to go through all the variables once more to check that everything is correct. When finished, exit `ccmake` and issue

```
cmake --build .
cmake --build . --target test
cmake --build . --target install
```

Additionally, `cmake --build . --target doc` may be issued for compiling the user guide, although a prebuilt one is available at `https://raullaasner.github.io/gadfit/doc/user_guide/user_guide.pdf`.

To clarifiy, when we issue `cmake --build`, CMake generates a build system, which could be the GNU makefiles (the default on most system) or some other build system. For using Ninja, use `-G Ninja` during the initial configuring, e.g.,

```
cmake -G Ninja ~gadfit
```

A Fortran compatible version of Ninja is maintained by Kitware and is available at `https://github.com/Kitware/ninja`.

In order to use `cmake-gui` instead of `ccmake`, start with

```
cmake ~gadfit
cmake-gui .
```

The rest is analogous.

The CMake configuration variables can also be put into an external file, an example of which is `initial_cache.cmake` in the root directory of GADfit. It contains a description and example initial values for the most important configuration variables. When done editing, run

```
cmake -C ~gadfit/initial_cache.cmake ~gadfit
```

from the build directory, followed by the `make` commands. A change of the content of the configuration file has no effect after the first configuring. Instead, it would be necessary to empty `~build` before using the configuration file again or, better, use `ccmake`.

When using GADfit, i.e., when linking to an executable code (such as when running the tests), the two additional dependencies (OpenCoarrays and a linear algebra library; see below) are by default automatically built when needed.

3.1.1 Parallelism

If the derivatives were calculated with finite differences, using `do concurrent` would be sufficient to parallelize most of GADfit. However, with AD, both function evaluation and the gradient calculation become impure procedures, which means that a simple parallelization scheme is not possible. GADfit is parallelized using Coarrays, a syntactic extension to the language that was included in the Fortran 2008 standard. Coarrays follow the SPMD parallelization scheme,

where the main program is replicated at the start of execution. Each instance, called an image, has its own set of private variables plus so-called co-variables that are addressable by other images. Coarrays allow to parallelize programs with little effort compared to MPI or OpenMP, although it is often still MPI that runs under the hood.

In order to compile a Coarray program with GFortran, the compiler flag `-fcoarray=lib` must be used. For GFortran, the Coarray support is provided by the OpenCoarrays project. It can be automatically downloaded and built with `cmake --build . --target example` or `cmake --build . --target test`. The Coarray communication library, `libcaf_mpi.a`, is installed in `~gadfit/tests`. Alternatively, one may issue, e.g.,

```
wget https://github.com/sourceryinstitute/opencoarrays/\
archive/2.8.0.tar.gz
tar xf 2.8.0.tar.gz
mkdir OpenCoarrays-2.8.0/build && cd OpenCoarrays-2.8.0/build
cmake -DCMAKE_Fortran_COMPILER=mpif90 ..
make -j caf_mpi_static # libcaf_mpi.a is in
                       # OpenCoarrays-2.8.0/build/lib
```

and set the `OPEN_COARRAYS` variable in the GADfit configuration to the full path of `libcaf_mpi.a`. If using the Intel compiler, there is no such external dependency.

3.1.2 Runtime linking

This section, for those new to Linux, provides a quick example of how to compile and run a Fortran program with calls to GADfit. Assuming we have successfully compiled GADfit as a shared library and created an input file `main.f90`, the following script, if called from the directory where `main.f90` resides, should successfully compile and execute. How to write an input file is explained in Sec. 3.3.

```
#!/bin/bash

GAD=/usr/local/gadfit

# GNU
FC=mpif90
FFLAGS="-J$GAD/include -L$GAD/lib"
LIBS="$LIBS /usr/local/opencoarrays/lib/libcaf_mpi.a"
# Intel
#FC=ifort
#FFLAGS="-coarray=shared -I$GAD/include -L$GAD/lib"

# Lapack
LIBS="$LIBS -llapack"
```

```
# Atlas
#FFLAGS="$FFLAGS -L/usr/local/atlas/lib"
#LIBS="$LIBS -ltatlas"

# MKL (for Intel and GNU compilers)
#MKLROOT=/opt/intel/mkl
#FFLAGS="$FFLAGS -L$MKLROOT/lib/intel64"
#export LD_LIBRARY_PATH=$MKLROOT/lib/intel64:$LD_LIBRARY_PATH
#LIBS="$LIBS -lmkl_intel_lp64 -lmkl_sequential -lmkl_core"
#LIBS="$LIBS -lmkl_gf_lp64 -lmkl_sequential -lmkl_core -lpthread"

export LD_LIBRARY_PATH=$GAD/lib:$LD_LIBRARY_PATH
$FC $FFLAGS main.f90 -o main -lgadfit $LIBS && \
mpirun -np 4 ./main # GNU
#./main             # Intel
```

This may need to be modified to reflect the user's environment. If `libcaf_mpi.a` was built automatically (see Sec. 3.1.1), then its location is `~gadfit/tests`. If GADfit was built in serial, `libcaf_mpi.a` need not be linked.

3.2 Using GADfit in other CMake projects

If you have built and installed GADfit and want to use it in a CMake project, only two CMake statement are necessary. Assuming that GADfit has been installed at a standard system location and the target name you want to link against GADfit is `myproject`, include the lines

```
find_package(gadfit)
target_link_libraries(myproject PRIVATE gadfit::gadfit)
```

in that project. If GADfit is not installed in a standard location, use the `PATHS` argument to `find_package`, e.g.,

```
find_package(gadfit PATHS $ENV{HOME}/libs/gadfit)
```

If additionally you need to ensure that GADfit is not older than a specific version, use

```
find_package(gadfit 1.3 PATHS $ENV{HOME}/libs/gadfit)
```

In this example, if the version of the installed copy is less than 1.3, CMake will exit with an error. Any version equal to or greater than 1.3 is fine. In order to see all the options of `find_package`, run

```
cmake --help find_package
```

or consult the online documentation.

If you want to link against GADfit without installing it first, the `PATHS` argument can point directly to the GADfit build directory:

```
find_package(gadfit PATHS $ENV{HOME}/builds/gadfit)
```

This is useful for developers.

3.3 Example input

Suppose we have made two measurements of a decay process, which are to be fitted against

$$I(t) = I_0 e^{-t/\tau} + bgr. \tag{35}$$

The decay curves correspond to the same physical phenomenon, meaning they share the same decay time τ, but have been performed in slightly different experimental conditions with different initial amplitudes I_0 and backgrounds bgr. To get the best estimate for the fitting parameters, the curves should be fitted simultaneously with τ as a global fitting parameter and I_{01}, I_{02}, bgr_1, and bgr_2 as local parameters. The data files `example_data1` and `example_data2` along with an example input, `example.f90`, that solves the nonlinear least squares problem, are found in `~gadfit/tests` and are copied to `~build/tests` after when invoking `cmake --build . --target example`. The data files are required to contain at least two columns (the rest are ignored). Any line beginning with a non-number is treated as a comment. In the following, we have chosen to put the fitting function into a Fortran module, followed by the main program which contains a small set of commands to perform the fitting procedure. Users with more knowledge of Fortran may wish to do it differently. We shall first describe how to build the fitting function.

3.3.1 Defining the fitting function

In GADfit, all user defined functions are derived from an abstract type `fitfunc` (making them *derived classes* in OOP terms). Two procedures must always be defined: a constructor, which specifies the number of fitting parameters, and the function body. While the user is advised to work through a full Fortran tutorial, in the following some effort has been made to explain the peculiarities of the language.

```
module test_f
  ! This module defines the function I(x) = I0*exp(-t/tau)+bgr.
  ! In Fortran, comments begin with an "!".

  use ad ! Imports the module containing the procedures for AD.
         ! Similar to the preprocessor #include directive.
  use fitfunction ! Imports the abstract class for the fitting
                  ! function.
  use gadf_constants ! Imports the floating point precision
                     ! specifier (kp).

  implicit none ! This should be present in all modules.
```

```
  ! We have decided to call the type of the new fitting function
  ! 'exponential'. init and eval are the constructor and the
  ! function body, which must be renamed and implemented below.
  type, extends(fitfunc) :: exponential
   contains
     procedure :: init => init_exponential
     procedure :: eval => eval_exponential
  end type exponential

contains
  ! The interfaces of the following procedures, i.e., the type of
  ! the arguments and of the return value, are fixed.
  subroutine init_exponential(this)
    class(exponential), intent(out) :: this
    ! Allocate memory to the fitting parameter array.
    allocate(this%pars(3))
    ! In C++, this would be something like
    ! this->pars = new advar[3];
  end subroutine init_exponential

  type(advar) function eval_exponential(this, x) result(y)
    class(exponential), intent(in) :: this
    real(kp), intent(in) :: x
    y = this%pars(1)*exp(-x/this%pars(2)) + this%pars(3)
    ! In this context, I0 = this%pars(1), tau = this%pars(2), and
    ! bgr = this%pars(3).
  end function eval_exponential
end module test_f
```

This might seem complicated at first, but have a look at `~gadfit/tests/function_template`, which is a stripped-down version of the above. The angle-bracketed words need to be replaced by user-defined values and the only effort lies in the description of the function body. In order to test whether the function has been correctly defined, one can use the same code as in the next section, but comment out the call to `gadf_fit`. If there are no data sets, then `gadf_print` must be explicitly supplied with the arguments `begin` and `end`, which define the argument range.

The above is a very simple example highlighting only some of the functionality. For more complex examples, see the tests in `~gadfit/tests`.

As a comment, we have used the module approach because the user might wish in the future to put the most commonly used fitting functions into a single module that can then be conveniently included, e.g., as a library in any source file. This is just a suggestion.

3.3.2 Fitting procedure

These are the commands necessary to run the example. For a complete overview of input variables, see Sec. 3.4.

```
use test_f ! Include the above-defined fitting function
use gadfit ! and the main library.

implicit none

type(exponential) :: f ! An instance of the fitting function

! Initialize GADfit with the fitting function and the number of
! data sets.
call gadf_init(f, 2)

! Include both decay curves. The argument must be full or
! relative path to the data. TESTS_BLD is expanded by the
! preprocessor to ~build/tests.
call gadf_add_dataset(TESTS_BLD//'/example_data1')
call gadf_add_dataset(TESTS_BLD//'/example_data2')

! The initial guess for all fitting parameters is 1.0. The
! first argument denotes the data set, the second argument the
! parameter, third argument its value, and fourth whether the
! parameter is allowed to vary or is kept fixed.
call gadf_set(1, 1, 1.0, .true.) ! I01
call gadf_set(2, 1, 1.0, .true.) ! I02
call gadf_set(1, 3, 1.0, .true.) ! bgr1
call gadf_set(2, 3, 1.0, .true.) ! bgr2
! Global parameters don't have the data set argument.
call gadf_set(2, 1.0, .true.)    ! tau

! The uncertainties of the data points determine their
! weighting in the fitting procedure. Here we are assuming shot
! noise, i.e., the error of each data point is proportional to
! the square root of its value. Default is no weighting.
call gadf_set_errors(SQRT_Y)

! Perform the fitting procedure starting with lambda=10. If the
! procedure doesn't converge, we should restart with a higher
! value or modify any of the other arguments to gadf_fit. All
! the arguments are optional with reasonable default values.
call gadf_fit(lambda=10.0)

! The results are saved into ~gadfit/tests
call gadf_print(output=TESTS_BLD//'/example_results')
```

```
  call gadf_close() ! Free memory
end program
```

In order to run the above code, one can manually link and compile from the command line, use a small script like in Sec. 3.1.2, or simply issue

```
cmake --build . --target example
```

from the build directory. If no linear algebra library has been specified, it will take a moment to build one.

The results of the calculation will be printed to the standard output and also to `~gadfit/tests/example_parameters`. The decay time should be about 20.5 units. If the errors are the true experimental uncertainties, then for a good fit the reduced sum of squares should be close to 1. The results can be visualized opening gnuplot in the directory `~build/tests` and issuing

```
p 'example_data1' t 'data1', 'example_data2' t 'data2', \
'example_results' u 1:2 w l t 'fit1', '' u 1:3 w l t 'fit2'
```

The rest of the examples in `~gadfit/tests` can be built and run with

```
cmake --build . --target test
```

The fitting functions used in the tests are

$$
\begin{aligned}
f_1(x) =& f_{\max} e^{-(x-x_0)^2/a^2} + bgr, \\
f_2(x) =& \pi \int_0^x t^a e^{-bt^2} \, \mathrm{d}t, \\
f(x) =& \int_0^\infty \mathrm{d}t \int_0^{x/b} \mathrm{d}y \frac{\ln[(e^y - 1)(1 + ab \operatorname{erf} t) + 1]}{xy} e^{-t},
\end{aligned}
$$

where erf is the error function.

3.4 Input

Before using GADfit, here are a some general remarks about argument processing in a Fortran.

- There is no automatic argument type conversion when calling a procedure. If the procedure expects a double precision number, then this is exactly what the user must supply. In C terms, if the function expects a 'double', then a 'float' will cause an error. While it can be compiler dependent, numbers such as 3.2 or 5e7 are generally interpreted as single precision. In GADfit, higher precision can be obtained by appending numbers with the "kind parameter" (`3.2_kp`, `5e7_kp`) in which case they become double or quadruple precision, depending on how GADfit was configured. With double precision, instead of `_kp` we can also use the `d` descriptor (`3.2d0`, `5d7`, `1d0`, etc.).

Internally, most procedures use at least double precision. However, for user convenience, some procedures are also available in single precision. For example, instead of

```
call gadf_set(1, 3.2_kp)
```

we can also have

```
call gadf_set(1, 3.2)
```

Single precision is not that important when setting initial parameter values. (The decimal point or an exponent must still be present, else the argument is interpreted as an integer.)

- Procedure arguments can be given as a value (`call gadf_fit(1.0)`) or as a name-value pair (`call gadf_fit(lambda=1.0)`). When using the latter option, the arguments can be given in any order. It is also useful when calling a procedure that has many optional arguments but the user wishes to provide only some of them.

The next section contains a detailed list of all procedures that are required for normal use. The procedure arguments are given according to the format `<type-specifier>::<variable>`, where the type-specifier can be `real(kp)` – double or quad precision real number; `real(real32)` – single precision real number; `integer` – integer; `character(*)` – character array (`'asdf'`); `logical` – truth value, can be either `.true.` or `.false.`. If there is more than one allowed type for the argument, the choices are separated by a slash. Optional arguments have the keyword `optional`. All procedures are subroutines except `integrate`, which is a function.

3.4.1 User controlled routines

```
gadf_init(f, num_datasets, ad_memory, sweep_size, trace_size,
          const_size, rel_error, rel_error_outer, ws_size,
          ws_size_inner, integration_rule)
```

Initializes the workspace. This procedure must always be called when performing a fitting procedure.

- `class(fitfunc) :: f`

  The fitting function. `class(fitfunc)` denotes a polymorphic variable. In practice, `f` has an explicit type like `type(exponential)` in the example.

- `integer, optional :: num_datasets`

  Number of data sets.

  Default: 1.

- `character(*), optional :: ad_memory`

  Allocates memory for the work variables of the AD reverse mode so that `forward_values` and `adjoints` contain x, `trace` $4x$, and `ad_constants` $\frac{1}{2}x$ elements, where x is such that the total amount of memory reserved is equal to that given by `ad_memory`, whose format is "number unit", where "unit" is B, kB, MB, or GB (also acceptable are b, kb, mb, or gb). Example: `ad_memory='1.25 MB'`. Overrides `sweep_size`, `trace_size`, and `const_size`, which can be used for fine-tuning memory allocation.

  Default: none.

- `integer, optional :: sweep_size`

  Size of `forward_values` and `adjoints`.

  Default: 10 000.

- `integer, optional :: trace_size`

  Size of `trace`.

  Default: $4 \times 10\,000$.

- `integer, optional :: const_size`

  Size of `ad_constants`.

  Default: $\frac{1}{2} \times 10\,000$.

- `real(kp), optional :: rel_error`

  Tolerance for the relative error of numerical integration. When dealing with double integrals, this applies to the outer integral.

  Default (single integrals): 100 × machine epsilon or
  Default (double integrals): 1000 × machine epsilon

- `real(kp), optional :: rel_error_inner`

  Tolerance for the relative error of the inner integral.

  Default: 100 × machine epsilon.

- `integer, optional :: ws_size`

  Size of the integration workspace. With double integrals this refers to the outer integral. It is assumed that there are at most this many integration subintervals.

  Default: 1000.

- `integer, optional :: ws_size_inner`

  Size of the integration workspace for the inner integral when using double integrals.

  Default: 1000.

- `integer, optional :: integration_rule`

  Sets the Gauss-Kronrod integration rule. Allowed values are `GAUSS_KRONROD_15P`, `GAUSS_KRONROD_21P`, `GAUSS_KRONROD_31P`, `GAUSS_KRONROD_41P`, `GAUSS_KRONROD_51P`, and `GAUSS_KRONROD_61P`.

  Default: `GAUSS_KRONROD_15P`.

`gadf_add_dataset(path)`

Reads a data set. This procedure must be called as many times as there are data sets, which is determined by the `num_datasets` argument to `gadf_init`.

- `character(*) :: path`

  Full or relative path to the data file.

`gadf_set(dataset_i, par, value, active)`

Defines the parameter as local, sets its value, and marks it either active or passive.

- `integer :: dataset_i`

  The data set index.

- `integer/character(*) :: par`

  Either the parameter index or its name.

- `real(real32)/real(kp) :: val`

  Parameter value in either single or higher precision.

- `logical, optional :: active`

  If `.true.` the parameter is active, else passive.

  Default: `.false.`.

`gadf_set(par, value, active)`

Similar to the other `gadf_set` except that this one defines either a global fitting parameter or a global constant. If only one data set is used, it doesn't matter which procedure is used.

```
gadf_set_verbosity(scope, digits, timing, memory, workloads,
                   delta1, delta2, cos_phi, grad_chi2, uphill,
                   acc, output)
```

Gives the user some control over how the results are displayed. All flags can be set to `.true.`. For instance, one might wish know $|\cos\phi|$ of Eq. (19) but not used it as a convergence criterion.

- `integer, optional :: scope`

  Whether to show only local or global parameters or both during the fitting procedure. Allowed values are `GLOBAL`, `LOCAL`, and `GLOBAL_AND_LOCAL`.

  Default: `GLOBAL_AND_LOCAL`.

- `integer, optional :: digits`

  How many significant digits of the fitting parameters are shown during the fitting procedure.

  Default: 7.

- `logical, optional :: timing`

  Whether to show the timing summary after the fitting procedure. The cpu and wall times are measured for $\boldsymbol{J}$, χ^2, $\boldsymbol{\Omega}$, and simple linear algebra operations ($\boldsymbol{J}^T$, $\boldsymbol{J}^T\boldsymbol{J}$, $\boldsymbol{J}^T(\boldsymbol{y}-\boldsymbol{f}), \ldots$). The cost of calculating $\boldsymbol{J}$ also includes the residual vector $\boldsymbol{y}-\boldsymbol{f}$, which is produced as part of the forward sweep. All other parts of the fitting procedure are serial. Since the serial portion of the code is negligible, all images should in principle do the same amount of work. The timing summary thus contains information about the load imbalance. It contains the following parts:

  - The cpu times per call are averaged over images and over the number of calls to the procedure.
  - The relative cost is the cpu time averaged over images and divided by the total cpu time spent in the main loop. With perfect scaling the relative costs should add up to 100%.
  - Detailed timing is shown for $\boldsymbol{J}$, χ^2, and $\boldsymbol{\Omega}$. The reported total cpu and wall times are over all serial and parallel parts of the main loop, and do not correspond exactly to the sum of the shown quantities.

  Default: `.false.`.

- `logical, optional :: memory`

  Whether to show memory usage after the fitting procedure. Only the peak usage is shown, which is not necessarily representative of the actual load.

  Default: `.false.`.

- `logical, optional :: workloads`

  Whether to show the workload of each image. If the loads are very uneven, it is worth considering using load balancing.

  Default: `.false.`.

- `logical, optional :: delta1`

  Whether to show $\boldsymbol{\delta}_1$, the velocity term.

  Default: `.false.`.

- `logical, optional :: delta2`

  Whether to show $\boldsymbol{\delta}_2$, the acceleration term.

  Default: `.false.`.

- `logical, optional :: cos_phi`

  Whether to show $|\cos\phi|$ [Eq. (19)].

  Default: `.false.`.

- `logical, optional :: grad_chi2`

  Whether to show $|\nabla\chi^2| = |2\boldsymbol{J}^T(\boldsymbol{y} - \boldsymbol{f})|$.

  Default: `.false.`.

- `logical, optional :: uphill`

  Whether to show $\cos(\boldsymbol{\delta}_i, \boldsymbol{\delta}_{i-1})$ [Eq. (7)].

  Default: `.false.`.

- `logical, optional :: acc`

  Whether to show the ratio of the contributions from the velocity and the acceleration terms [Eq. (6)].

  Default: `.false.`.

- `character(*), optional :: output`

  Where to send the output. Can be a file or, in general, any I/O device. On Linux, using `'/dev/null'` suppresses all output except errors and warnings, which are sent to stderr.

  Default: stdout (standard output).

```
gadf_set_errors(e)
```

Specifies the data point errors – σ_i in Eq. (3). If this procedure is not called, the errors are set to 1.

- `integer, optional :: e`

  Allowed values are `NONE` – $\sigma_i = 1$ (the default); `SQRT_Y` – $\sigma_i = \sqrt{y_i}$ (shot noise); `PROPTO_Y` – $\sigma_i = y_i$; `INVERSE_Y` – $\sigma_i = 1/y_i$; `USER` – user defined uncertainties, must be supplied as the 3rd column in the data files.

```
gadf_fit(lambda, lam_up, lam_down, accth, grad_chi2, cos_phi,
         rel_error, rel_error_global, chi2_rel, chi2_abs,
         lam_incs, uphill, max_iter, damp_max, nielsen, umnigh,
         ap)
```

Performs the fitting procedure. The procedure stops if any of the convergence criteria is satisfied.

- `real(real32), optional :: lambda`

  The adaptive damping parameter.

  Default: 1.

- `real(real32), optional :: lam_up`

  Factor by which λ is increased for accepted steps.

  Default: 10.

- `real(real32), optional :: lam_down`

  Factor by which λ is decreased for rejected steps.

  Default: 10.

- `real(real32), optional :: accth`

  Acceleration threshold [α in Eq. (6)]. If zero, the acceleration term is not calculated.

  Default: 0.

- `real(real32), optional :: grad_chi2`

  Tolerance for $|\nabla\chi^2| = |2\boldsymbol{J}^T(\boldsymbol{y} - \boldsymbol{f})|$.

  Default: 0.

- `real(real32), optional :: cos_phi`

  Tolerance for the cosine of the angle between the residual vector and the range of the Jacobian [Eq. (19)].

  Default: 0.

- `real(real32), optional :: rel_error`

  Tolerance for the relative change in any fitting parameter for two consecutive iterations.

  Default: 0.

- `real(real32), optional :: rel_error_global`

  Same as `rel_error` but applies only to global parameters.

  Default: 0.

- `real(real32), optional :: chi2_rel`

  Tolerance for the relative change in χ^2 for two consecutive iterations.

  Default: 0.

- `real(real32), optional :: chi2_abs`

  Tolerance for the reduced sum of squares.

  Default: 0.

- `real(kp), optional :: DTD_min(:)`

  Minimum values of the diagonal entries of the damping matrix.

  Default: 0.

- `integer, optional :: lam_incs`

  Number of times λ is allowed to increase consecutively without terminating the fitting procedure.

  Default: 2.

- `integer, optional :: uphill`

  The exponent b in Eq. (7). If zero, uphill steps are not allowed.

  Default: 0.

- `integer, optional :: max_iter`

  Iteration limit.

  Default: unlimited.

- `logical, optional :: damp_max`

  Whether to update the damping matrix with the largest entries of $J^T J$ yet encountered. If `.false.`, the classical Marquardt scheme is used.

  Default: `.true.`.

- `logical, optional :: nielsen`

  Whether to update λ with $\max[1/\lambda_\downarrow, 1-(2\rho-1)^3]$. See the paragraph on updating λ in Sec. 2.1. ρ includes only the velocity term.

  Default: `.false.`.

- `logical, optional :: umnigh`

  Whether to update λ according to Umrigar and Nightingale.

  Default: `.false.`.

- `logical, optional :: load_balancing`

  Whether to use load balancing. If `.true.`, the input data is redistributed among the images in an effort to achieve equal workloads. The redistribution is based on the workloads of the previous iteration and is repeated after each iteration. Load balancing makes sense with integrals and other constructs whose cost heavily depends on the argument.

  Default: `.false.`.

```
gadf_print(begin, end, points, output, grouped, logplot,
           begin_kp, end_kp)
```

Prints the results to an I/O device. The results include the theoretical curve(s), fitting parameter values, and summary of resources used. If no fitting procedure has been performed, only the theoretical curve is printed.

- `real(real32), optional :: begin`

  The lower bound for the x-values.

  Default: taken from the input data.

- `real(real32), optional :: end`

  The upper bound for the x-values.

  Default: taken from the input data.

- `integer, optional :: points`

  Number of points printed per curve.

  Default: 200.

- `character(*), optional :: output`

  Output file root name.

  Default: `'out'`.

- `logical, optional :: grouped`

  Whether to group all curves into a single file.

  Default: `.true.`.

- `logical, optional :: logplot`

  Whether to produce results suitable for a $(\log x, y)$-plot.

  Default: `.false.`.

- `real(kp), optional :: begin_kp`

  Same as `begin` but with double/quad precision; overrides `begin`.

  Default: none.

- `real(kp), optional :: end_kp`

  Same as `end` but with double/quad precision; overrides `end`.

  Default: none.

`gadf_close()`

Frees all work variables. After this, it is safe to call `gadf_init` again.

`integrate(f, pars, lower, upper, rel_error, abs_error)`

Performs numerical integration. Can be used as part of a fitting function. The result is of type `advar`.

- `type(advar) function f(x, pars)`
  `  real(kp), intent(in) :: x`
  `  type(advar), intent(in out) :: pars(:)`
  `end function f`

  The integrand.

- `type(advar) :: pars(:)`

  Parameters passed to the integrand.

- `type(advar)/real(kp)/integer :: lower`

  Lower integration bound. Can be an active/passive fitting parameter, a real number, `-INFINITY`, or `INFINITY`.

- `type(advar)/real(kp)/integer :: upper`

  Upper integration bound. Same comments as with `lower`.

- `real(kp), optional :: relative_error`

  Tolerance for the relative error of the integral. Overrides `gadf_init`.

  Default: whatever was specified with `gadf_init` or 100 (1000) $\times$ machine epsilon for the inner (outer) integral.

- `real(kp), optional :: absolute_error`

  Tolerance for the absolute error.

  Default: 0.

3.5 Internals

The previous section was about the main commands that would normally be used. This section describes more advanced usage of GADfit for experimental, debugging, or other purposes. Some knowledge of Fortran is assumed.

Basic usage of the `fitfunc` class. As was mentioned earlier, the user can test the fitting function by calling `gadf_print` without performing a fitting procedure. It is, however, possible to work directly with the fitting function, which inherits from the `fitfunc` class. The following code snippet prints the value of a function with the argument 2.0. It is assumed that GADfit has been compiled with double precision.

```
type(test) :: f              ! 'test' extends fitfunc.
real(dp) :: dummy
!call init_integration()     ! if necessary or
!call init_integration_dbl() ! if necessary.
call f%init()
call f%set(1, 1d0)           ! Has just one parameter.
dummy = f%eval(2d0)          ! Converts AD variable to real.
```

```
write(*,*) dummy              ! Function value at 2.0.
call f%destroy()
call free_integration()       ! Optional but never hurts.
```

Here and elsewhere, a dummy variable must be used since a component of a function result of derived type cannot be directly referenced in Fortran.

If the only aim is to print the function value(s) at some point(s), then the code can be slightly reduced by using the procedures of Sec. 3.4. After calls to `gadf_init` and `gadf_set`, multiple instances of the function, each with an independent set of parameters, are copied to the `fitfuncs` array, which is protected (read-only public). The above code then reduces to

```
type(test) :: f
real(dp) :: dummy
call gadf_init(f)
call gadf_set(1, 1d0)
dummy = fitfuncs(1)%eval(2d0)
write(*,*) dummy
call gadf_close()
```

Another reason for directly working with the `fitfunc` class is the calculation of derivatives using finite differences, which can be used for testing new AD elemental operations. In the next snippet, the derivatives are calculated with respect to parameters 1 and 3, while parameter 2 is passive.

```
type(test) :: f
real(dp) :: grad(2)
call f%init()
call f%set(1, 1.2d0)
call f%set(2, -0.1d0)
call f%set(3, 1d17)
call f%grad_finite(2d0, [1,3], grad)
write(*,*) grad
call f%destroy()
```

In order to calculate the 2nd directional derivative in the $(0.3, 0.7)$ direction in the space spanned by parameters 1 and 3, the above code can be modified by inserting

```
write(*,*) f%dir_deriv_2nd_finite(2d0, [1,3], [0.3d0,0.7d0])
```

after the parameters have been initialized. See the documentation of `grad_finite` and `dir_deriv_2nd_finite` in `fitfunction.f90` for a better understanding.

Both finite differences and AD are always available regardless of the value of `USE_AD` during configuration. This option only determines which one is used during the fitting procedure.

Basic usage of the AD forward mode. In the forward mode, parameters are made active by modifying both their `index` and `d` fields. In the interest of compactness, here we shall set the parameter values directly instead of using the more safe `set` procedure.

```
type(test) :: f
type(advar) :: dummy
call f%init()
f%pars = [1.2d0, -0.1d0, 1d17]
f%pars%index = [1,0,0]
f%pars%d = [1,0,0]
dummy = f%eval(2d0)
write(*,*) dummy%d
f%pars%index = [0,0,1]
f%pars%d = [0,0,1]
dummy = f%eval(2d0)
write(*,*) dummy%d
call f%destroy()
```

In order to calculate the 1st and 2nd directional derivatives in the $(0.3, 0.7)$ direction, the above code becomes

```
type(test) :: f
type(advar) :: dummy
call f%init()
f%pars = [1.2d0, -0.1d0, 1d17]
f%pars%index = [1,0,1]
f%pars%d = [0.3,0.0,0.7]
dummy = f%eval(2d0)
write(*,*) 'First  dir deriv:', dummy%d
write(*,*) 'Second dir deriv:', dummy%dd
```

Basic usage of the AD reverse mode. The reverse mode is a bit more difficult to use manually since the user has to be more acquainted with the internal work variables of `automatic_differentiation.f90`. In particular, it is necessary to set `index_count` to the number of active parameters (index of the next AD variable will be `index_count`+1), and to initialize the array of intermediate values. The gradient calculation with respect to parameters 1 and 3, as in the previous examples, can then be achieved as follows.

```
type(test) :: f
real(dp) :: dummy
call ad_init_reverse()
call f%init()
f%pars = [1.2d0, -0.1d0, 1d17]
f%pars%index = [1,0,2]
index_count = 2
```

```
  forward_values(1:2) = [f%pars(1)%val, f%pars(3)%val]
  dummy = f%eval(2d0)
  !write(*, '(g0)') trace(:trace_count)
  call ad_grad(2)
  write(*,*) adjoints(1:2)
  call f%destroy()
  call ad_close()
```

Uncommenting the line after function evaluation shows what the execution trace looks like after the forward sweep. If the function contains an integral, additional calls to `init_integration` or `init_integration_dbl` and `free_integration` are required.

Numerical integration. It is also possible to use GADfit just for evaluating integrals, although if this is the only purpose, there are better tools out there. The following example shows the calculation of $\int_{0.5}^{\infty} e^{-x}\,\mathrm{d}x$:

```
  type(advar) :: dummy, pars(0)
  call init_integration()
  dummy = integrate(f, pars, 0.5d0, INFINITY)
  print*, dummy%val
  call free_integration()
contains
  type(advar) function f(x, pars) result(y)
    real(kp), intent(in) :: x
    type(advar), intent(in out) :: pars(:)
    y = exp(-x)
  end function f
```

Creation of new elemental operations. New elemental operations can be introduced in `automatic_differentiation.f90`. The main steps are:

- A new operation code is created.
- A module procedure is overloaded.
- In the reverse mode, the operation body is supposed to do the following: the function body is evaluated; the newly created result variable is given a unique index; the result of evaluation is saved in `forward_values`; the indices of all arguments, that of the result variable, and the operation code are saved in `trace`; any constants are saved in `ad_constants`; `index_count`, `trace_count`, and `const_count` are increased accordingly. In the forward mode, the implementation is straightforward. In either case, if the argument is a passive variable, only the function body is evaluated. For a better understanding, have a look at the implementation of any unary operation in the source code.

- In the reverse mode, the calculation of the adjoints is implemented in the return sweep (`ad_grad`).

The function body may also refer to an external function. The dilogarithm function, Li_2, is an example of importing special functions from GSL. Li_2 is available if `GSL_DIR` points to the GSL root directory during configuration. This is not required for running any of the tests.

4 Troubleshooting

A good place to bring up any issues is
`https://github.com/raullaasner/gadfit/issues`.

References

[1] D. W. Marquardt, J. Soc. Indust. Appl. Math. **11**, 431 (1963).

[2] M. K. Transtrum, B. B. Machta, and J. P. Sethna, Phys. Rev. Lett. **104**, 060201 (2010).

[3] M. K. Transtrum, B. B. Machta, and J. P. Sethna, Phys. Rev. E **83**, 036701 (2011).

[4] M. K. Transtrum and J. P. Sethna, arXiv:1201.5885 [physics.data-an] (2012).

[5] H. B. Nielsen, Tech. Rep. IMM-REP-1999-05 (1999).

[6] A. Griewank and A. Walther, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, SIAM, USA, 2nd edition (2008).