Automatic Implicit Function Theorem

Dmitri Goloubentsev
Matlogica

Evgeny Lakshtanov
University of Aveiro
Matlogica

Vladimir V. Piterbarg
NatWest Markets

April 11, 2022

Abstract

The Implicit Function Theorem, or IFT, is a powerful tool for calculating derivatives of functions that solve inverse, i.e. calibration, problems prevalent in financial applications. It is commonly believed that a degree of manual intervention is required to enable financial code to take advantage of the IFT even when using Automatic Adjoint Differentiation (AAD). In this note we explain in mathematical terms, and demonstrate on a simple example with Python code, how the *Automatic IFT*, a special version of the IFT, enables fully-automated differentiation of exact-fit calibration routines. We show that the Automatic IFT gives an approximate solution to nearly-exact calibration problems typical in practice, where we also derive numerical stability estimates. Furthermore, we extend the approach to the general best-fit calibration set-up.

1 Introduction

Risk.net paper [4] introduced Automatic Adjoint Differentiation (AAD) to the financial industry, with AAD being a subject of intense research ever since, see for example [1–3, 6–9, 11, 12]. While relatively straightforward for direct function evaluations, the usage of AAD in inverse, or calibration, problems that are prevalent in finance requires significantly more care. It is generally understood, and well-articulated in e.g. [11], that using the Implicit Function Theorem (IFT) is preferred to applying AAD directly to non-linear solvers that calibration problems often require. However, there are subtleties in the use of IFT in conjunction with AAD that may not be as widely appreciated. In this paper, we demonstrate that the most straightforward application of the IFT (see e.g. [8, Section 2]) is the "wrong" way to use it.

We introduce a simple yet not well-known modification, that we call the *Automatic Implicit Function Theorem* (AIFT), of the procedure that makes it much more straightforward to incorporate into a typical AAD-enabled application. The main feature of our approach is that the explicit knowledge of the auxiliary variables involved in calculations, and derivatives with respect to them, is not required. That makes our approach possibly the only sensible way of introducing AAD in large and complex financial codebases. Although versions of this algorithm are known to experts, they are still not generally appreciated by the wider quant finance community with many still believing that the anticipated computational gains are impossible to achieve without manual coding based on domain-specific knowledge of the pricing process. The main contribution of this paper is in showing how to fully automate the application of IFT in a wide range of typical quant finance applications.

After explaining the "right" way to apply AAD to non-linear solvers, we also cover the case of best-fit optimizers, an example being the well-known Levenberg-Marquardt algorithm.

This paper provides annotated Python code that demonstrates our ideas directly. The code, and its extensions, is available at `https://github.com/piterbarg/aift/`. A C++ example based on `QuantLib`, an open-source financial library, is also available. One can set up a personal environment to explore it directly, see [10] for details.

2 The basics of AAD

To understand AIFT, our proposed refinement to the application of the IFT, it is instructive to explore the mechanics of AAD calculations in some detail. Suppose we have the following composite function:

$$x_2 = f_2(x_4, x_5), \tag{1}$$
$$x_1 = f_1(x_3, x_4), \tag{2}$$
$$x_0 = f_0(x_1, x_2), \tag{3}$$

where all the arguments are of the same dimension D for simplicity. The objective of AAD is to calculate, in addition to x_0, the derivatives of f_0 with respect to the "leaf" arguments x_3, x_4, x_5. We introduce the notation $\frac{\partial f}{\partial y}$ for the matrix with elements

$$\left(\frac{\partial f}{\partial y}\right)_{k,l} = \frac{\partial f_k}{\partial y_l}.$$

Using this notation, let us denote the adjoints with "bars"as customary:

$$\bar{x}_i \triangleq \bar{x}_0 \frac{\partial f_0}{\partial x_i},\ i = 1, \ldots, 5,$$

where these include intermediate, non-leaf, x_i as well. Here all the $\bar{x}_i$ are D-dimensional vectors, $\bar{x}_0$ is a fixed D-dimensional vector that we are free to choose and the operation on the right-hand side is the row-vector by matrix multiplication, that is in longhand

$$(\bar{x}_i)_k = \sum_{l=1}^{D} (\bar{x}_0)_l \frac{\partial (f_0)_l}{\partial (x_i)_k}, \quad k = 1, \ldots, D.$$

With the above notations, the AAD algorithm proceeds as follows.

Algorithm 1 (AAD) *Adjoint (backward) differentiation is carried out by these steps, backwards compared to the flow of the valuation steps (1)–(3):*

1. *Initialize with $\bar{x}_i = 0_D$ for $i = 1, \ldots, 5$ (0_D is a D-dimensional row vector of 0-s) and $\bar{x}_0$ a fixed D-dimensional row vector;*
2. *Update per (3),*
$$\bar{x}_1 = \bar{x}_1 + \bar{x}_0 \frac{\partial f_0}{\partial x_1}, \quad \bar{x}_2 = \bar{x}_2 + \bar{x}_0 \frac{\partial f_0}{\partial x_2}; \tag{4}$$
3. *Update per (2),*
$$\bar{x}_3 = \bar{x}_3 + \bar{x}_1 \frac{\partial f_1}{\partial x_3}, \quad \bar{x}_4 = \bar{x}_4 + \bar{x}_1 \frac{\partial f_1}{\partial x_4}; \tag{5}$$
4. *Update per (1),*
$$\bar{x}_4 = \bar{x}_4 + \bar{x}_2 \frac{\partial f_2}{\partial x_4}, \quad \bar{x}_5 = \bar{x}_5 + \bar{x}_2 \frac{\partial f_2}{\partial x_5}. \tag{6}$$

It is not hard to check by hand that after completing the algorithm we have

$$\bar{x}_i = \bar{x}_0 \frac{\partial f_0}{\partial x_i},\ i = 3, 4, 5,$$

and that, in our example, $\bar{x}_4$ picks up the contributions from both f_1 and f_2. In particular, by setting $\bar{x}_0 = 1_j$ to be the vector with 1 in j-th position and 0-s elsewhere, we recover the columns of $\frac{\partial f_0}{\partial x_i}$. Each of the gradient updates in Algorithm 1 is mechanically derived from the corresponding function evaluation in (1)–(3) which is, indeed, the promise of the first "A" (Automatic) in "AAD". It is also important to note that none of the gradient updates (4)–(6) require explicit calculations of matrices (Jacobians) but are always just updates of row vectors by matrix multiplication. While this may seem like a minor point, it is quite critical to the performance of the algorithm, especially in the "smart" application of the IFT. Another important point is that we have the flexibility to choose the initial adjoints $\bar{x}_0$ for the AAD calculation depending on what we want to achieve.

3 Naive IFT

Let

$$X = X(c, w),\ X : \mathbb{R}^N \times \mathbb{R}^M \to \mathbb{R}^N$$

be a function representing a "model". Here c stands for the coefficients of the model, and w stands for the auxiliary variables ("weights"). Let x denote the "market"variables, and

$$\Omega = \Omega(c, x, w),\ \Omega : \mathbb{R}^N \times \mathbb{R}^N \times \mathbb{R}^M \to \mathbb{R}^N$$

be a calibration objective function explicitly linked to X, often in the form

$$\Omega(c, x, w) = X(c, w) - x. \tag{7}$$

We define the "market-calibrated model coefficients" function

$$C = C(x, w),\ C : \mathbb{R}^N \times \mathbb{R}^M \to \mathbb{R}^N$$

as the solution to

$$\Omega(C(x, w), x, w) = 0 \tag{8}$$

in the first (c) argument. Once the model is calibrated, we often want to use the calibrated coefficients to calculate some other value of interest

$$U(x, w) = V(C(x, w), w) \tag{9}$$

using the function

$$V = V(c, w)\ V : \mathbb{R}^N \times \mathbb{R}^M \to \mathbb{R}.$$

For example, x may represent a vector of benchmark market-observed rates, c the coefficients of the interest rate curve parameterization that we fit to the benchmarks, w some interpolation parameters, and U a calculation of a non-benchmark rate using the calibrated curve.

Very often we are not only interested in U but also its derivatives $\frac{\partial U}{\partial x}$ with respect to the market variables x. By the chain rule from (9),

$$\frac{\partial U}{\partial x} = \frac{\partial V}{\partial c}\frac{\partial C}{\partial x} \tag{10}$$

(using the same notation as in Section 2, for instance $\frac{\partial V}{\partial c}$ stands for the N-dimensional vector of the derivatives with respect to the coefficients). The quantity $\frac{\partial V}{\partial c}$ is normally readily available via AAD. The $\frac{\partial C}{\partial x}$ term in (10), however, cannot be easily obtained via AAD directly as the calculation of C involves an invocation of a non-linear, often iterative solver (8). This is where the IFT comes in. Differentiating (8) in x, we get

$$\frac{\partial \Omega}{\partial c}\frac{\partial C}{\partial x} + \frac{\partial \Omega}{\partial x} = 0,$$

and so

$$\frac{\partial C}{\partial x} = -\left(\frac{\partial \Omega}{\partial c}\right)^{-1}\frac{\partial \Omega}{\partial x}. \tag{11}$$

(in the common special case (7) we simply have $\frac{\partial C}{\partial x} = \left(\frac{\partial X}{\partial c}\right)^{-1}$). Then we can rewrite (10) as

$$\frac{\partial U}{\partial x} = -\frac{\partial V}{\partial c}\left(\frac{\partial \Omega}{\partial c}\right)^{-1}\frac{\partial \Omega}{\partial x}. \tag{12}$$

We note that $\frac{\partial \Omega}{\partial c}$ is usually required anyway when solving (8) (say, by Newton's method). Hence, all the terms in (12) are readily available (the first from evaluating the pricing function V with AAD turned on, the second from the non-linear solver, the last from AAD applied to Ω), and the gradient $\frac{\partial U}{\partial x}$ is easy to calculate. In this simple and stylized example, a direct application of IFT is sufficient.

The situation becomes more complicated – but more realistic – when X, and thus Ω, has additional dependencies on x. To consider this case, we extend our simple example by making the weights w depend on x:

$$w = W(x).$$

At this point, it is important to emphasize the distinction we make between the direct dependence of Ω on x, and the additional dependencies of Ω on x via auxiliary parameters such as w. The former is typically localized in the function Ω, easily understood from the code, and can be handled by automated tools such as AAD easily. The latter may not be quite as straightforward as various calibration "helpers", "wrappers", "smoothers" and similar connections with other objects in the code make tracing dependencies through the codebase manual, tedious and error-prone. We will highlight the implications of this important practical consideration in a moment.

From a mathematical standpoint, adding extra dependency just adds more terms to our gradient calculations. The calibration problem (8) now becomes

$$\Omega(C(x, W(x)), x, W(x)) = 0, \tag{13}$$

and the calculation chain (9) looks like

$$U(x) = V(C(x, W(x)), W(x)). \tag{14}$$

The gradient we want is now given by

$$\frac{\partial U}{\partial x} = \frac{\partial V}{\partial c}\left(\frac{\partial C}{\partial x} + \frac{\partial C}{\partial w}\frac{\partial W}{\partial x}\right) + \frac{\partial V}{\partial w}\frac{\partial W}{\partial x}. \tag{15}$$

As before, we know $\frac{\partial C}{\partial x}$ from (11). However, now (15) also involves $\frac{\partial C}{\partial w}$. This quantity can also be obtained from (8) by the application of the IFT to the gradient with respect to w:

$$\frac{\partial \Omega}{\partial w}\frac{\partial C}{\partial w} + \frac{\partial \Omega}{\partial w} = 0,$$

giving us

$$\frac{\partial C}{\partial w} = -\left(\frac{\partial \Omega}{\partial c}\right)^{-1}\frac{\partial \Omega}{\partial w} \tag{16}$$

(compare to (11)). A direct application of IFT gives us the following algorithm.

Algorithm 2 (Naive IFT) *To calculate (15), we carry out the following steps:*

1. *Solve (13) to obtain c^*, the calibrated model coefficients;*
2. *Obtain $\frac{\partial V}{\partial c}$ by AAD as a by-product of calculating $V(c, w)$ at $c = c^*$;*
3. *Obtain $\frac{\partial \Omega}{\partial x}$ by AAD as a by-product of calculating $\Omega(c, x, w)$ at $c = c^*$. The parameters x are explicit in the definition of Ω and can be "AAD"-ed easily;*
4. *Extract $\frac{\partial \Omega}{\partial c}$ from the Newton solver used in calibration and then calculate $\frac{\partial C}{\partial x}$ via (11);*
5. *Obtain $\frac{\partial W}{\partial x}$ by AAD as a by-product of calculating $W(x)$. Note that this may happen at the point in code that is far away from the solver invocation;*
6. *Find all instances of auxiliary parameters such as weights w that may affect Ω and V. Once they are identified, calculate $\frac{\partial V}{\partial w}$ and $\frac{\partial \Omega}{\partial w}$ by directly invoking AAD at the relevant parts of the calculations;*
7. *Combine all gradients in (15) to obtain the desired $\frac{\partial U}{\partial x}$.*

Let us discuss step 6 of Algorithm 2 in more detail. The critical difference between calculating $\frac{\partial}{\partial w}$ and $\frac{\partial}{\partial x}$ gradients lies in the practical realm. As already mentioned, the dependencies on w are typically not clearly encapsulated in the valuation function V or the objective function Ω. Hence, automated tools such as AAD are difficult to integrate. Also note that $\frac{\partial \Omega}{\partial w}$ calculated in step 6 and required in (16) is a matrix of dimensions $N \times M$. Hence, the calculation needs either M direct or N adjoint gradient passes. In addition to being cumbersome to orchestrate, the algorithm is slowed down considerably as both N and M can be quite large.

4 Automatic IFT

Recall the basics of AAD as explained in Section 2. With those in mind, consider the calculation of the objective function $\Omega(c^*, x, w)$ where c^* is the set of calibrated model coefficients obtained by solving (8). Let $\bar{\Omega}$ (an analogue of $\bar{x}_0$ from Section 2) be a yet unspecified row vector of dimension N (the output dimension of Ω). By calculating $\Omega(c, x, w)$ with AAD enabled and setting the "input" adjoints $\bar{\Omega}$ we obtain the following "output" adjoints

$$\bar{x} = \bar{x} + \bar{\Omega}\frac{\partial \Omega}{\partial x}, \quad \bar{w} = \bar{w} + \bar{\Omega}\frac{\partial \Omega}{\partial w}, \tag{17}$$

where $\bar{x}$, $\bar{w}$ on the right-hand sides are the accumulated adjoints before Ω is invoked. By examining (15) we note that we need to have

$$\bar{x} = \hat{x}, \ \bar{w} = \hat{w},$$

where

$$\hat{x} \triangleq \frac{\partial V}{\partial c}\frac{\partial C}{\partial x}, \quad \hat{w} \triangleq \frac{\partial V}{\partial c}\frac{\partial C}{\partial w}. \tag{18}$$

By (11) and (16) we have

$$\hat{x} = \frac{\partial V}{\partial c}\left(\frac{\partial \Omega}{\partial c}\right)^{-1}\frac{\partial \Omega}{\partial x}, \quad \hat{w} = \frac{\partial V}{\partial c}\left(\frac{\partial \Omega}{\partial c}\right)^{-1}\frac{\partial \Omega}{\partial w}. \tag{19}$$

Comparing (17) and (19), we make our key observation: setting

$$\bar{\Omega} = \frac{\partial V}{\partial c}\left(\frac{\partial \Omega}{\partial c}\right)^{-1} \tag{20}$$

to be the input adjoint, and calculating the objective function *once* with the AAD enabled, we obtain $\bar{x}$ and $\bar{w}$ (per (17)) as the output, which are exactly the required gradients:

$$\bar{x} = \hat{x} = \frac{\partial V}{\partial c}\frac{\partial C}{\partial x}, \quad \bar{w} = \hat{w} = \frac{\partial V}{\partial c}\frac{\partial C}{\partial w}. \tag{21}$$

Recall that we identified the calculation of $\frac{\partial \Omega}{\partial w}$ as the main conceptual and performance bottleneck of the "naive" Algorithm 2. In the calculation flow we just described, this bottleneck has been eliminated. Not only do we not need to calculate the matrix $\frac{\partial \Omega}{\partial w}$ explicitly, we also do not need to trace all uses of all the auxiliary variables that may appear inside the objective function, so *we do not even need to identify all w explicitly*. By calculating Ω with AAD enabled with the right input adjoint $\bar{\Omega}$, all the contributions of all the variables will be added to the correct output adjoints.

Let us now summarize our observations into the "correct" application of the IFT in the AAD work-flow for the same example as in Algorithm 2.

Algorithm 3 (AIFT) *To calculate (15), we carry out the following steps:*

1. *Solve (13) to obtain c^*, the calibrated model coefficients;*
2. *Obtain $\frac{\partial V}{\partial c}$ by AAD as a by-product of calculating $V(c, w)$ at $c = c^*$;*
   (a) *At the same time $\bar{w} = \frac{\partial V}{\partial w}$ is calculated. Note that this will be stored "on tape" and later updated.*
3. *Extract $\frac{\partial \Omega}{\partial c}$ from the Newton solver used for calibration;*
4. *Calculate $\Omega(c^*, x, w)$ once with AAD enabled, feeding $\bar{\Omega}$ as defined in (20) as the input adjoint. The output is $\bar{x}$ and $\bar{w}$ as in (21), where the extra term $\frac{\partial V}{\partial c}\frac{\partial C}{\partial w}$ is **added** to $\bar{w}$ calculated in Step 2a by the mechanics of AAD;*
5. *Calculate $W(x)$ with AAD enabled and the input adjoint $\bar{w}$. This will update $\bar{x} = \bar{x} + \bar{w}\frac{\partial W}{\partial x}$;*
6. *The output, $\bar{x}$ is equal to $\frac{\partial U}{\partial x}$ in (15) as required.*

5 Automatic IFT for best-fit optimizers

Let us go back to the calibration problem (8). In practice, as the market variables x are subject to noise, practitioners prefer to enrich the set of fitting equations with some number (say P) of regularization conditions. Suppose that the objective function Ω now has $N+P$ outputs:

$$\Omega = \Omega(c, x, w),\ \Omega : \mathbb{R}^N \times \mathbb{R}^N \times \mathbb{R}^M \to \mathbb{R}^{N+P},$$

while the number of model coefficients c stays the same. An exact fit in this situation is clearly not possible in general, so C is then defined as the following argmin value:

$$C(x, w) = \operatorname{argmin}_c L(c, x, w),\ \text{where } L(c, x, w) \triangleq \|\Omega(c, x, w)\|^2. \tag{22}$$

Let us now explain how AIFT can be extended to this, more complex, case.

5.1 Generic solution for best-fit problems

A generic extension of AIFT to the best-fit calibration problem is obtained by replacing the optimization problem (22) with the first-order optimality conditions. (We note that smoothness requirements for this substitution are nearly always satisfied in practical applications.) Specifically, the solution $C = C(x, w)$ to (22) satisfies

$$(\nabla L)(C(x, w), x, w) = 0, \tag{23}$$

where ∇L stands for the gradient of L. In fact (22) and (23) are equivalent in a small neighbourhood of the solution to (22). Noting that (23) and (8) are exactly the same save for the form of the objective function, we can reduce the problem of applying IFT automatically to (22) to the problem we have already solved by using Algorithm 3 (AIFT) from Section 4 for the equation $\nabla L = 0$.

This approach to best-fit optimizers is generic and can be easily automated in an AAD library, but it has certain drawbacks. To apply Algorithm 3 to ∇L we need, well, ∇L itself, which is not a priori supplied by the user. More importantly, the algorithm needs the adjoints, i.e. partial derivatives, of ∇L, $\frac{\partial \nabla L}{\partial c}$ and the like, which essentially requires calculating the matrix of *second*-order derivatives (the Hessian) of L (and, hence, of the original objective function Ω). We can of course calculate all these by AAD, see a discussion in e.g. [7, (16),(17)] and related general instructions for the solver case in [3, Algorithm 3.1]. However, the cost of calculating second-order derivatives, by AAD or manually, is certainly higher than that of calculating the first-order derivatives needed in the exact-fit case.

5.2 Approximate solution for nearly-exact-fit problems

In practice, the objective function Ω is often set up so that L is a sum of a fitting part and a regularization part, the latter modulated by a small parameter. This is the case in a common set-up of representing a calibration problem as a non-linear least-squares optimization. For example one might specify

$$\Omega = \Omega(c, x, w) = ((\Omega_{\text{fit}})_1, \dots, (\Omega_{\text{fit}})_N, \varepsilon c_1, \dots, \varepsilon c_N)^\top,$$

where

$$\Omega_{\text{fit}} = \Omega_{\text{fit}}(c, x, w),\ \Omega_{\text{fit}} : \mathbb{R}^N \times \mathbb{R}^N \times \mathbb{R}^M \to \mathbb{R}^N$$

is the exact-fit part, $\varepsilon > 0$ is small, $P = N$, and the regularization part penalizes large values of c_n, $n = 1, \dots, N$:

$$L(x, c, w) = \|\Omega_{\text{fit}}(c, x, w)\|^2 + \varepsilon^2 \|c\|^2.$$

If the regularization parameter is sufficiently small, it is natural to assume that the original IFT idea should still work, provided we replace the inverse $(\frac{\partial \Omega}{\partial c})^{-1}$ in (20) by the pseudo-inverse

$$\left(\left(\frac{\partial \Omega}{\partial c} \right)^\top \frac{\partial \Omega}{\partial c} \right)^{-1} \left(\frac{\partial \Omega}{\partial c} \right)^\top.$$

This has been suggested in e.g. [12, p.479] and [9] in the context of (naive) IFT. Let us demonstrate that this is indeed a valid approach by deriving a bound for the approximation error, we believe this is novel. Clearly, replacing the inverse with the pseudo-inverse for Ω in the AIFT algorithm is trivial as we just back-propagate the objective function Ω once, and it is much more computationally efficient than the generic solution that applies the AIFT algorithm to ∇L to obtain non-approximated adjoints.

Let us state the approximation error bound that arises when using the pseudo-inverse with the objective function Ω versus the actual inverse with the objective function ∇L. Recall that x and c are N-dimensional vectors, and w is an M-dimensional vector. We denote by x' one of the components $x_1, \dots, x_N$ of x. By $\hat{x}'_{\text{exact}}$ we denote the corresponding component of $\hat{x}$ (as defined in (18)) from the IFT solution for the objective ∇L, and by $\hat{x}'_{\text{appr}}$ the same but calculated with the pseudo-inverse in the IFT for the objective function Ω. The same notations are applied to w. We have the following bound for the difference $|\hat{x}'_{\text{appr}} - \hat{x}'_{\text{exact}}|$.

Proposition 4 *For the exact and approximate adjoints*

$$\hat{x}'_{\text{exact}} = \frac{\partial V}{\partial c}\left(\frac{\partial \nabla L}{\partial c}\right)^{-1}\frac{\partial \nabla L}{\partial x'}, \tag{24}$$

$$\hat{x}'_{\text{appr}} = \frac{\partial V}{\partial c}\left(\left(\frac{\partial \Omega}{\partial c}\right)^{\top}\frac{\partial \Omega}{\partial c}\right)^{-1}\left(\frac{\partial \Omega}{\partial c}\right)^{\top}\frac{\partial \Omega}{\partial x'}, \tag{25}$$

the following error estimate holds:

$$|\hat{x}'_{\text{appr}} - \hat{x}'_{\text{exact}}| \le L^* \left\|\frac{\partial V}{\partial c}\right\| \left(R_1 F^*_{x'} J^* + \frac{R_2}{s^2_{\min}} + R_1 R_2\right), \tag{26}$$

where

$$L^* = L(C(x,w),x,w),$$

$$R_1 = \frac{H^*}{s^2_{\min} - L^* H^*} \ \textit{(assuming } L^* H^* < s^2_{\min}\textit{)}, \quad R_2 = H^*_{x'},$$

$$s_{\min} = \ \textit{the smallest singular value of } \frac{\partial \Omega}{\partial c},$$

$$H^* = \max_{1\le i,j\le N}\left\|\frac{\partial^2 \Omega}{\partial c_i \partial c_j}\right\|, \quad H^*_{x'} = \max_{1\le i\le N}\left\|\frac{\partial^2 \Omega}{\partial c_i \partial x'}\right\|, \quad F^*_{x'} = \left\|\frac{\partial \Omega}{\partial x'}\right\|, \quad J^* = \max_{1\le i\le N}\left\|\frac{\partial \Omega}{\partial c_i}\right\|.$$

The same estimate holds for the absolute error for $\hat{w}'$ by replacing x' with w' in the formulas above.

We prove this proposition in [5].

In the next section we discuss an example of implementing AIFT for the exact-fit calibration problem of Sections 3, 4. A Jupyter notebook that implements the pseudo-inverse approximate solution to the best-fit calibration problem from this section is also available at https://github.com/piterbarg/aift/.

6 An implementation example, Python

To fully appreciate the differences in implementation complexity and the level of automation that can be achieved in coding IFT by hand vs. the AIFT, it is instructive to consider an example. In this section, we give a short Python example that demonstrates both approaches in practice. A Jupyter notebook is available at https://github.com/piterbarg/aift/.

The overall plan is as follows. We start by presenting a stylized calibration problem of fitting a parametric curve to some knot points $\{t_i, x_i\}$ and evaluating it at some other input t to obtain the final value. Our goal is to enable the calculation of the derivatives of this final value with respect to the inputs by applying AAD to the valuation/calibration code. Then we show how one could use IFT manually, highlighting all the extra code that a human (as opposed to a machine) would need to write. Finally, we demonstrate how the same goal can be achieved by AIFT in a fully automated way.

For didactic purposes, we explicitly write most of the code that would be written by an AAD library with AIFT enabled. At the same time, we short-cut some of the less interesting calculations by using limited AAD functionality provided by the `autograd`[1] package.

Let us start by defining a (Python) function `spolyval` below, a concrete embodiment of the "model" function $X(c, w)$ introduced in Section 3, implementing a ("stretched") polynomial with coefficients `coefs`. Weights `w = [w[0],w[1]]` essentially determine the interpolation between knot times `ts`. The way in which the weights affect the output in this example is somewhat contrived as we prioritize simplicity of exposition over realism.

```
from scipy.optimize import least_squares
from autograd import grad, jacobian
import autograd.numpy as np

def spolyval(coefs, ts, w):
    '''
    A 'stretched polynomial' function, a polynomial in wts,
    where wts = w[0]*ts + w[1]*ts**2.
    Weights w here control the shape of the function between knots.

    coefs:  polynomial coefs
    ts:     points where the function is evaluated
    w:      weights to transform ts into wts
    '''
    tsw = w[0]*ts + w[1]*ts**2
    val = 0.0
    for n in range(len(coefs)):
        val =  val + coefs[n] * tsw**n
    return val
```

Weights such as `w` are often wrapped in layers of code ("helpers", "wrappers", etc.), and we simulate this programming pattern by creating the `PricingHelper` class. This class also serves to introduce the dependence $w = W(x)$ under certain conditions. We note that the condition that triggers the dependence is set in the class constructor, which can be called "far away" from the calibration code, once again underscoring our point that tracing auxiliary dependencies like this one by hand is hard and error-prone.

```
class PricingHelper:
    def __init__(self, w):
        self.w_ = w
        self.updatable = False
        # If w is none we link w's to xs's in a particular way
        # to introduce the extra dependence of the result of spoly_interp
        # on xs via w (admittedly, somewhat artificially). The actual update
        # happens in the update(...) function that the clients are supposed
        # to call when the xs are known.
        if w is None:
            self.updatable = True

    def update(self, xs, ts):
        '''
        Update the weights depending on the inputs ts (not used
        in this example) and xs.
        '''
        if self.updatable:
```

[1] https://pypi.org/project/autograd/

```
38                 self.w_ = np.array([1.0, np.sum(xs**2)])
39
40         def spolyval(self,c,ts):
41             return spolyval(c, ts, self.w_)
```

The function `spoly_interp(...)` corresponds to the composition of the valuation function $V = V(c, w)$ with the implicit function $C(x, w)$. The inner Python function `obj_f(...)` at line 50 implements the objective function $\Omega(c, x, w)$, which is calibrated by the `least_squares` function from `scipy` at line 54.

```
42     def spoly_interp(xs, ts, t, pricing_helper):
43         '''
44         Fit a stretched polynomial to (ts,xs) and evaluate it at t
45         Here pricing_helper (via pricing_helper.w_) is defining
46         the interpolation between the knots.
47         '''
48         pricing_helper.update(xs,ts)
49
50         def obj_f(c, x, pricing_helper = pricing_helper):
51             return pricing_helper.spolyval(c, ts) - x
52
53         x0 = np.zeros_like(ts)
54         res = least_squares(lambda c : obj_f(c, xs), x0)
55         c_fit = res.x
56         return pricing_helper.spolyval(c_fit, t)
```

Here is an example of how this function is called.

```
57     # points we interpolate
58     ts = np.array([0.,1,2,3,4])
59     xs = np.array([2.,1,3,4,0])
60
61     # the point at which we evaluate our interpolator
62     t = 3.5
63
64     # We can try different values of w. 'None' is the default that triggers
65     # the calculation w = w(x)
66     w_to_use = None
67     # w_to_use = [1.0,30.0]
68
69     # Set up the pricer helper
70     pricing_helper = PricingHelper(w_to_use)
71
72     # calculate the interpolated value
73     v = spoly_interp(xs,ts,t, pricing_helper)
74     print(v)
```

With the base code defined, let us now proceed to enhance it with gradient calculations. We want (a version of) `spoly_interp(...)` to not only return the value of the calibrated polynomial at `t` but also its derivatives with respect to `xs`.

We add the calculations of $\frac{\partial W(x)}{\partial x}$ to `PricingHelper` by sub-classing it. While here we do it by hand for brevity, `autograd` or some other AAD library can do it automatically.

```
75     class PricingHelperIft(PricingHelper):
76         def __init__(self, w):
77             super().__init__(w)
78
```

```
79          def update(self, xs, ts):
80              super().update(xs,ts)
81
82              # Capture the gradients if w is in fact a function of x. We could call
83              # autograd here but choose to code this by hand for brevity.
84              if self.updatable:
85                  self.dw_dx_ = np.vstack((np.zeros_like(xs), 2*xs))
86              else:
87                  self.dw_dx_ = np.zeros((2,len(xs)))
```

Calculations in `spoly_interp(...)` involve a non-linear solver, and so we need to use IFT. Let us first apply it naively, as described in Section 3. It is worth pointing out once again that the signature of `spoly_interp(...)` (line 42) does not involve `w`, yet we know that $w = W(x)$ introduces extra dependency of the output on x that needs to be captured in the gradient calculations. This hints at the difficulties that are encountered in practice with this approach, as the code below needs to be aware of the variable w to calculate `dobj_dw` ($= \frac{\partial \Omega}{\partial w}$), see lines 122–123.

Lines 120–126 explicitly demonstrate the crux of the problem with the naive application of IFT. These lines have to be written by a human with domain-specific knowledge of the problem being solved that has traced all the auxiliary variables that might affect calculations of the gradients at this point. Moreover, similar lines of code need to be written for all variables affecting the gradients.

We highlight the most problematic lines with a red box in the listing below.

```
88      def spoly_interp_ift(xs, ts, t, pricing_helper):
89          '''
90          This is a modification of spoly_interp() that supports gradients via
91          Naive IFT. We use autograd and need to use manual gradient manipulations
92          to collect them all.
93
94          The original function spoly_interp(...) fits a stretched polynomial to
95          (ts,xs) and evaluates it at t. Here pricing_helper (via pricing_helper.w_)
96          is defining the interpolation between knots.
97          '''
98
99          # Update the weights w and extract the relevant gradients
100         pricing_helper.update(xs,ts)
101         dw_dx = pricing_helper.dw_dx_
102
103         # The original objective function
104         def obj_f(c, x, pricing_helper = pricing_helper):
105             return pricing_helper.spolyval(c, ts) - x
106         # We need an unwrapped version of the objective function for autograd
107         # to be able to calculate dobj_dw below.
108         def obj_f_wrapper(c, x, w):
109             helper_ = PricingHelper(w)
110             return helper_.spolyval(c, ts) - x
111
112         x0 = np.zeros_like(ts)
113         res = least_squares(lambda c: obj_f(c,xs), x0)
114
115         c_fit = res.x
116         v = pricing_helper.spolyval(c_fit, t)
117         # calc the gradients using IFT
118         dobj_dc = jacobian(obj_f, argnum = 0)(c_fit,xs)
119         dobj_dx = jacobian(obj_f, argnum = 1)(c_fit,xs)
120         dc_dx = -np.linalg.lstsq(dobj_dc,dobj_dx, rcond = None)[0]
```

```
        # Calculate the gradient with respect to w. We need to keep adding
        # these for all "hidden" variables that are used in obj_f
        w = np.array(pricing_helper.w_.copy()) # a bit of a hoop here for autograd
        dobj_dw = jacobian(obj_f_wrapper, argnum = 2)(c_fit,xs,w)

        dc_dw = -np.linalg.lstsq(dobj_dc,dobj_dw, rcond = None)[0]
        dc_dx += (dc_dw @ dw_dx)

        dv_dc = grad(spolyval, argnum = 0)(c_fit, t, w)
        dv_dx = dv_dc @ dc_dx

        # need to add the dw_dx contribution to the final valuation as well
        dv_dw = grad(spolyval, argnum = 2)(c_fit, t, w)
        dv_dx += dv_dw @ dw_dx

        return v, dv_dx
```

Let us now demonstrate the AIFT approach as described in Section 4. For educational purposes, we avoid using any particular AAD library "behind the scenes" (nor, to be fair, are we aware of any Python library that implements IFT in the way we recommend). We introduce (a very rudimentary version of) the AAD "tape", the data structure that saves the partial derivatives and adjoints.

```
    def init_state_adj(ncoefs):
        '''
        Initialize state_adj. This will be done by the AAD library.
        '''
        state_adj = {
            'sp_bar': np.array([1]),
            'c_bar' : np.zeros(ncoefs),
            'x_bar': np.zeros(ncoefs),
            'w_bar' : np.zeros(2),
            'f_bar': np.zeros(ncoefs),
        }

        return state_adj
```

Additionally we implement adjoint versions of `class PricingHelperAdj(PricingHelperIft)` and `obj_f_adj(...)`. In a production AAD library, these are generated automatically. To avoid clutter we do not reproduce the code here, but it can be found in the source notebook at https://github.com/piterbarg/aift/.

The AIFT-enabled version of `spoly_interp(...)` is listed below. We recommend reading the code in parallel with Algorithm 3. The key step of "seeding" $\bar{\Omega}$ with the correct value is highlighted with a green box.

```
    def spoly_interp_aad(xs, ts, t, pricing_helper):
        # Step 0. Initialize the state_adj
        state_adj = init_state_adj(len(ts))

        # Step 1. Update the weights w and extract the relevant gradients
        pricing_helper.update(xs,ts)
        dw_dx = pricing_helper.dw_dx_

        # The original objective function
        def obj_f(c, x, pricing_helper = pricing_helper):
            return pricing_helper.spolyval(c, ts) - x

```

```
        # Step 2. Fit the objective function and extract the coefs c we fit
        x0 = np.zeros_like(ts)
        res = least_squares(lambda c: obj_f(c,xs), x0)
        c_fit = res.x

        # Step 3. Calculate the value of spolyfit using the fitted coefs c_fit
        v = pricing_helper.spolyval(c_fit, t)

        # Gradient to coefs, known in the Newton method so no extra calcs here
        dobj_dc = jacobian(obj_f, argnum = 0)(c_fit, xs)

        # Adjoint for Step 3. I.e. propagate backwards until the call to the solver
        pricing_helper.spolyval_adj(c_fit, t, state_adj)
        c_bar = state_adj['c_bar']

        # Compute the correct adjoints of the objective function:
        obj_f_bar = -np.linalg.lstsq(dobj_dc.T, c_bar, rcond = None)[0]
        state_adj['f_bar'] = obj_f_bar

        # Adjoint for Step 2. Propagate through the objective function. Note that
        # we do not have to compute dobj_dw unlike the Naive IFT approach
        obj_f_adj(c_fit, ts, xs, pricing_helper, state_adj)
        x_bar = state_adj['x_bar']
        w_bar = state_adj['w_bar']

        # Adjoint for Step 1. Propagate through w=w(x)
        x_bar += w_bar @ dw_dx

        return v, x_bar
```

It should be clear from this example that AIFT-enabled AAD calculations can indeed be fully orchestrated automatically by a machine without manual intervention. In particular, all the adjoint steps in `spoly_interp_aad(...)` are derived from the corresponding valuation steps, and there is no need to call $\frac{\partial \Omega}{\partial w}$ calculations explicitly.

7 An implementation example, C++

To further demonstrate the power of AIFT in a production-like setting, we prepared a C++ example based on an open-source derivatives pricing library `QuantLib`. It can be explored via the link provided in [10] using a modern web browser. In this project, the following steps are implemented:

- A multi-curve best-fit calibration with `Eigen::LevenbergMarquardt`;
- Portfolio pricing using `QuantLib` APIs;
- Back-propagation of the first two steps according to the methodology of Section 5 i.e. using AIFT for best-fit optimizers with `Matlogica AADC` as the AAD library;
- Live-risk demo for a portfolio of 1000 interest rate swaps computing NPV and full risk to market rates in less than 0.5 seconds;
- Verification of correctness using the "bump and recalibrate" approach.

We do not include the source code in this paper as it is much easier to explore it using the links mentioned above.

8 Conclusions and Acknowledgements

In this paper, we introduced and demonstrated the power of the Automatic IFT approach which appears to be little-known in the broad quantitative finance community. AIFT is a version of the Implicit Function Theorem that eliminates the need for manual intervention when introducing AAD into financial code that solves calibration problems. The method is exact for non-linear multi-dimensional function inversion problems. We extended it as an approximate solution to a broader class of nearly-exact calibration problems demonstrating that our method covers a wide range of practical approaches to calibration, and derived error estimates for the approximations. Finally, we gave a fully generic and exact, albeit somewhat costlier, solution for any best-fit calibration set-up.

Detailed worked examples, in annotated Python code, are available at https://github.com/piterbarg/aift/, and production-quality C++ examples based on `QuantLib` APIs can be found at [10].

We are grateful to Luca Capriotti, Antoine Savine and Evgeny Goncharov for insightful comments and feedback.

References

[1] L. Capriotti, Y. Jiang, A. Macrina, *Real-time risk management: An AAD-PDE approach*, International Journal of Financial Engineering, Vol. 02, No. 04, 1550039 (2015).

[2] L. Capriotti, S. Lee, *Adjoint Credit Risk Management* (2013), https://ssrn.com/abstract=2342573

[3] B. Christianson, *Reverse accumulation and implicit functions*, optimization Methods and Software, Volume 9, Issue 4 (1998), pp. 307–322

[4] M. Giles, P. Glasserman, *Smoking Adjoints: Fast Evaluation of Monte Carlo Greeks*, 2006, Risk.net

[5] D. Goloubentsev, E. Lakshtanov, V. Piterbarg, *Automatic Implicit Function Theorem* (December 14, 2021). Available at SSRN: https://ssrn.com/abstract=3984964

[6] J. Lotz, M. Schwalbach, U. Naumann, *A case study in adjoint sensitivity analysis of parameter calibration*, Procedia Computer Science, 80 (2016), pp. 201–211.

[7] U. Naumann, J. Lotz, K. Leppkes, M. Towara, *Algorithmic differentiation of numerical methods: Tangent and adjoint solvers for parameterized systems of nonlinear equations*, ACM Transactions on Mathematical Software (TOMS), 41(4) (2015), pp. 1–21.

[8] M. Henrard, *Calibration in finance: Very fast Greeks through algorithmic differentiation and implicit function*, Procedia Computer Science, 18 (2013), pp. 1145–1154.

[9] M. Henrard, *Algorithmic differentiation in finance: Root finding and least square calibration*, OpenGamma Quantitative Research, (7) (2013).

[10] MatLogica, *Multi-curve interest rate fitting using AADC and* ***QuantLib****. Portfolio live-risk*, https://matlogica.com/practical-AAD-minimizer-demo.php

[11] A. Savine, *From model to market risks: The implicit function theorem (IFT) demystified* (2018), https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3262571

[12] A. Savine, *Modern computational finance: AAD and parallel simulations* (2018).