Algorithmic Adjoint Differentiation (AAD) for Swap Pricing and DV01 Risk

Saïd Business School, University of Oxford, UK

Nicholas Burgess

nburgessx@gmail.com

May 2022

Abstract

We present Algorithmic Adjoint Differentiation (AAD), also known as Automatic Adjoint Differentiation, which computes the derivative(s) of computer code. In finance this leads to a relatively new and novel approach, pioneered by (Giles and Glasserman, 2006), to compute financial risks. When trading in electronic rates markets, AD can be used to produce exact swap risks, such as DV01, at high speed and give users a competitive advantage (Burgess 2021a).

AD is well presented in finance, see (Capriotti, 2010), (NAG, n.d.) and (Savine, 2018). However, in what follows we present algorithmic differentiation from an implementation perspective with a focus on how to apply AAD to interest rate swaps (IRS) for interest rate risk (DV01) calculations. Firstly, we give a brief overview of interest rate swaps and how to price them. Secondly we discuss swap risks, PV01 and DV01 risk calculations and the different approaches on how to compute them, namely via analytical closed-form solutions, numerical bumping and curve Jacobians. Thirdly we introduce algorithmic differentiation (AD) and show how to implement AD in both tangent and adjoint modes on a simple function. Fourthly we show how to compute DV01 risk using AD for an interest rate swap. Fifthly we discuss how to professionally implement AD using both open-source and commercial software solutions.

Throughout this implementation paper we present AD case studies in C++ code, for a simple function and for swaps. The source code has also been pre-loaded onto a free online compiler and can be run with a single button click, see the code footnotes for details. Furthermore, the raw source code is available at https://github.com/nburgessx/Papers/tree/main/SwapAD.

Key words: Interest Rate Swaps, Swap Pricing, Swap Risk, DV01, Algorithmic Differentiation (AD), Tangent Mode, Adjoint Mode, C++, AD By Hand, Professional AD Implementation

1. Interest Rate Swaps

An interest rate swap (IRS) is a financial product whereby one party exchanges a series of fixed payments (the fixed leg) for a series of floating payments (the floating leg), as illustrated in (figure 1) and (figure 2) below. A swap can be considered a mechanism to exchange a fixed rate loan for a variable or floating rate loan, where the floating rate of interest is linked to a floating rate such as USD LIBOR[1] or a risk-free rate (RFR) such as USD SOFR.

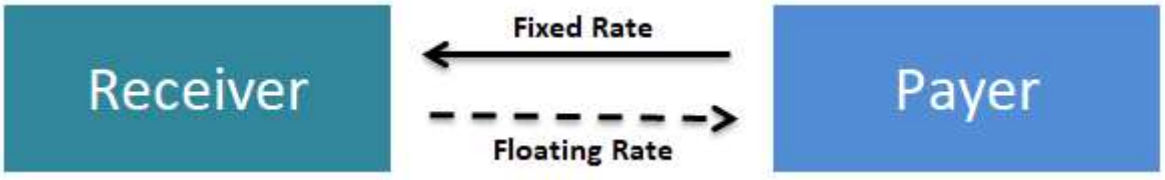

Figure 1: IRS Exchange of Fixed for Floating Rates

The party receiving the fixed coupons would describe the swap as a receiver swap and likewise if paying the fixed payments a payer swap. Interest rate swaps are quoted in the market place as a par rate i.e. the fixed interest rate that makes the swap worth zero or par as at the start or effective date of the swap. The par rate can be considered an average[2] of the floating LIBOR / RFR floating interest rates.

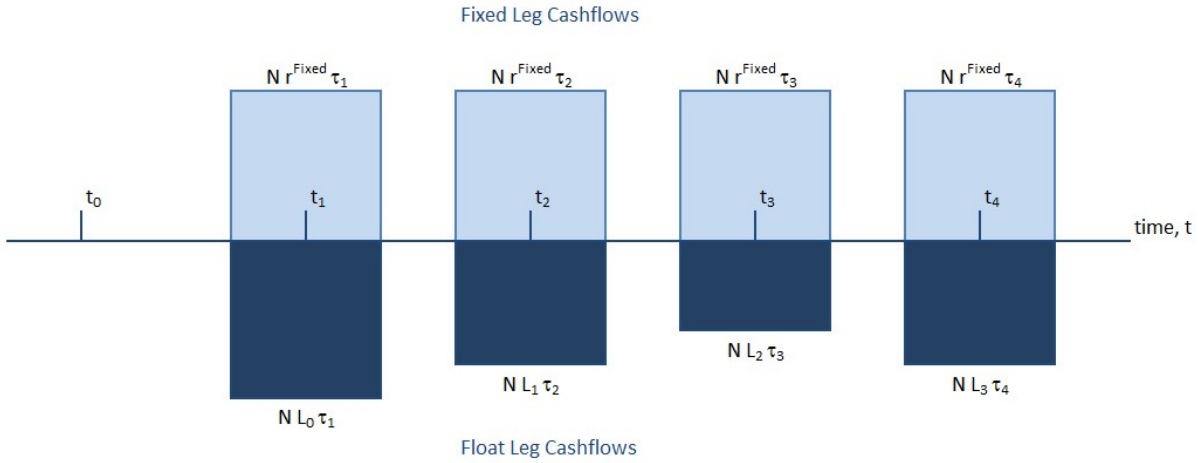

Figure 2: Interest rate swap cash flow illustration

[1] London Interbank Offer Rate. This is the rate at which a panel of London banks lend to each other. This rate is being replaced by risk-free rates (RFRs), which are based on market transactions.

[2] A weighted average by notional and discount factor.

1.1 Swap Pricing

The price or present value (PV) of an interest rate swap can be calculated using the formula below, see (Burgess 2016) for details and an Excel pricing workbook.

$$Swap\ PV = \varphi(PV(Fixed\ Leg) - PV(Float\ Leg)) \tag{1a}$$

Which expands as follows,

$$Swap\ PV = \varphi\left(\underbrace{\sum_{i=1}^{n} N\ r\ \tau_i\ P(0,t_i)}_{Fixed\ Leg} - \underbrace{\sum_{j=1}^{m} N\left(l_{j-1}+s\right)\tau_j\ P(0,t_j)}_{Floating\ Leg}\right) \tag{1b}$$

Where N is the notional, r the fixed rate, τ the coupon year fraction, $P(0,t_i)$ the ith coupon discount factor and φ is a pay or receive indicator function with $\varphi = 1$ if receiving and $\varphi = -1$ if paying the fixed leg.

1.2 Swap Risk

After pricing and executing a swap investors and traders compute swap risk to measure and / or hedge their interest rate risk. We give an outline of risk calculations in (Burgess 2016) and (Burgess 2022).

The DV01 calculation is of particular interest to market participants. It measures the sensitivity of the swap to a 1 basis point move[3] in interest rates and captures forward rate and discount factor risk impact. PV01 is similar, however only looks at the impact on forward rates and the forward rate risk as shown in (figure 3).

There are several ways to compute risk, as outlined below,

- **Analytical Risk**

  This involves deriving a closed-form solution by hand, which can be complex and is not always possible. For an analytical solution to Swap DV01, see (Burgess, 2016).

[3] Typically, an up-shift in interest rates, although some investors prefer to measure a down-shift.

Figure 3: USD Swap Price and Risk (Source: Bloomberg, SWPM <GO>)
Bloomberg are showing risk sensitivities for an interest rate down-shift of 1 basis point (bps)

- **Numerical Risk (Bumping)**

  Here we bump yield curve calibration instruments and reprice the swap. The change in price gives the risk. For total DV01 we bump all instruments at the same time for a 1 bps parallel shift. However, to generate bucketed DV01 we must perturb each instrument and would need to bump and reprice the swap(s) each time. This method is slow and computationally expensive.

- **Risk using Curve Jacobians**

  When yield curves have been calibrated using a gradient decent solver such as Newton-Raphson. The solver slope or '**Jacobian**' can be kept, stored and used to give the change in swap price for a change in forwards and / or discount factors, see (Burgess 2021b) for details.

- **Algorithmic Differentiation**

  This method can be used to compute the risk automatically at the same time as computing the price. This method produces exact derivatives that can be used to compute exceptionally fast, and accurate risks, however requires knowledge of algorithmic differentiation, can be highly complex and error prone. We outline this method in detail next.

2. Algorithmic Differentiation Explained

Algorithmic Differentiation (AD), also known as automatic differentiation, is a mathematical, computer science technique for computing accurate derivatives quickly. It was pioneered by (Giles and Glasserman, 2006) in finance can be used to compute exact derivatives with low latency. There are two main modes, namely **tangent mode** and **adjoint mode**. For many models, adjoint AD (AAD) can be used to compute sensitivities 10s, 100s or even 1000s of times faster than finite differences (NAG, n.d.). AD operates directly on analytics and each line of code is differentiated. AD can be applied to a single trade or a portfolio (or vector) of trades.

If for example we have a computer algorithm that computes y via multiple nested operations such as $y = h(g(f(x)))$ we could illustrate the series of operations as follows,

$$x \to f(x) \to g(\,f(x)\,) \to h(\,g(\,f(x)\,)\,) \to y \tag{2}$$

In short-hand we could write (2) as,

$$x \to f(x) \to g(f) \to h(g) \to y \tag{3}$$

Working **forwards** from the input value x, we can compute the derivative of each operation to compute the total derivative dy/dx as follows.

$$\frac{\cancel{df}}{dx}.\frac{\cancel{dg}}{\cancel{df}}.\frac{\cancel{dh}}{\cancel{dg}}.\frac{dy}{\cancel{dh}} = \frac{\boldsymbol{dy}}{\boldsymbol{dx}} \tag{4}$$

However we could also work **backwards** from the output value y to arrive at the same result,

$$\frac{dy}{\cancel{dh}}.\frac{\cancel{dh}}{\cancel{dg}}.\frac{\cancel{dg}}{\cancel{df}}.\frac{\cancel{df}}{dx} = \frac{\boldsymbol{dy}}{\boldsymbol{dx}} \tag{5}$$

Furthermore AD can be used on systems of equations and matrices. Tangent mode works forward from the left and performs **matrix-matrix multiplication** followed by a final matrix-vector product,

$$\left(\left(\left(\frac{\partial x_1}{\partial y}\frac{\partial x_2}{\partial x_1}\right)\frac{\partial x_3}{\partial x_2}\right)\dots\frac{\partial x_m}{\partial x_{m-1}}\right)\frac{\partial y}{\partial x_m} \tag{6}$$

With adjoint mode we work backwards from the right, however now everything is matrix-vector products, which is much faster.

$$\frac{\partial x_1}{\partial x}\left(\frac{\partial x_2}{\partial x_1}\left(\frac{\partial x_3}{\partial x_2}\dots\left(\frac{\partial x_m}{\partial x_{m-1}}\frac{\partial y}{\partial x_m}\right)\right)\right) \tag{7}$$

2.1 Tangent Mode (AD)

In tangent mode we differentiate code working forwards starting with the trade inputs and follow the natural order of the original program. This method computes price sensitivities to one input at a time and we must call the tangent method several times, once for each input parameter. Consequently, tangent forward mode has roughly the same cost as finite differences and numerical bumping, but computes gradients exactly to machine precision.

Dot Notation:
When using tangent mode **'dot'** notation is used to denote derivatives being differentiated with respect to the function input. For example given y = f(x) then y dot would indicate $\dot{\boldsymbol{y}} = \boldsymbol{dy/dx}$.

Consider the below simple function,

$$\text{Function:} \quad y = 2x^2 \tag{8}$$

$$\text{Tangent:} \quad \dot{y} = 4x.\dot{x} \tag{9}$$

The tangent dot notation in (equation 9) is equivalent to,

$$\text{Tangent:} \quad \frac{dy}{dx} = 4x.\frac{\boldsymbol{dx}}{\boldsymbol{dx}} \tag{10}$$

So what is the point of specifying $\dot{\boldsymbol{x}}$ which denotes dx/dx? Isn't this value just one? The answer is subtle, when programming in tangent mode $\dot{\boldsymbol{x}}$ is specified as an input and used to enable/disable the tangent derivative calculation. In (equation 9) above setting $\dot{\boldsymbol{x}} = \mathbf{1}$ enables the derivative calculation giving dy/dx = 4x, however when $\dot{\boldsymbol{x}} = \mathbf{0}$ we have dy/dx = 0.

Next let us consider how to apply tangent AD code to a simple function comprising of a series of simple incremental operations.

Function: $$f(x_1, x_2) = 2x_1^2 + 3x_2 \quad (11)$$

Solution: $$\frac{df}{dx_1} = 4x_1 \text{ and } \frac{df}{dx_2} = 3 \quad (12)$$

When $x_1 = 2$ and $x_2 = 3$ we have,

$$\frac{df}{dx_1} = 8 \text{ and } \frac{df}{dx_2} = 3 \quad (13)$$

Let's write this function in C++ code[4] and implement (equation 11) as a series of operations spanning multiple lines of C++ code.

```
01    double function( double x1, double x2 )
02    {
03        double a = x1*x1;          // Step 1:    a = x1^2
04        double b = 2*a;            // Step 2:    b = 2x1^2
05        double c = x2;             // Step 3:    c = x2
06        double d = 3*c;            // Step 4:    d = 3x2
07        double f = b + d;          // Step 5:    f = 2x1^2 + 3x2
08        return f;
09    }
```

Code 1: **Simple Function: $f(x_1, x_x) = 2x_1^2 + 3x_2$**

Source code: AD-Simple-Function.cpp
Available at: https://github.com/nburgessx/Papers/blob/main/SwapAD
To run see: https://onlinegdb.com/kKqaS6hJT

Now let's add tangent AD code to this function, working forwards using dot notation.

```
01    double tangent( double x1, double x2, double x1_dot, double x2_dot )
02    {
03        double a = x1*x1;              // Step 1:     a = x1^2
04        double a_dot = 2*x1*x1_dot;    // Tangent:    a' = 2x1 . x1'      a' = 2x1
05        double b = 2*a;                // Step 2:     b = a
06        double b_dot = 2*a_dot;        // Tangent:    b' = 2 . a'         b' = 4x1
07        double c = x2;                 // Step 3:     c = x2
08        double c_dot = x2_dot;         // Tangent:    c' = x2'            c' = 1
09        double d = 3*c;                // Step 4:     d = 3c
10        double d_dot = 3*c_dot;        // Tangent:    d' = 3 . c'         d' = 3
11        double f = b + d;              // Step 5:     f = 2x1^2 + 3x2
12        double f_dot = b_dot + d_dot;  // Tangent:    f' = b' + d'
13        return f_dot;                  // Result:     f' = 4x1 + 3
14    }
```

Code 2: **Simple Function $f(x_1, x_x) = 2x_1^2 + 3x_2$ with Tangent Derivative**

[4] To run these examples in C++ perhaps use a free online web compiler such as https://www.onlinegdb.com/.

Source code: AD-Simple-Function.cpp
Available at: https://github.com/nburgessx/Papers/blob/main/SwapAD
To run see: https://onlinegdb.com/kKqaS6hJT

Note the function f(x1, x2) takes two inputs, but we only have one f_dot derivative output on (line 13). This means in tangent mode we can only get one derivative output at a time. So to get the derivative to each input we would have to call the tangent method several times, once per input variable as follows,

```
01    tangent(2.0, 3.0, 1.0, 0.0);    // Input: x1 = 2, x2 = 3, x1_d = 1, x2_d = 0    Output: 8
02    tangent(2.0, 3.0, 0.0, 1.0);    // Input: x1 = 2, x2 = 3, x1_d = 0, x2_d = 1    Output: 3
```

Code 3: **Function Derivatives using Tangent Mode**

As the function $f(x_1, x_2) = 2x_1^2 + 3x_2$ with derivatives $df/dx_1 = 4x_1$ and $df/dx_2 = 3$ as per (equations 11 and 12) then (code 3) returns output values of 8 and 3 as outlined in (equation 13).

Now AD code can also be used for vectors and systems of equations. Consider a system of equations F with **n inputs** and **m outputs** as follows,

$$y = F(x), \qquad F: \mathbb{R}^n \to \mathbb{R}^m \tag{14}$$

Tangent mode operates forwards on a with an operational cost determined as follows,

$$\dot{F}(x, \dot{x}) = \underbrace{F'(x)}_{\mathbb{R}^{m \times n}} \cdot \underbrace{\dot{x}}_{\mathbb{R}^n} = \dot{y} \tag{15}$$

This means our derivative $F'(x)$ would be evaluated at a cost of $\boldsymbol{\mathcal{O}(n) \cdot Cost(F)}$. This is to say that the cost of the calculating the function F is once per input variable i.e. n times. Consequently tangent mode is preferred when the n<m i.e. when the number of function inputs n is less than the number of functions m.

2.2 Adjoint Mode (AAD)

When using adjoint mode we differentiate code in reverse order, starting with price sensitivities. Adjoint mode follows the reverse order of the original program, consequently we must compute the function value first (**'forward sweep'**) and store the intermediate values before applying adjoint AD in reverse (**'back propagation'**). This method shifts one price **output** at a time and generates price sensitivities exactly to machine precision for all price inputs in one go.

Bar Notation:

Adjoint mode is computed using **'bar'** notation for derivatives to denote the variable is to be differentiated with respect to the function input. For example given y = f(x) and working **in reverse** order gives $\bar{\boldsymbol{x}} = \boldsymbol{dy/dx}$.

Once again let us consider same simple function from equation 8),

$$\text{Function:} \quad y = 2x^2 \tag{16}$$

$$\text{Adjoint:} \quad \bar{x} = 4x.\bar{y} \tag{17}$$

The adjoint bar notation in (equation 17) is equivalent to,

$$\text{Adjoint:} \quad \frac{dy}{dx} = 4x.\frac{\boldsymbol{dy}}{\boldsymbol{dy}} \tag{18}$$

So again we ask what is the point of specifying $\bar{\boldsymbol{y}}$ denoting dy/dy? Isn't this value just one? Again when programming adjoint mode $\bar{\boldsymbol{y}}$ is specified as an input and allows us to enable/disable the adjoint derivative calculation. In the example above and (equation 17) setting $\bar{\boldsymbol{y}} = \mathbf{1}$ enables the derivative calculation giving dy/dx = 4x and when $\bar{\boldsymbol{y}} = \mathbf{0}$ we have dy/dx = 0.

Next let us consider how to add to add adjoint AD code to the same simple function from (8), requoted below for convenience.

$$\text{Function:} \quad f(x_1, x_2) = 2x_1^2 + 3x_2 \tag{19}$$

$$\text{Solution:} \quad \frac{df}{dx_1} = 4x_1 \text{ and } \frac{df}{dx_2} = 3 \tag{20}$$

When $x_1 = 2$ and $x_2 = 3$ we have,

$$\frac{df}{dx_1} = 8 \text{ and } \frac{df}{dx_2} = 3 \tag{21}$$

This function was implemented in (code 1) above let's apply adjoint AD (AAD) to this method, remembering that we are working backwards and in reverse. Consequently we must perform a forward sweep to evaluate the underlying function and store intermediate values for back-propagation and reverse calculation of derivative as follows,

```
01  void adjoint( double x1, double x2, double f_bar )
02  {
03      // Forward Sweep
04      double a = x1*x1;                   // Step 1:    a = x1^2
05      double b = 2*a;                     // Step 2:    b = 2x1^2
06      double c = x2;                      // Step 3:    c = x2
07      double d = 3*c;                     // Step 4:    d = 3x2
10      double f = b + d;                   // Step 5:    f = 2x1^2 + 3x2
08
09      // Back Propagation
10      double b_bar = f_bar;               // Step 5:    b_bar  = 1      from input variable
11      double d_bar = f_bar;               // Step 5:    d_bar  = 1      from input variable
12      double c_bar = 3*d_bar;             // Step 4:    c_bar  = 3
13      double x2_bar = c_bar;              // Step 3:    x2_bar = 3      df/dx2 = 3
14      double a_bar = 2*b_bar;             // Step 2:    a_bar  = 2
15      double x1_bar = 2*x1*a_bar;         // Step 1:    x1_bar = 4x1    df/dx1 = 4x1
16
17      // Display Results
18      std::cout << "df/dx1: " << x1_bar << std::endl;     // x̄1 = df/dx1 = 4x1
19      std::cout << "df/dx2: " << x2_bar << std::endl;     // x̄2 = df/dx2 = 3
20  }
```

Code 4: **Simple Function $f(x_1, x_x) = 2x_1^2 + 3x_2$ with Adjoint Derivative**

Source code: AD-Simple-Function.cpp
Available at: https://github.com/nburgessx/Papers/blob/main/SwapAD
To run see: https://onlinegdb.com/kKqaS6hJT

Note the function f(x1, x2) takes two inputs and in adjoint mode we capture both the x1 and x2 derivatives, namely x1_bar (line 13) and x2_bar (line 15). This means that we capture a function's derivatives with respect to all inputs in one go and only need to call the adjoint method once as follows,

```
01      adjoint(2.0, 3.0, 1.0);     // Input: x1 = 3, x2 = 2, f_bar    Output: df/dx1 = 8 and df/dx2 = 3
```

Code 5: **Function Derivatives using Adjoint Mode**

As the function $f(x_1, x_2) = 2x_1^2 + 3x_2$ with derivatives df/dx_1 = 4x1 and df/dx_2 = 3 as per (equations 19 and 20) then (code 5) returns output values of 8 and 3 as per (equation 21).

Now AD code can be used for vectors and systems of equations. Consider a system of equations F with **n inputs** and **m outputs** as follows,

$$y = F(x), \qquad F: \mathbb{R}^n \to \mathbb{R}^m \tag{22}$$

Adjoint mode operates backwards on a **matrix of vectors** having an operational cost determined as follows,

$$\bar{F}(x, \bar{y}) = \underbrace{\bar{y}}_{\mathbb{R}^m} \cdot \underbrace{F'(x)}_{\mathbb{R}^{mxn}} = F^T(x) \cdot \bar{y} = \bar{x} \tag{21}$$

This means our derivative $F'(x)$ would be evaluated at a cost of $\boldsymbol{\mathcal{O}(m) \cdot Cost(F)}$, that is the cost of the calculating the function F once per output function i.e. m times. Consequently adjoint mode is preferred when m<n i.e. when the number of function inputs n is greater than or equal to the number of functions m.

3. Swap Risk Examples

In this section we give an overview of how to use algorithmic differentiation for interest rate swap risk and specifically DV01 calculations. For other products and AD examples we refer the reader to (Savine, 2018) for vanilla options using Black-Scholes and (Capriotti, 2010) for equity basket options.

3.1 What is Swap PV01 and DV01 Risk?

Historically trading systems used LIBOR curves for all pricing. There was no separate OIS or RFR discount curve. Forwards and Discount factors in this regime were derived from the same curve. PV01 measured how swap prices changed for a parallel shift of all LIBOR rates by 1 basis point (bps). When LIBOR forward rates are shifted the corresponding discount factors also change in value, consequently PV01 captured both forward risk and discount factor risk. PV01 risk (Present Value of 1 bps) captured the risk in local currency and DV01 risk (Dollar Value of 1 bps) measured the PV01 in USD. When financial markets moved to a multi-curve environment, where forwards and discount factors are derived from separate curves, these risk measures were modified such that PV01 captured only the forward risk in local currency and DV01 captured both the forward risk and discount factor risk in local currency.

- **PV01 (Forward Risk)**

We shift all forwards by 1 basis point (bps) and measure the change in swap price.

➢ **Discount Factor Risk**

We shift all discount factors (by the equivalent of a 1 bps change in forwards[5]) and measure the change in swap price.

➢ **DV01 (Forward & Discount Factor Risk)**

DV01 measures forward and discount factor risk and is equal to PV01 plus discount factor risk. It measures the impact to both forwards and discount factors to a change in forward rates by 1 bps.

Financial practitioners are interested in the swap total DV01 (presented here), but also in bucketed DV01. The latter distributes the total DV01 risk to categories or buckets corresponding to the yield curve calibration instruments from which we derive our forward rates and discount factors. For more details on bucketed DV01, see (Burgess, 2021b).

3.2 Computing the Swap Price & DV01 Risk

Using (equation 1) for a swap with a single fixed coupon and a single float coupon we could price the swap in C++ code as follows,

```
01 // Swap Inputs
02 //   phi      Pay or Receive Fixed: Pay = 1,  Receive = -1
03 //   n        Swap Notional
04 //   r        Fixed rate
05 //   tau      Accrual year fraction
06 //   t        Coupon Payment Time
07 //   f        Floating Forward Rate
08 //   s        Floating Spread
09 //   z        Discounting Zero Rate for Discount Factor, where df = exp(-z*t)
10
11 double swap_pv(double phi, double n, double r, double tau, double t, double f, double s,
   double z)
12 {
03     double df         = exp(-z*t);                  // Step 1.  Discount Factor using zero rate, z
04     double pv_fixed = phi*n*r*tau*df;               // Step 2.  Fixed PV = φ N r τ1P(0,t1)
05     double pv_float  = -phi*n*(f+s)*tau*df;         // Step 3.  Float PV = φ N(l1 + s)τ1P(0,t1)
06     double pv_swap = pv_fixed+pv_float;             // Step 4.  Swap PV = Fixed PV + Float PV
07     return pv_swap;
08  }
```

Code 6: **Swap Price**

Source code: AD-Simple-Swap.cpp
Available at: https://github.com/nburgessx/Papers/blob/main/SwapAD
To run see: https://onlinegdb.com/H18OONnqi

[5] Note the discount factor – forward rate relationship is expressed as: $DF(0,T) = \exp\left(-\int_{u=0}^{T} f(0,u)du\right)$, which we simplify to DF(0,T) = exp(-zT) in this paper for simplicity.

Working forwards with tangent mode we can compute the swap DV01 as follows,

```
01 double tangent(double phi, double n, double r, double tau, double t, double f, double s,
   double z, double f_dot, double z_dot)
02 {
03     double df              = exp(-z*t);                    // Step 1.
04     double df_dot          = -t*exp(-z*t)*z_dot;
05     double pv_fixed        = phi*n*r*tau*df;               // Step 2.
06     double  pv_fixed_dot   = phi*n*r*tau*df_dot;
07     double pv_float        = -phi*n*(f+s)*tau*df;          // Step 3.
08     double pv_float_dot    = -phi*n*tau*df*f_dot
                              - phi*n*f*tau*df_dot;
09     double pv_swap         = pv_fixed+pv_float;            // Step 4.
10     double pv_swap_dot     = pv_fixed_dot + pv_float_dot;
11     return pv_swap_dot;
12 }
```

Code 7: **Swap DV01 using AD in Tangent Mode**

As an example to compute the price and risk for a 1Y USD 1,000,000 swap to receive fixed 2% vs float flat (no spread) with annual coupons, we call the tangent method as follows in (code 8) below. The change in price with respect to forwards is controlled by the f_dot parameter (dPV/dForward) and the change in price with respect to discount factors is controlled by z_dot (dPV/dDiscountFactor) via the zero rate. As DV01 captures the change in price for a 1bps (0.01%) change to both forwards and discount factors we set both f_dot and z_dot to 1 bps (i.e. 0.0001 in decimal) as follows,

```
01 // inputs( phi, n, r, tau, t, f, s, z, f_dot, z_dot )
02 tangent( 1, 1000000, 0.02, 1, 1, 0.01, 0, 0.02, 0.0001, 0.0001 );  // Output DV01 Risk
```

Code 8: **Swap Price and DV01 Risk using Tangent Mode**

Source code: AD-Simple-Swap.cpp
Available at: https://github.com/nburgessx/Papers/blob/main/SwapAD
To run see: https://onlinegdb.com/H18OONnqi

Working backwards using adjoint mode we can compute all risk sensitivities in one go. In this case we can compute the PV01 forward risk, discount risk and DV01 risks simultaneously. We present the swap DV01 using the adjoint method next in (code 9) below.

```
01  double adjoint(double phi, double n, double r, double tau, double t, double f, double s,
    double z, double pv_bar)
02  {
03      // Forward Sweep
04      double df           = exp(-z*t);            // Step 1.  Discount Factor using zero rate, z
05      double pv_fixed     = phi*n*r*tau*df;       // Step 2.  Fixed PV = ϕ N r τ_1 P(0,t_1 )
06      double pv_float     = -phi*n*(f+s)*tau*df;  // Step 3.  Float PV = ϕ N(l_1+s) τ_1 P(0,t_1 )
07      double pv_swap      = pv_fixed+pv_float;    // Step 4.  Swap PV = Fixed PV + Float PV
08
09      // Backward Propagation
10      double pv_fixed_bar = pv_bar;                                       // Step 4.
11      double pv_float_bar = pv_bar;                                       // Step 4.
12      double f_bar        = -phi*n*tau*df*pv_float_bar*shift_size_f;      // Step 3. *
13      double df_bar       = -phi*n*f*tau*pv_float_bar*shift_size_df;      // Step 3. *
14      df_bar              += phi*n*r*tau*pv_fixed_bar*shift_size_df;      // Step 2. *
15      double z_bar        = -t*exp(-z*t)*df_bar;                          // Step 1.
16
17      // DV01 Result
18      return f_bar + df_bar;   // Sensitivity to 1 bps change in forwards and discount factors
19  }
```

Code 9: **Swap DV01 using AD in Adjoint Mode**

** We scale f_bar and df_bar on lines 12-14 to present risk relative to a 1 bps change in forwards*

Source code: AD-Simple-Swap.cpp
Available at: https://github.com/nburgessx/Papers/blob/main/SwapAD
To run see: https://onlinegdb.com/H18OONnqi

We want our forward and discount factor derivative outputs to be relative to a 1 bps change in forwards and zero rates. To achieve this we scale our f_bar and df_bar outputs in (code 9) on (lines 12-14) by the appropriate shift sizes, see the raw source code and code commentary for details.

```
01  // inputs( phi, n, r, tau, t, f, s, z, pv_bar )
02  adjoint( 1, 1000000, 0.02, 1, 1, 0.01, 0, 0.02, 1 );  // Output DV01 Risk
```

Code 10: **Swap DV01 Risk using Adjoint Mode**

For demonstration clarity we illustrated above how to price a swap and compute the DV01 risk using tangent and adjoint AD modes in a simplified setting for swaps with a single coupon. To complete the demonstration for more generic swaps with multiple coupons, we refer the reader to the documented source code in the below appendix.

4. Professional AD Implementation

Whilst algorithmic differentiation is extremely powerful and can be implemented by hand for simple functions and routines, such as for swaps, applying AD by hand to more complex production code

can be a lengthy undertaking, error prone and require specialist knowledge, as code maintenance updating and refactoring becomes more complex. For this reason, financial practitioners and Quants often use AD software to support computing derivatives and risk sensitivities.

- **By Hand**

  Suitable and helpful for simple functions, basic products, well understood code patterns and medium sized projects.

- **Adept Expression Templates**

  Automatic **D**ifferentiation using **Exp**ression **T**emplates (Adept) is a free C++ software library that enables algorithms to be automatically differentiated and very useful for a wide range of applications that involve mathematical optimization, see (Adept n.d.). It uses an operator overloading approach, so very little code modification is required. Moreover, the way that expression templates have been used and several other important optimizations mean that reverse-mode differentiation is significantly faster than many other C++ libraries that provide equivalent functionality.

- **DCO/C++ and DCO/MAP (by NAG)**

  Derivative Code by Overloading in C++ (DCO) is a commercial software solution by (NAG n.d.). It comes with an ever growing number of features including,

  - Derivatives of arbitrary order (not just first order)
  - Vector tangent and adjoint modes
  - Very fast computation with expression templates and highly optimized tape
  - Tape compression and direct tape manipulation
  - Activity analysis and full control over memory use
  - Sparsity pattern detection

  If adjoints of GPU code are required then dco/c++ can be coupled with dco/map, a **Meta Adjoint Programming (MAP)** tool that uses template metaprogramming in C++ to generate adjoint code by overloading at compile time.

5. Conclusion

In conclusion we have presented algorithmic differentiation, both tangent and adjoint modes, and demonstrated how AD computes the derivative(s) of computer code. We looked at how to compute financial risks for interest rate swaps, such as DV01. We note that when trading in electronic rates markets, AD can be used to produce exact swap risks at high speed and give users a competitive advantage.

Firstly, we gave a brief overview of interest rate swaps and how to price them. Secondly we discussed swap risks, PV01 and DV01 risk calculations and the different approaches on how to compute them, namely via analytical closed-form solutions, numerical bumping, curve Jacobians. Thirdly we introduced algorithmic differentiation and showed how to implement AD in both tangent and adjoint modes on a simple function. Fourthly we showed how to compute DV01 risk for an interest rate swap using AD and finally we discussed how to professionally implement AD using both open-source and commercial software solutions. Throughout this implementation paper we presented AD case studies in C++ code and provided the raw source code.

To conclude Adept and NAG provide professional software solutions exist to facilitate and support AD computation using templates and derivative code by overloading (DCO) respectively. Regardless of the approach taken it is advisable to compare AD derivatives and risk outputs against their numerical bumping counterparts to check the quality of the risk results.

Appendix

Swap Pricing and DV01 Risk using Tangent and Adjoint AD

The below code is written in C++ to price an interest rate swap and compute swap risk using AD tangent and adjoint modes.

The source code is available at: https://github.com/nburgessx/Papers/tree/main/SwapAD, which can also be run here https://onlinegdb.com/uNgecMD9y using a free online compiler, simply press the run button (top-left).

```
01 #include <cmath>          // for math methods e.g. exponential function
02 #include <vector>         // for vectors
03 #include <iostream>       // for input/output to console
04 #include <iomanip>        // for input/output precision
05 using namespace std;
06
```

```
// Compute the swap present value
double price_swap( int payReceive,            // [IN]: Pay or Receive Fixed: 1 = pay, -1 = receive
                   double notional,           // [IN]: Swap Notional
                   double fixed_rate,         // [IN]: Fixed Leg: fixed rate
                   vector<double> fixed_tau,  // [IN]: Fixed Leg: coupon accrual year fractions
                   vector<double> fixed_t,    // [IN]: Fixed Leg: coupon payment time in years
                   double float_spread,       // [IN]: Float Leg: floating spread
                    vector<double> float_tau, // [IN]: Float Leg: coupon accrual year fractions
                    vector<double> float_t,   // [IN]: Float Leg: coupon payment times
                    vector<double> float_rates, // [IN]: Float Leg: floating forward rates
                    double zero_rate )        // [IN]: Discounting zero rate
{
  // Fixed Leg PV
  double fixed_pv = 0.0;
  double fixed_annuity = 0.0;
  for (size_t i = 0; i < fixed_t.size(); ++i)
  {
     fixed_pv += notional * fixed_rate * fixed_tau[i] * exp(-zero_rate*fixed_t[i]);
     fixed_annuity += notional * fixed_tau[i] * exp(-zero_rate*fixed_t[i]);
  }

  // Float Leg PV
  double float_pv = 0.0;
  for (size_t j = 0; j < float_t.size(); j++)
  {
     float_pv += notional*(float_rates[j] + float_spread)*float_tau[j] * exp( zero_rate*float_t[j]);
  }
    // Swap PV
    double swap_pv = payReceive * (fixed_pv - float_pv);
    return swap_pv;
}

// Compute the swap present value with risks using tangent mode
// Tangent mode uses forward differentiation where we perturb risk inputs
// Tangent risks are denoted 'dot' and risk variables have the suffix '_dot'
double swap_price_tangent_mode( int payReceive,  // [IN]: PayRecFixed: Pay = 1,  Receive = -1
              double notional,                   // [IN]: Swap Notional
              double fixed_rate,                 // [IN]: Fixed Leg: fixed rate
                vector<double> fixed_tau,        // [IN]: Fixed Leg: coupon accrual year fractions
                vector<double> fixed_t,          // [IN]: Fixed Leg: coupon payment time in years
                double float_spread,             // [IN]: Float Leg: floating spread
                vector<double> float_tau,        // [IN]: Float Leg: coupon accrual year fractions
                vector<double> float_t,          // [IN]: Float Leg: coupon payment times
                vector<double> float_rates,      // [IN]: Float Leg: floating forward rates
                double zero_rate,                // [IN]: Discounting zero rate
                vector<double> float_rates_dot,// [IN]: RISK INPUT - forward rate risk, bump size
                double zero_rate_dot )           // [IN]: RISK INPUT - discounting risk, bump size
{
    // Fixed Leg PV
    double fixed_pv = 0.0;
    double fixed_pv_dot = 0.0;
```

```
    for (size_t i = 0; i < fixed_t.size(); ++i)
    {
fixed_pv += notional * fixed_rate * fixed_tau[i] * exp(-zero_rate*fixed_t[i]); // df = exp(-z.t)
fixed_pv_dot += -fixed_t[i] * notional * fixed_rate * fixed_tau[i] * exp(-zero_rate*fixed_t[i])
                * zero_rate_dot;
    }

    // Float Leg PV
    double float_pv = 0.0;
    double float_pv_dot = 0.0;
    for (size_t j = 0; j < float_t.size(); j++)
    {
float_pv += notional * (float_rates[j] + float_spread) * float_tau[j] * exp(-zero_rate*float_t[j]);
float_pv_dot += notional * float_tau[j] * exp(-zero_rate*float_t[j]) * float_rates_dot[j];
float_pv_dot += -float_t[j] * notional * (float_rates[j] + float_spread) * float_tau[j]
                * exp(-zero_rate*float_t[j]) * zero_rate_dot;
    }

    // Swap PV
    double swap_pv = payReceive * (fixed_pv - float_pv);
    double swap_pv_dot = payReceive * (fixed_pv_dot - float_pv_dot);

    // Return Result
    Return swap_pv_dot;
}

// Compute the swap present value with risks using adjoint mode
// Adjoint mode uses backwards differentiation we perturb risk outputs to calculate all risks for
// each output. Reverse differentiation requires that we do a forward sweep to gather all
// variables before differentiating backwards. Adjoint risks are denoted 'bar' and risk variables
// have the suffix '_bar'
double swap_price_adjoint_mode( int payReceive,  // [IN]: PayRecFixed: Pay = 1,  Receive = -1
                 double notional,                // [IN]: Swap Notional
                double fixed_rate,               // [IN]: Fixed Leg: fixed rate
                vector<double> fixed_tau,        // [IN]: Fixed Leg: coupon accrual year fractions
                vector<double> fixed_t,          // [IN]: Fixed Leg: coupon payment times
                double float_spread,             // [IN]: Float Leg: floating spread
                vector<double> float_tau,        // [IN]: Float Leg: coupon accrual year fractions
                vector<double> float_t,          // [IN]: Float Leg: coupon payment time
                vector<double> float_rates,      // [IN]: Float Leg: floating forward rates
                double zero_rate,                // [IN]: Discounting zero rate in decimal
                double swap_pv_bar )             // [IN]: Enable Risk: 1=On, 2=Off
{
  // Compute Adjoint shift shift_sizes
  // We would typically use the curve jacobian for this and/or add an adjoint method
  // to the yield curve forward and discount factor interpolation methods
  double shift_size_f = 0.0001;
  double shift_size_z = 0.0001;

  // Fixed Discount Factor Shift Sizes
  vector<double> fixed_df;
```

```
109 vector<double> fixed_shifted_df;
110 vector<double> shift_size_fixed_df;
111
112 // Forward Sweep for Price
113 // ----------------------
114
115 // STEP 1: Fixed Leg PV
116 double fixed_pv = 0.0;
117 for (size_t i = 0; i < fixed_t.size(); ++i)
118 {
119    fixed_df[i] = exp(-zero_rate*fixed_t[i]); // Step 1.1
120    fixed_pv += notional * fixed_rate * fixed_tau[i] * fixed_df[i]; // Step 1.2
121 }
122
123 // STEP 2: Float Leg PV
124 double float_pv = 0.0;
125 for (size_t j = 0; j < float_t.size(); ++j)
126 {
127    float_df[j] = exp(-zero_rate*float_t[j]); // Step 2.1
128    float_pv += notional * (float_rates[j] + float_spread) * float_tau[j] * float_df[j]; // Step 2.2
129 }
130
131 // STEP 3: Swap PV
132 double swap_pv = payReceive * (fixed_pv - float_pv );
133
134 // Back Propagation for Risk
135 // -------------------------
136
137 // STEP 3. Risk from Swap PV Calculation
138 // double swap_pv = payReceive * (fixed_pv - float_pv );
139 double fixed_pv_bar = payReceive * swap_pv_bar;
140 double float_pv_bar = -payReceive * swap_pv_bar;
141
142 // STEP 2. Risk from Float Leg PV Calculation
143 // Note: We must follow loop steps in reverse order!!!
144 double float_rates_bar = 0.0;
145 double discount_factor_bar = 0.0;
146 double zero_rate_bar = 0.0;
147
148 for (size_t j = float_t.size(); j-- >0;)
149 {
150   // Apply shift sizes to scale output risks to 1bps as required
151   // float_pv += notional*(float_rates[j]+float_spread)*float_tau[j]*df; // Step 2.2
152    float_rates_bar += notional * float_tau[j] * float_df[j] * float_pv_bar * shift_size_f;
153    discount_factor_bar += notional * (float_rates[j] + float_spread) * float_tau[j]
154                                * float_pv_bar * shift_size_float_df[j];
155
156   // double discount_factor = exp(-zero_rate*float_t[j]); // Step 2.1
157    zero_rate_bar += -float_t[j] * exp(-zero_rate*float_t[j]) * discount_factor_bar;
158 }
159
```

```
160   // STEP 1. Risk from Fixed Leg PV Calculation
161   // Note: We must follow loop steps in reverse order!!!
162   for (size_t i = fixed_t.size(); i-- >0;)
163   {
164       // Apply shift sizes to scale output risk to 1bps as required
165      // fixed_pv += notional * fixed_rate * fixed_tau[i] * discount_factor; // Step 1.2
166      discount_factor_bar += notional * fixed_rate * fixed_tau[i] * fixed_pv_bar
167                                   * shift_size_fixed_df[i];
168     // double discount_factor = exp(-zero_rate*fixed_t[i]); // Step 1.1
169      zero_rate_bar += -fixed_t[i] * exp(-zero_rate*fixed_t[i]) * discount_factor_bar;
170   }
171
172   // Return DV01 Risk
173   return float_rates_bar + discount_factor_bar;
174 }
```

Code 11: **Swap Price, Tangent and Adjoint AD Risk Methods**

Source code: AD-Swap.cpp
Available at: https://github.com/nburgessx/Papers/blob/main/SwapAD
To run see: https://onlinegdb.com/uNgecMD9y

References

(Adept, n.d.) A combined automatic differentiation and array library for C++
Available at: http://www.met.reading.ac.uk/clouds/adept/

(Burgess N., 2016) How to Price Swaps in Your Head - An Interest Rate Swap & Asset Swap Primer
Available at: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=2815495

(Burgess, N., 2021a) Machine Earning – Algorithmic Trading Strategies for Superior Growth, Outperformance and Competitive Advantage. International Journal of Artificial Intelligence and Machine Learning, 2(1), 38-60. Available at: https://ssrn.com/abstract=3816567

(Burgess N., 2021b) NYU Yield Curve Seminar - An Overview of Yield Curve Calibration & LIBOR Reform. Available at: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3820145

(Capriotti, L., 2010) Fast Greeks by Algorithmic Differentiation
Available at: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=1619626

(Giles M. and Glasserman P., 2006) Smoking Adjoints: Fast Monte Carlo Greeks.
Available at: RISK Magazine, January 2006

(NAG, n.d.) Numerical Algorithms Group, Automatic Differentiation Solutions
Available at: https://www.nag.com/content/nagr-automatic-differentiation-solutions

(Savine A., 2018) Textbook: Modern Computational Finance: AAD and Parallel Simulations,

ISBN 978-1119539452. Available at:
https://www.amazon.co.uk/Modern-Computational-Finance-Parallel-Simulations/dp/1119539455