131890-S Optimization methods **Winter 2023/2024**

Lecture 3: Numerical approximations

Daniel Kaszyński *22 December 2023 r.*

3.1 Finite differences

A finite difference is a mathematical expression of the form $f(x+b) - f(x+a)$. If we divide the finite difference by $b-a$, we get the difference quotient. These approximation of derivatives are often used in finite difference methods for the numerical solution of differential equations. The difference quotients were discussed during lecture about derivatives of a function. This section will serve as a recap of those finite differences.

3.1.1 Forward and backward differences

The most popular derivative approximation methods are forward and backward finite differences. These are the ones we used most often in previous lectures.

Definition 1: Derivative of a function

By a derivative of a function $f : \mathbb{R} \to \mathbb{R}$ described by a **forward finite difference** we call a function:

$$f'(x) = \frac{df}{dx}(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{3.1}$$

By a derivative of a function $f : \mathbb{R} \to \mathbb{R}$ described by a **backward finite difference** we call a function:

$$f'(x) = \frac{df}{dx}(x) = \lim_{h \to 0} \frac{f(x) - f(x-h)}{h} \tag{3.2}$$

Numerical differentiation algorithms estimating the derivative of a mathematical function using forward finite difference or backward finite difference have error $O(h)$ (this can be proven using Taylor's Theorem).

From Taylor's Theorem we know that:

$$f(x+h) = f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + \frac{1}{6}f'''(x)h^3 + ... \tag{3.3}$$

We are able to transform the expression into:

$$f'(x)h = f(x+h) - f(x) - \frac{1}{2}f''(x)h^2 - \frac{1}{6}f'''(x)h^3 - ... \tag{3.4}$$

and dividing this expression by h we get:

$$f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{1}{2}f''(x)h - \frac{1}{6}f'''(x)h^2 - ... \tag{3.5}$$

So we can deduce that the error in the forward difference is of order $O(h)$.

Using the same methodology, we can transform the formula below:

$$f(x-h) = f(x) - f'(x)h + \frac{1}{2}f''(x)h^2 - \frac{1}{6}f'''(x)h^3 + ... \tag{3.6}$$

into a formula for the backward difference:

$$f'(x) = \frac{f(x) - f(x-h)}{h} + \frac{1}{2}f''(x)h - \frac{1}{6}f'''(x)h^2 - ... \tag{3.7}$$

The backward difference has also error of order $O(h)$.

3.1.2 Central difference

In addition to these two methods of calculating the derivative of a function, there is also a third one - using **central difference**. This method is sometimes called a symmetric derivative.

Definition 2: Symmetric derivative of a function

By a derivative of a function $f : \mathbb{R} \to \mathbb{R}$ described by a **central difference** we call a function:

$$f'(x) = \frac{df}{dx}(x) = \lim_{h \to 0} \frac{f(x+h) - f(x-h)}{2h} \tag{3.8}$$

Numerical differentiation algorithms estimating the derivative of a mathematical function using central difference have error $O(h^2)$. This is preferable as the error is smaller than previous methods.

Similarly to the previously described approximations, we can use Taylor's Theorem to derive a formula for central difference. However, this time we will need two starting equations:

$$f(x+h) = f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + \frac{1}{6}f'''(x)h^3 + ... \tag{3.9}$$

$$f(x-h) = f(x) - f'(x)h + \frac{1}{2}f''(x)h^2 - \frac{1}{6}f'''(x)h^3 + ... \tag{3.10}$$

By subtracting the two formulas above, we get:

$$f(x+h) - f(x-h) = 2f'(x)h + \frac{1}{3}f'''(x)h^3 + ... \tag{3.11}$$

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - \frac{1}{3}f'''(x)h^3 - ... \tag{3.12}$$

3.2 Subtractive cancellation error

When working on calculating derivatives of functions, we may encounter not only mathematical but also technical/hardware problems. One popular problem of this nature is **subtractive cancellation error**.

Subtractive cancellation error can occur while dealing with floating-point arithmetic. When computers work with real numbers, they must somehow store a potentially infinite number of decimal places in those numbers. For this purpose, they use, among others, float variables, which are a finite approximation of real

numbers. Most programming languages use a technical standard for floating-point arithmetic called **IEEE 754**. Unfortunately, precision is partially lost this way, but this is due to the computer's finite memory.

Subtractive cancellation error is a phenomenon that may be present when subtracting two nearly equal numbers. Floating-point numbers in computers have limited precision, and when you subtract two numbers that are very close in value, the result may suffer from loss of significant digits.

Example 1. Let $a = 0.3 + 0.3 + 0.4 - 1$ and $b = -1 + 0.3 + 0.3 + 0.4$.

Following simple mathematics, we can conclude that a is equal to b. Unfortunately, calculations performed on float variables may not give us the same result.

```r
a <- 0.3+0.3+0.4-1
b <- -1+0.3+0.3+0.4

print(a == b) # FALSE
print(a) # 0
print(b) # 5.551115e-17
```

Listing 1: Subtraction cancellation error example in R language

In this example, calculations performed to get a and b may result in very close, but different values. Those results might not be as accurate as one might expect due to subtractive cancellation error. The precision of the result depends on the number of significant digits that can be represented in the floating-point format. Therefore, we cannot assume that when subtracting float variables there will be no error which, although small, may spoil some simple comparisons and calculations.

To mitigate subtractive cancellation errors, various numerical analysis techniques and algorithms can be employed, such as rearranging the expression to avoid subtracting nearly equal numbers or using higher precision arithmetic when necessary. Additionally, understanding the limitations of floating-point arithmetic and being aware of potential sources of error is crucial when working with numerical computations in computer programs.

3.3 Complex Step Derivative

To avoid the problem of subtractive cancellation errors discussed in the previous section, various types of methodologies and formula transformations can be applied. One possible solution is to use **Complex Step Derivative**. The main idea behind this method is to take advantage of Taylor's equation and imaginary numbers to remove the need to subtract two float variables.

Let us start with Taylor's equation and let's use imaginary numbers to state it:

$$f(x + ih) = f(x) + f'(x)ih - \frac{1}{2}f''(x)h^2 - \frac{1}{6}f'''(x)ih^3 + ... \tag{3.13}$$

Assuming that we want to calculate the first derivative of the function, we need to transform the formula:

$$f'(x)ih = f(x + ih) - f(x) + \frac{1}{2}f''(x)h^2 + \frac{1}{6}f'''(x)ih^3 + ... \tag{3.14}$$

Then we need to isolate the derivative. We can start by dividing both sides by h:

$$f'(x)i = \frac{f(x + ih) - f(x)}{h} + \frac{1}{2}f''(x)h + \frac{1}{6}f'''(x)ih^2 + ... \tag{3.15}$$

To obtain only the derivative, we must also take care of the imaginary part:

$$f'(x) = Im\left(\frac{f(x+ih) - f(x)}{h}\right) + \frac{1}{6}f'''(x)h^2 + ... \tag{3.16}$$

Fortunately, we were able to drop $\frac{1}{2}f''(x)h$ because it did not contain an imaginary part. This simplified our formula, but it's still not as good as we'd hope because we haven't gotten rid of the subtraction of two function instances. To do this, we should split the first part of our formula:

$$f'(x) = Im\left(\frac{f(x+ih)}{h}\right) - Im\left(\frac{f(x)}{h}\right) + \frac{1}{6}f'''(x)h^2 + ... \tag{3.17}$$

We can notice that $Im\left(\frac{f(x)}{h}\right)$ has no imaginary part. So we can remove it from our equation. This leaves us with a formula that helps to avoid the subtractive cancellation error problem:

$$f'(x) = Im\left(\frac{f(x+ih)}{h}\right) + \frac{1}{6}f'''(x)h^2 + ... \tag{3.18}$$

This formula is the essence of Complex Step Derivative. Numerical differentiation algorithms estimating the derivative of a mathematical function using central difference have error $o(h^2)$. This is preferable as the error is smaller than previous methods. This is the fourth way we mentioned to calculate the derivative of a function.

3.4 Comparison of the finite differences

To better understand the differences between the presented finite differences, it is worth conducting a series of tests and comparisons.

Let $f = sin(x^2)$. Using the rules of symbolic differentiation, we can work out that $f'(x) = 2xcos(x^2)$.

Let $u = x^2$. Then, $\frac{du}{dx} = 2x$ and $\frac{df}{du} = cos(u) = cos(x^2)$. If we put this information together, we get the following equation:

$$f'(x) = \frac{df}{dx} = \frac{du}{dx}\frac{df}{du} = 2xcos(x^2) \tag{3.19}$$

Just to be sure, we can use the **R** programming language to calculate the derivative of the function f. To do this, we must first import the *Deriv* library, which is used to calculate derivatives:

```
1 if(!require(Deriv)) install.packages('Deriv');
```

Listing 2: Import of *Deriv* library

When using external libraries, it is often worth checking their documentation. In **R** programming language it can be done by adding '?' sign in frotn of library name:

```
1 # Access to library documentation
2 ?Deriv
```

Listing 3: Access to *Deriv* library documentation

Then, using this library, we can calculate the derivative of the function f:

```
f <- function(x) sin(x^2);
df <- Deriv(f)

cat('f = ', deparse(f)[2], '\n')
cat('df = ', deparse(df)[2], '\n')
```

Listing 4: Derivative of function f calculated using *Deriv* library

As a result of these calculations we get: $f = sin(x^2)$ and $df = 2xcos(x^2)$.

The plots of the function f and its derivative df would look as follows:

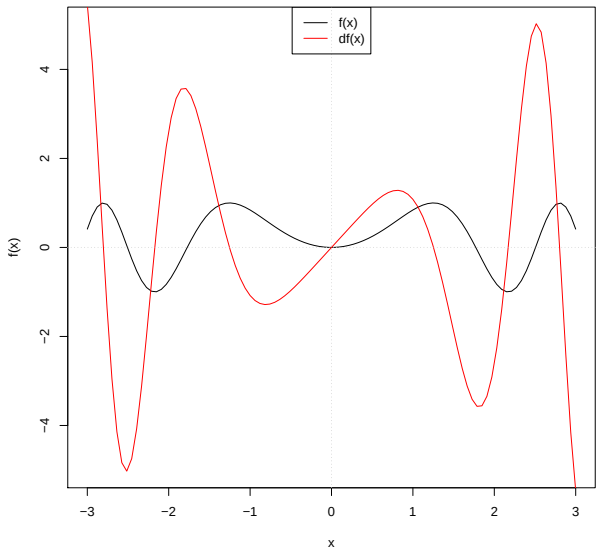

Figure 3.1: The plots of the function f and its derivative df

Of course, in order to compare different types of approximations of derivative functions in **R** programming language, we first need to have implementations of these methods. Knowing the definitions and formulas for these methods, creating these implementations is not a difficult task. For example, they may look like this:

```
diff_forward <- function (f, x, h = 10^-6) (f(x + h) - f(x)) / (h);
diff_backward <- function (f, x, h = 10^-6) (f(x) - f(x - h)) / (h);
diff_central <- function (f, x, h = 10^-6) (f(x + h) - f(x - h)) / (2*h);
diff_complex <- function (f, x, h = 10^-6 ) Im(f(x + h*1i )) / (h);
```

Listing 5: Finite differences implementations

Next, we can check the differences in the values obtained by these approximations at the given point $x_0 = 1$. The following code snippet gives us values for each of finite differences:

```
x0 <- 1

cat('df = ' , format( df(x0), nsmall = 20 ), " \n ")
cat('----------------------------------------' , " \n ")
cat('diff_forward = ' , format( diff_forward(f, x0), nsmall = 20), " \n ")
cat('diff_backward = ' , format( diff_backward(f, x0), nsmall = 20), " \n ")
cat('diff_central = ' , format( diff_central(f, x0), nsmall = 20), " \n ")
```

```
8 cat('diff_complex = ' , format( diff_complex(f, x0), nsmall = 20), " \n ")
```

Listing 6: Values obtained by finite differences in point $x_0 = 1$

df = 1.08060461173627953002

---

diff_forward = 1.08060346903915416306

diff_backward = 1.08060575443325035394

diff_central = 1.08060461179171340973

diff_complex = 1.08060461173868294082

As we can see, these values are close to each other and do not deviate too much from the real value of the function's derivative. Method diff_complex achieves the best result because it is closest to the actual value.

In addition to checking individual function values for each of the finite differences, we can do many other comparisons. For example, we can plot how fast relative error converges to 0 in terms of all of the numerical differences. The plot that would be created for those purposes should have exponential scale, because everything that is interesting to us will take place in a very narrow ranges. To improve visibility, logarithmic scale is also recommended.

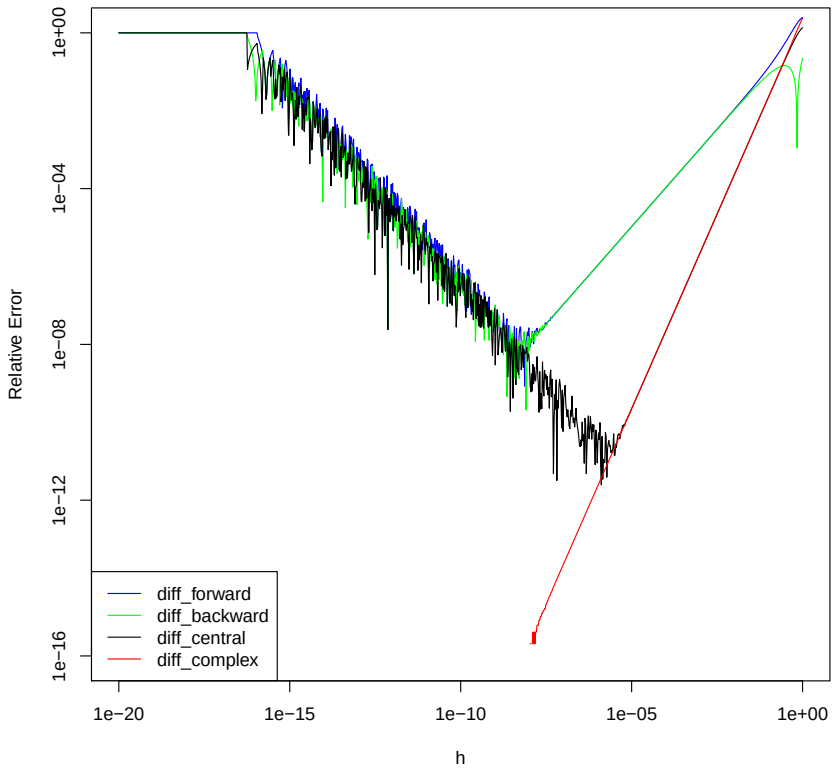

Figure 3.2: Relative error for each of the finite differences on f

As we can see in the plot above, the characteristics of how relative error changes depending on h differ for each approximation method. By tracking the values relative to the h axis from right to left, we can see how relative error converges to zero (or in most cases just attempts to converge). For each finite difference we can define the following behavior:

- **diff_forward** and **diff_backward** - Forward finite difference and backward finite difference behaved very similar to each other while trying to converge relative error to zero. Both methods achieved a minimum error at an h of approximately $1e-08$. From that point on, however, the error began to increase. It was caused by described earlier subtractive cancellation error (look section 3.2). The error caused by subtracting two float variables for smaller h values significantly affected the quality of the results in those cases.
- **diff_central** - Central difference behaved similarly to the two previously described methods, with a few notable differences. First of all, at the beginning this method reached its minimum much faster at around $1e-05$. The slope of the relative error convergence for this approximation was much steeper. This was caused by the fact that central difference has error $O(h^2)$. Furthermore, the relative error values remained on average slightly below those obtained by forward and backward finite differences.
- **diff_complex** - Complex step derivative approximation performed the best out of all described methodologies. Not only did it converge to zero at a rate comparable to central difference, but also it didn't suffer from the problem of the subtractive cancellation error. This allowed this method to achieve relative error values close to zero to the point that the **R** language did not distinguish them from zero (which is why they disappeared from the plot).

3.5 Automatic differentiation

Automatic Differentiation, also known as algorithmic differentiation or autodiff, is a technique used to efficiently and accurately evaluate the derivatives of mathematical functions. The primary goal of this technique is to automatically and systematically compute the derivatives of a given function, making it especially useful in optimization, machine learning, and scientific computing.

Automatic differentiation sets itself apart from symbolic differentiation and numerical differentiation. Symbolic differentiation encounters challenges in converting a computer program into a unified mathematical expression, often resulting in inefficient code. On the other hand, numerical differentiation, employing the method of finite differences, may introduce round-off errors during the discretization process and face issues related to cancellation. These traditional methods struggle when calculating higher derivatives. In contrast, automatic differentiation effectively addresses and resolves all these issues.

To fully understand how automatic differentiation works, we must first become familiar with a few basic ideas behind it. First of all, the decomposition of differentials provided by the **chain rule** of partial derivatives is fundamental to automatic differentiation.

The chain rule is a fundamental concept in calculus that describes how to find the derivative of a composite function. Mathematically, if you have the composition of two functions $f(x)$ and $g(x)$ such that $f(g(x))$, then the chain rule states that the derivative of this composition with respect to x is the product of the derivative of f with respect to its argument $g(x)$ and the derivative of g with respect to x:

$$\frac{df}{dx}f(g(x)) = \frac{d}{dx}(f \circ g)(x) = \frac{df}{dg}\frac{dg}{dx} = f'(g(x))g'(x) \tag{3.20}$$

Another thing worth paying attention to is the fact that automatic differentiation uses computational graphs (explicitly or implicitly). A computational graph is a representation of a mathematical expression or a com-

putational process. It is commonly used to visualize and understand the flow of computations involved in evaluating a function or performing a series of operations.

Most mathematical formulas can be broken down into a series of basic arithmetic operations (e.g., addition, multiplication, exponentiation). To create a computational graph, we first create nodes. Nodes in a computational graph represent mathematical operations or functions. Each node corresponds to a specific computation, such as addition, multiplication, or a more complex operation. Edges in the graph depict the flow of data or dependencies between the operations. An edge from one node to another indicates that the output of the first operation is used as an input for the second operation. The inputs to the computational graph are usually represented as nodes with no incoming edges, while the outputs are nodes with no outgoing edges.

Example 2. Let $f(x_1, x_2) = (cos(\frac{x_1}{x_2}) + \frac{x_1}{x_2} - sin(x_2)) \cdot (\frac{x_1}{x_2} - sin(x_2))$. Computational graph for such a function would look as follows:

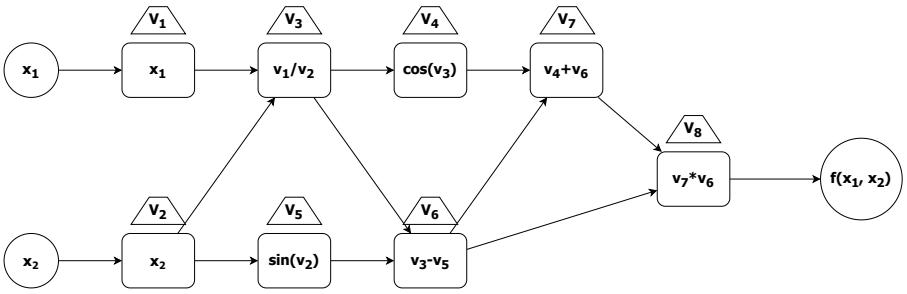

Figure 3.3: Example computational graph of function $f(x_1, x_2)$

Another important part of the algorithm is the use of *Dual Numbers*. In each node of the computational graph not only we will calculate primals of the function, but we will simultaneously compute their derivatives. This simple trick will help us reuse already calculated in previous steps components of the formula in future operations. At the same time, we limit ourselves to calculating derivatives of only simple arithmetic operations.

In the **R** programming language, we can represent such *Dual Numbers* using object-oriented programming:

```
DualNumber <- function(val, eps=0) {
    obj <- list(val = val, eps = eps)
    class(obj) <- "DualNumber"
    return(obj)
}
```

Listing 7: Dual Number implementation

Last but not least, there is also the issue of calculating the results of individual operations in the computational graph. An elegant way to approach this problem is to use **operator overloading**. For each operator (such as '+' or '-'), we can use it to define a different behavior specifically tailored to our *Dual Numbers*.

Lets create overloads for each of the basic operations. Firstly, we can start with addition operator - '+':

```
"+" <- function (x, y) {
  if (class(x) == "DualNumber") {
    val <- x$val + y$val
    eps <- x$eps + y$eps
    return(DualNumber(val, eps))
```

```
  } else {
    .Primitive("+")(x, y)
  }
}
```

Listing 8: Addition operator overload

Each operator is a function that takes two parameters as input and gives us the result. First, we should distinguish between the effect of the operator for *Dual Numbers* and other values (for which the effect of the operator will remain unchanged). If we want to add two *Dual Numbers*, we must both add their values and add the values of their derivatives. After that we can return newly created result as a *Dual Number*.

Proceeding in a similar way, we can implement operator overloading for the subtraction operator - '-':

```
"-" <- function (x, y) {
  if (class(x) == "DualNumber") {
    val <- x$val - y$val
    eps <- x$eps - y$eps
    return(DualNumber(val, eps))
  } else {
    .Primitive("-")(x, y)
  }
}
```

Listing 9: Subtraction operator overload

When overloading the multiplication operator - '*', we should recall *Leibniz product rule* (a formula used to find the derivatives of products of two functions):

$$(f(x)g(x))' = f'(x)g(x) + f(x)g'(x) \tag{3.21}$$

```
"*" <- function (x, y) {
  if (class(x) == "DualNumber") {
    val <- x$val*y$val
    eps<-y$val*x$eps + x$val*y$eps
    return(DualNumber(val, eps))
  } else {
    .Primitive("*")(x, y)
  }
}
```

Listing 10: Multiplication operator overload

As the last operator, that we will do for the sake of this example implementation, we can overload power operator. The formula for the derivative of power function may be helpful here:

$$f'(x) = \frac{d}{dx}(x^k) = kx^{k-1} \tag{3.22}$$

We just need to remember to multiply the value obtained using this formula by the previously calculated value of the derivative:

```
"^" <- function (x, k) {
  if (class(x) == "DualNumber") {
    val <- x$val^k
    eps <- k*x$val^(k-1)*x$eps
    return(DualNumber(val, eps))
  } else {
    .Primitive("^")(x, k)
  }
}
```

Listing 11: Power operator overload

Dual Numbers and overloaded operators are enough for the automatic differentiation algorithm to work. Now we can move on to testing the algorithm.

Example 3. Let $A = 5$ with sensitivity over 1-st variable, $B = 4$ with sensitivity over 2-st variable and $C = 7$ with sensitivity over 3-st variable. Furthermore, let $f(x_1, x_2, x_3) = x_1^2 + x_1x_2 + x_2 + x_3$:

```
# Declaration of DualNumer values
A <- DualNumber(5, c(1,0,0))
B <- DualNumber(4, c(0,1,0))
C <- DualNumber(7, c(0,0,1))

# Calculation of function f(A, B, C) values
A^2 + A*B + A + C
```

Listing 12: Automatic differentiation example

When we run this calculations, we would get following results:

$val

[1] 57

$eps

[1] 15 5 1

attr(,"class")

[1] "DualNumber"

As we can see, when performing a simple operation on numbers of the *Dual Number* type, we can obtain both the value of the f function and the value of the derivative over each of the input variables. What's more, obtained values for *$eps* is precisely a gradient of function f in given point.