# Does functional programming matter for functional programming languages? A case study in numerical differentiation and integration

**Introduction:** In his seminal paper, John Hughes ([1990](https://www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf)) promotes functional programming also with several examples from numerical mathematics. He attributes the presented numerical algorithms the quality “quite good” (p. 16). A survey of modern numerical textbooks (Bartels, 2016; Bärwolff, 2016; Friedrich and Pietschmann, 2020; Knorrenschild, 2013; Scholz, 2016) indicates that his appeal was not reflected by the community of numerical practitioners as neither his numerical algorithms nor functional programming are discussed in them. This is not surprising as numerical mathematics was a branch of software already productively developed in 1990 (e.g. the [numerical recipes](https://en.wikipedia.org/wiki/Numerical_Recipes) series started in 1986). John Hughes specifically mentions (p. 2) that his promotion might have limited appeal for FORTRAN programmers. Numeric computation is a stronghold for this [bedrock](https://en.wikipedia.org/wiki/Fortran) of a programming language. Furthermore, numerical practitioners might not have been the targeted audience. The choice of numerical algorithms is reasonable because of the broad appeal of numerical mathematics for software engineers. If numerical mathematics should have more welcomed the numerical algorithms of John Hughes is a question outside the scope of our paper. We focus on a narrower question: Are these numerical algorithms a suitable choice for functional programmers themselves? In order to attain an answer, we test the numerical algorithms at their task at hand: approximating the values of derivatives and integrals. We find this an interesting case study to investigate potentials and pitfalls of functional programming at a key task of computing: calculating.

Our case study has the following framework: We implemented the algorithms of Hughes for the approximation of derivates and integrals in the functional programming languages [Frege](https://github.com/Frege/frege) and [Haskell](https://www.haskell.org/). As contrast, we implemented according algorithms from the numerics textbook of Günter Bärwolff (2016) in these languages. The algorithms in this book are designed for imperative languages. In order to ensure that the adaption of the imperative algorithms to the functional programming languages fulfills the standards, we additionally consulted the Haskell numeric library [numeric-tools](https://hackage.haskell.org/package/numeric-tools). We also provide translations of an algorithm from this library in Frege [work in progress]. For these algorithms, we provide an evaluation of their performance and their suitability for the functional programmer. We additionally discuss their mathematical rigor.

In the case of differentiation, a non-numerical approach to calculate derivatives called automatic differentiation is shown to be tailor-made for Haskell ([crypto.standford.edu](https://crypto.stanford.edu/~blynn/haskell/ad.html) and [Brice](https://crypto.stanford.edu/~blynn/haskell/ad.html), 2015). We additionally provide translations of these Haskell algorithms into Frege. However, differences in the numerical type classes and data types between Frege and Haskell (at least preliminarily) prevent us to implement the full scope of the original programs. Still, the Frege-implementations thus far are useful versions of automatic differentiation.

In the following, we first discuss (numerical) differentiation. As the main discussion points are encapsulated in these chapters, the chapter on integration is restricted on presenting the algorithms. A general discussion finalizes our work.

## Numerical and automatic differentiation

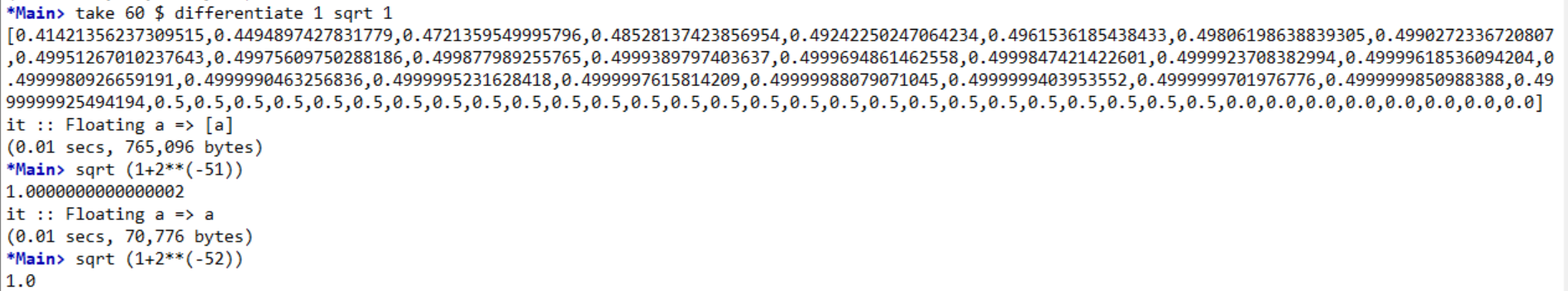
Unless stated otherwise, we always provide a Frege translation and a corresponding Haskell translation of the focal program. We aim to keep the translations as close as possible to each other.

For a continuous function , the derivative at the -value as the limit value . The sequence of difference quotients is expected to increasingly approximate the value of if the absolute values of are decreasing from element to element (). We speak of **numerical** differentiation if the approximation of the derivative bases on such sequences. We emphasize that our alternative form of differentiation, i.e. automatic differentiation, also involves numeric methods.

**The numerical differentiation algorithms of Hughes:** Translations of the algorithms for numerical differentiation from Hughes are given in the files **DiffHughes.fr** (Frege) and **DiffHughes.hs** (Haskell). We consider Frege as the focal language of our study. The Frege code will always be more elaborately commented especially with regard to the modus operandi of the program. Frege is “[Haskell for the JVM](https://github.com/Frege/frege)” and the two translations are thus fairly congruent. For each tested input choice, the two programs consistently produced the same output (data not shown). The two programs can produce approximations of the derivatives. For example for the function at , we attain with the instruction

> within 0.01 (differentiate 1 *sqrt* 1)

a value that deviates by less than 0.01 from the true value of (As the algorithms are translated from Hughes (1990), we refer to this paper for a detailed explanation of the instructions used in our paper. A description of Hughes’ algorithm in mathematical terminology is given in Box.docs). However, we stress that a satisfactory outcome is not guaranteed. Using the stream of difference quotients generated by differentiate 1 *sqrt* 1, we demonstrate a problematic case:



The first output shows that the sequence of difference quotients initially converges and eventually attains the limit value . However, the final values of this sequence deviate from this value. This deviation can be explained by floating point arithmetics: for sufficiently small -values, the numeric square root function (wrongfully) calculates (compare the final two outputs in the REPL depiction above). The calculation of the difference quotient, i.e. (f (x + h) - f x) / h, is then wrong beyond the measure attributed to floating point rounding errors.

In its further continuation, the discussed sequence of difference quotients will eventually switch from 0.0 to NaN (because will be so small that the floating point arithmetics attributes it the value The resulting division by zero causes the NaN outcome). Due to such NaN outcomes, the instruction

> within 0.001 (super (differentiate 1 *sqrt* 1))

will not abort. We emphasize that the function super requires the calculation of logarithm values which might be a further cause for NaN outcomes.

In the two previous paragraphs, we discuss two forms of failures for the numeric calculation of the convergence of difference quotients towards the limit value . We use this discussion to pinpoint an inadequacy of the function within. For a sequence , convergence is defined as whereby is the limit value, is an arbitrary positive value and integer for a natural number which depends on the choice of . The function within bases on an deviation of this definition to whereby is the smallest natural number for which the inequality is satisfied (e.g. it is not checked if ). This choice bases on the assumption that . In practice, this approach should indeed regularly result in satisfactory outcomes. Still, we discuss in the next section that numerical methodology has higher standards with respect to accuracy of results. Furthermore, the choice of implies that the user of within has no a priori knowledge of the number of calculation steps required for executing this function: the sequence is traversed until the inequality is satisfied whereby the user will not a priori know the length of this traversal. In contrast, the method presented in the next chapter provides more control on the number of calculation steps.

We mentioned floating point inaccuracies as cause for failures. This implies switching from floating point numbers to a data type like BigDecimal as a potential solution. As the following example illustrates, this change in data type can indeed fix some of these issues:



The depicted outcome corresponds to the last 0.0-value in the (previously) discussed sequence of difference quotients. Instead of 0.0, this outcome is practically the expected limit value of . Unfortunately, the fact that we obtain a BigDecimal outcome does not translate to all other instructions. In the file **DiffHughesBigDecimal.hs**, we adjusted the functions improve and super such that they accept BigDecimals as argument values. A comment in this file provides an instruction that delivers an outcome. We had to struggle to find this instruction as most other instructions failed because the calculation with BigDecimal values exhausted the resources of the REPL. [Furthermore, BigDecimal has additional limitations: e.g. there is no square root function for Java-BigDecimal values.]

**Standard numerical integration:** The programs in the files **PolynDifferentiation.fr** and **PolynDifferentiation.hs** contain our implementation of the numerical differentiation method descripted by Bärwolff (2016, p. 130-131). Our implementation differs from this template in that we generate the required interpolation polynomials by the function given by the Haskell library [Math.Polynomial](https://hackage.haskell.org/package/polynomial-0.7.3/docs/src/Math-Polynomial-Interpolation.html#polyInterp) (as the authors of the Haskell library [Math.polynomial.Lagrange](https://hackage.haskell.org/package/polynomial-0.7.3/docs/Math-Polynomial-Lagrange.html) attribute this method of generating interpolation polynomials the quality of being quicker and more stable than alternative methods). Generating the polynomials involves the auxiliary function neville. The Frege implementation of neville deviates from the Haskell template (given by [Math.Polynomial](https://hackage.haskell.org/package/polynomial-0.7.3/docs/src/Math-Polynomial-Interpolation.html#polyInterp)). The reason is that the template uses parallel list comprehension. To our knowledge, parallel list comprehensions are not a feature in Frege.

For the function , the following instruction returns an approximation of the derivative value :

> firstDerivative *sqrt* 1.0 [0.3, 0.2, 0.15, 0.1]

The result deviates by less than from the expected result. This deviation can be reduced to less than by changing the list of -values to [0.03, 0.02, 0.015, 0.01]. The textbook of Bärwolff does not provide much information on the choice of the values in this list. Their approach is motivated by that it avoids errors caused by small positive floating point numbers (p. 129; as reminder, small -values caused the issues in the approach of Hughes). Furthermore, Bärwolff mentions (p. 129) that practical experience suggest that 4 values suffices for this list. The runtime of firstDerivative is determined by composing the tableau of Neville (i.e. the function Neville) and it is thus . More importantly, the runtime is a priori determined by the -value. In conclusion, our results show that for , fairly accurate results can be obtained with standard numerical differentiation in Frege and in Haskell. This approach is designed to evade errors caused by small -values. Because we attribute the failures in the approach of Hughes to such values, we have good reason to assume that standard numerical differentiation is safer in this respect. Furthermore, the runtime is a priori defined and polynomial in standard numerical differentiation.

**Automatic differentiation:** An alternative to numerical differentiation, i.e. automatic differentiation, is shown ([Lynn](https://crypto.stanford.edu/~blynn/haskell/ad.html), [Brice](https://www.danielbrice.net/blog/automatic-differentiation-is-trivial-in-haskell/)) to be easily and fruitfully implementable in Haskell (see also the according Haskell library [ad](https://hackage.haskell.org/package/ad)). Automatic differentiation calculates the derivatives using the rules of derivation and the commonly used derivation formulas for ordinary functions like the trigonometric functions (for a more detailed description, see [Wikipedia](https://en.wikipedia.org/wiki/Automatic_differentiation)). Basically, derivatives are practically calculated in the same way as high school students learn it. Consequently, automatic differentiation inherits some of the advantages that come with this form of calculating derivatives. We thus view the transfer of automatic differentiation to Frege as a worthwhile endeavor. The program of Ben Lynn works in Frege (file **Test.fr**) but the crux of his approach involves symbolic numbers (Data.Number.Symbolic in Haskell). To our knowledge, symbolic numbers are not yet added to the Frege library. Furthermore, we view the approach of Daniel Brice as more user friendly. Hence, our focus is on the program of Daniel Brice (files **AutoDiff2.fr** and **AutoDiff2.hs**).

The program of Daniel Brice introduces the numerical data type Dual u u'. For example, Dual 2 1 is the number 2 as value of the variable while Dual 2 0 is this number as a constant. The underlying idea is Dual whereby in Dual 2 1, the function is and and thus and . In Dual 2 0, the function is and . For , the derivative value is the last number in the following output:

Ein Bild, das Text enthält.

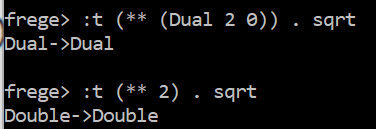
Automatisch generierte Beschreibung

This outcome shows the correct result rather than an approximation. The first output-number is the function evaluated at , i.e. .

Our aim is that the additional numerical data type Dual interacts as flawlessly as possible with the conventional numerical data types like Double. The following Frege-outputs for the function (i.e. an elaborative -function) are representative for the corresponding Haskell-output:

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

The argument in the first two instructions (above) is a Dual number while the argument in the latter two instructions is a (conventional) number (here a Double). In the second and fourth instruction, the exponent is a Dual number. The output of the third instruction is a Double number and the other three outputs are Dual numbers rather than integers. Both the instructions and their outputs illustrate that Dual numbers can be intermixed with conventional numbers. In order to achieve this feature in Frege, we had to deviate from the template given by Daniel Brice. The type of our -function in the two languages, i.e.

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

highlights the difference between the implementations: Using a Dual exponent sets the type to (Dual -> Dual) in both languages. Using a integer exponent sets the type to (Floating c => c -> c) in the Haskell program and to (Double -> Double) in the Frege program. In Haskell, it suffices to implement an instance Floating Dual. In the Frege program, we additionally implemented the class Floating and an instance Floating Double. Frege has the class Floating and the instance Floating Double already implemented in the library [frege.prelude.Math](https://github.com/Frege/frege/blob/master/frege/prelude/Math.fr). We refrain from importing this class and instance from this library. We exemplify the reason by the type declaration for the square root function in the class Floating: sqrt ∷ r → Double. The outcome is a Double because frege.prelude.Math uses the sqrt-function from Java and the Java-function returns a Double outcome. However, we require Dual outcomes for Dual arguments and hence, our class needs to be defined as sqrt ∷ r → r. Hence, we re-defined the class Floating and accordingly, the instance Floating Double. We emphasize that these re-definitions are implemented to the minimal degree required for automatic differentiation to function as intended. We do not view the re-definitions as valid substitutes for the original definitions in frege.prelude.Math and we encourage using the originals for any other purpose.

Ein Bild, das Text enthält.

Automatisch generierte BeschreibungIn particular if differentiation is a primary reason for a program, we view automatic differentiation as competitive if not superior to numerical differentiation. If additional Mathematics is required then forgoing the library frege.prelude.Math (as we did) likely causes further problems. Our implementations of automatic differentiation in Haskell and Frege highlights a case where the differences in the [numerical data classes and types](https://github.com/Frege/frege/wiki/Differences-between-Frege-and-Haskell) between the languages can be disadvantageous for the Frege programmer. In order to highlight the inferiority of our Frege implementation, we copy the final example of Daniel Brice:

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

The defined function is more polymorphic in the Haskell program (left hand side) than in the Frege program. In the frege-repl (right hand side), we had to use a lambda function (final instruction) to show that the function definition per se would actually apply for Dual arguments.

## Numerical integration

An integral is defined as whereby . The limit on the right hand side of the equation can be approximated by setting in increasing values of in the associated term. We refer to any such form of approximation as numerical integration. All method that we discuss fall into this rubric.

**The integration algorithms of Hughes:** Our implementations of the numerical methods of John Hughes (1990) are given in **IntegrationHughes.fr** and **IntegrationHughes.hs**. The function integrate f a b generates a stream of the values of for . This function is improved by the function integrate2 as it recycles -calculations from previous elements in the calculation of the next element. Hence the functions integrate and integrate2 correspond to the function differentiate in **DiffHughes.fr**. Correspondingly, convergence to the limit value can be speed up by the functions improve and super. As consequence, the numerical integration of John Hughes inherits the advantages and disadvantages of his differentiation methods. We view them as properly discussed in the subchapter “the differentiation algorithms of Hughes”.

**Standard numerical integration:** The functions simpson f a b n and gaussTschebyscheffQuadratur f a b n in the file **PolynDifferentiation.fr** approximate the limit using the Simpson rule respectively Gaussian quadrature. The corresponding algorithms are translated from the algorithms of Bärwolff (2016) presented at p. 160 and p. 171. Both algorithms are based on a *foldl* (*fold* in Frege) and hence, we expect the performance to be . More importantly, the number of calculation steps is a priori set. In order to exemplify the accuracy of the functions, we show outputs for two different -values:

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

For both -values, gaussTschebyscheffQuadratur delivers the better approximation of the limit (see the four outputs above). For both functions, the approximation improves with the increase of the -value. The errors in all four approximation are considerably less than . In fact, the erroneousness of the approximation can be limited (see p. 157fff and p. 171ff in Bärwolff, 2016) by the choice of parameters (in particular ).

We consider this quality of the standard numerical integration that the accuracy of the approximation can be a priori determined by the programmer as superior over the approach proposed by John Hughes (1990). His approach is represented by the function within. Given a stream then within outputs the first value with (whereby is the parameter eps of within). This approach presumes whereby is the limit value of the stream. In practice, this equivalency might frequently apply but cannot be guaranteed (proof: take the stream 1, 2, 1000, 1000, 1000, … with the output is 2 but with ). We emphasize that also the accuracy of our standard numerical integration methods is not guaranteed as the upper limit for the errors applies only for a defined set of conditions (see p. 157fff and p. 171ff in Bärwolff, 2016). However, this uncertainty is considerably more narrowly defined than the corresponding uncertainty discussed for the function within.

## Discussion

We took it as our task to evaluate the numerical programs from the seminal paper of John Hughes. These programs run in the two functional programming languages Frege and Haskell: they can provide sufficiently accurate approximations of derivatives and integrals. However, we found conditions where calculating approximations fails whereby the conditions involve a standard mathematical function (i.e. *sqrt*) and a fairly low accuracy. Of course, the according deficiencies in the algorithms could be addressed and to a certain degree fixed respectively damped e.g. by a form of exception handling. We propose an alternative route: Using classical numeric methods instead of the algorithms proposed by John Hughes. The imperative algorithms from a numerical textbook can readily be translated into the discussed functional programming languages. The behavior of the according numerical differentiation and integration is known with respect to stability and accuracy. The time performance of the textbook algorithms is polynomial. Indeed, the numerical library [Numeric.Tools](https://hackage.haskell.org/package/numeric-tools) also takes this approach (see e.g. the reference at the end of the code of [Numeric.Tools.Differentiation](https://hackage.haskell.org/package/numeric-tools-0.2.0.1/docs/src/Numeric-Tools-Differentiation.html)).

In the context of the function within, we criticize that the performance is not a priori determined. The approach of traversing a sequence until the difference between two subsequent elements is sufficiently small is commonly used in science (e.g. [Spichtig and Kawecki, 2004](https://pubmed.ncbi.nlm.nih.gov/15266372/), used it to find equilibria). In such endeavors, the approach is mostly used to find a final objective. If the aim is not programming a calculator then differentiation and integration are typically expedients. A programmer that requires e.g. integration as a side task wants this side task to function as reliably as possible. Thus in the context of numerical differentiation and integration, the answer to the question “does functional programming matter for the functional programmer?” is a clear “No! Go the classical route.” We re-emphasize: we do not believe that John Hughes’ aim was to propose competitive alternatives to classical numerical differentiation and integration. We view his choice of numeric examples due to their generalist appeal as a natural choice to highlight the potentials of functional programming. Our negative conclusion in no way chips away from the seminal characteristic of his paper.

If differentiation is the task at hand then we view automatic differentiation as a considerable choice. Our implementation of the (Haskell) template from Daniel Brice covers a decent range of the original functionality. We attribute the deficits to the [differences](https://github.com/Frege/frege/wiki/Differences-between-Frege-and-Haskell) in the numerical type classes and data types between Frege and Haskell. We suspect that attaining full functionality would require a careful alteration of a copy of frege.prelude.Math (i.e. our approach at a larger scale). As Frege is “[Haskell for the JVM](https://github.com/Frege/frege)”, the differences in the numerical types are probably due to the choice to use the mathematical functions from Java. As this choice seems judicious, we fear that any attempt to attain full functionality might fail. Anyhow, implementing automatic differentiation requires much more tinkering in Frege than in Haskell. We thus view it as an example where the differences in the numerical types are disadvantageous for Frege. However, the approach we took should be considered in this judgement: We used a program that was shown to be perfectly suited for Haskell. The Haskell template we used was thus already at the top and all path go down whenever you are on the top!

# Bibliography

This is a stub as all internet references are preliminary linked directly in the text. In the final version, they will be added here!

Bartels Sören, 2016, Numerik 3x9: Drei Themengebiete in jeweils neun kurzen Kapiteln, Springer Spektrum.

Bärwolff Günter, 2016, Numerik für Ingenieure, Physiker und Informatiker, Springer Spektrum.

Knorrenschild Michael, 2013, Numerische Mathematik: Eine beispielorientierte Einführung, Fachbuchverlag Leipzig im Carl Hanser Verlag.

Scholz Daniel, 2016, Numerik interaktiv: Grundlagen verstehen, Modelle erforschen und Verfahren anwende mit taramath, Springer Spektrum

Friedrich Hermann und Pietschmann Frank, 2020, Numerische Methoden: Ein Lehr- und Übungsbuch, De Gruyter Studium.