Comparative Verification of the Digital Library of Mathematical Functions and Computer Algebra Systems

André Greiner-Petter[1] (✉), Howard S. Cohl[2], Abdou Youssef[2,3], Moritz Schubotz[1,4], Avi Trost[5], Rajen Dey[6], Akiko Aizawa[7], and Bela Gipp[1]

[1] University of Wuppertal, Wuppertal, Germany, {greinerpetter,schubotz,gipp}@uni-wuppertal.de
[2] National Institute of Standards and Technology, Mission Viejo, CA, U.S.A., howard.cohl@nist.gov
[3] George Washington University, Washington, D.C., U.S.A, ayoussef@gwu.edu
[4] FIZ Karlsruhe, Berlin, Germany, moritz.schubotz@fiz-karlsruhe.de
[5] Brown University, Providence, RI, U.S.A., avitrost@gmail.com
[6] University of California Berkeley, Berkeley, CA, U.S.A., rajhataj@gmail.com
[7] National Institute of Informatics, Tokyo, Japan, aizawa@nii.ac.jp

Abstract. Digital mathematical libraries assemble the knowledge of years of mathematical research. Numerous disciplines (e.g., physics, engineering, pure and applied mathematics) rely heavily on compendia gathered findings. Likewise, modern research applications rely more and more on computational solutions, which are often calculated and verified by computer algebra systems. Hence, the correctness, accuracy, and reliability of both digital mathematical libraries and computer algebra systems is a crucial attribute for modern research. In this paper, we present a novel approach to verify a digital mathematical library and two computer algebra systems with one another by converting mathematical expressions from one system to the other. We use our previously developed conversion tool (referred to as LaCASt) to translate formulae from the NIST Digital Library of Mathematical Functions to the computer algebra systems `Maple` and `Mathematica`. The contributions of our presented work are as follows: (1) we present the most comprehensive verification of computer algebra systems and digital mathematical libraries with one another; (2) we significantly enhance the performance of the underlying translator in terms of coverage and accuracy; and (3) we provide open access to translations for `Maple` and `Mathematica` of the formulae in the NIST Digital Library of Mathematical Functions.

Keywords: Presentation to Computation, LaCASt, LaTeX, Semantic LaTeX, Computer Algebra Systems, Digital Mathematical Library

1 Introduction

Digital Mathematical Libraries (DML) gather the knowledge and results from thousands of years of mathematical research. Even though pure and applied mathematics are precise disciplines, gathering their knowledge bases over many years results in issues which every digital library shares: consistency, completeness, and accuracy. Likewise, Computer Algebra Systems (CAS)[8] play a crucial role in the modern era for pure and applied mathematics, and those fields which rely on them. CAS can be used to simplify, manipulate, compute, and visualize mathematical expressions. Accordingly, modern research regularly uses DML and CAS together. Nonetheless, DML [8, 19] and CAS [1, 12, 25] are not exempt from having bugs or errors. Durán et al. [12] even raised the rather dramatic question: "*can we trust in [CAS]?*"

Existing comprehensive DML, such as the Digital Library of Mathematical Functions (DLMF) [11], are consistently updated and frequently corrected with errata[9]. Although each chapter of the DLMF has been carefully written, edited, validated, and proofread over many years, errors still remain. Maintaining a DML, such as the DLMF, is a laborious process. Likewise, CAS are eminently complex systems, and in the case of commercial products, often similar to black boxes in which the magic (i.e., the computations) happens in opaque private code [12]. CAS, especially commercial products, are often exclusively tested internally during development.

An independent examination process can improve testing and increase trust in the systems and libraries. Hence, we want to elaborate on the following research question.

How can digital mathematical libraries and computer algebra systems be utilized to improve and verify one another?

Our initial approach for answering this question is inspired by our previous studies on translating DLMF equations to CAS [8]. In order to verify a translation tool from a specific LaTeX dialect to `Maple`[10]. , we performed *symbolic* and *numeric* evaluations on equations from the DLMF. Our approach presumes that a proven equation in a DML must be also valid in a CAS. In turn, a disparity in between the DML and CAS would lead to an issue in the translation process. However, assuming a correct translation, a disparity would also indicate an issue either in the DML source or the CAS implementation. In turn, we can take advantage of the same approach to improve and even verify DML with CAS and vice versa. Unfortunately, previous efforts to translate mathematical expressions from various formats, such as LaTeX [9, 19, 34], MathML [36], or OpenMath [22, 35], to CAS syntax have shown that the translation will be the most critical part of this verification approach.

[8] In the sequel, the acronyms CAS and DML are used, depending on the context, interchangeably with their plurals.

[9] https://dlmf.nist.gov/errata/ [accessed 09/01/2021]

[10] The mention of specific products, trademarks, or brand names is for purposes of identification only. Such mention is not to be interpreted in any way as an endorsement or certification of such products or brands by the National Institute of Standards and Technology, nor does it imply that the products so identified are necessarily the best available for the purpose. All trademarks mentioned herein belong to their respective owners.

In this paper, we elaborate on the feasibility and limitations of the translation approach from DML to CAS as a possible answer to our research question. We further focus on the DLMF as our DML and the two general-purpose CAS `Maple` and `Mathematica` for this first study. This relatively sharp limitation is necessary in order to analyze the capabilities of the underlying approach to verify commercial CAS and large DML. The DLMF uses semantic macros internally in order to disambiguate mathematical expressions [32, 40]. These macros help to mitigate the open issue of retrieving sufficient semantic information from a context to perform translations to formal languages [19, 36]. Further, the DLMF and general-purpose CAS have a relatively large overlap in coverage of special functions and orthogonal polynomials. Since many of those functions play a crucial role in a large variety of different research fields, we focus in this study mainly on these functions. Lastly, we will take our previously developed translation tool LaCASt [9, 19] as the baseline for translations from the DLMF to `Maple`. In this successor project, we focus on improving LaCASt to minimize the negative effect of wrong translations as much as possible for our study. In the future, other DML and CAS can be improved and verified following the same approach by using a different translation approach depending on the data of the DML, e.g., MathML [36] or OpenMath [22].

In particular, in this paper, we fix the majority of the remaining issues of LaCASt [8], which allows our tool to translate twice as many expressions from the DLMF to the CAS as before. Current extensions include the support for the mathematical operators: sum, product, limit, and integral, as well as overcoming semantic hurdles associated with Lagrange (prime) notations commonly used for differentiation. Further, we extend its support to include `Mathematica` using the freely available *Wolfram Engine for Developers* (WED)[11] (hereafter, with `Mathematica`, we refer to the WED). These improvements allow us to cover a larger portion of the DLMF, increase the reliability of the translations via LaCASt, and allow for comparisons between two major general-purpose CAS for the first time, namely `Maple` and `Mathematica`. Finally, we provide open access to all the results contained within this paper, including all translations of DLMF formulae, an endpoint to LaCASt[12], and the full source code of LaCASt[13].

The paper is structured as follows. Section 2 explains the data in the DLMF. Section 3 focus on the improvements of LaCASt that had been made to make the translation as comprehensive and reliable as possible for the upcoming evaluation. Section 4 explains the symbolic and numeric evaluation pipeline. Since Cohl et al. [8] only briefly sketched the approach of a numeric evaluation, we will provide an in-depth discussion of that process in Section 4. Subsequently, we analyze the results in Section 5. Finally, we conclude the findings and provide an outlook for upcoming projects in Section 6.

1.1 Related Work

Existing verification techniques for CAS often focus on specific subroutines or functions [6, 7, 13, 21, 25, 26, 30, 31], such as a specific theorems [28], differential

[11] https://www.wolfram.com/engine/ [accessed 09/01/2021]
[12] https://lacast.wmflabs.org/ [accessed 01/01/2022]
[13] https://github.com/ag-gipp/LaCASt [accessed 04/01/2022]

equations [23], or the implementation of the `math.h` library [29]. Most common are verification approaches that rely on intermediate verification languages [6, 21, 23, 25, 26], such as *Boogie* [2, 30] or *Why3* [5, 26], which, in turn, rely on proof assistants and theorem provers, such as *Coq* [4, 6], *Isabelle* [23, 33], or *HOL Light* [20, 21, 25]. Kaliszyk and Wiedijk [25] proposed on entire new CAS which is built on top of the proof assistant HOL Light so that each simplification step can be proven by the underlying architecture. Lewis and Wester [31] manually compared the symbolic computations on polynomials and matrices with seven CAS. Aguirregabiria et al. [1] suggested to teach students the known traps and difficulties with evaluations in CAS instead to reduce the overreliance on computational solutions.

Cohl et al. [8] developed the aforementioned translation tool LaCASt, which translates expressions from a semantically enhanced LaTeX dialect to `Maple`. By evaluating the performance and accuracy of the translations, we were able to discover a sign-error in one the DLMF's equations [8]. While the evaluation was not intended to verify the DLMF, the translations by the rule-based translator LaCASt provided sufficient robustness to identify issues in the underlying library. To the best of our knowledge, besides this related evaluation via LaCASt, there are no existing libraries or tools that allow for automatic verification of DML.

2 The DLMF dataset

In the modern era, most mathematical texts (handbooks, journal publications, magazines, monographs, treatises, proceedings, etc.) are written using the document preparation system LaTeX. However, the focus of LaTeX is for precise control of the rendering mechanics rather than for a semantic description of its content. In contrast, CAS syntax is coercively unambiguous in order to interpret the input correctly. Hence, a transformation tool from DML to CAS must disambiguate mathematical expressions. While there is an ongoing effort towards such a process [18, 27, 37, 38, 39, 41], there is no reliable tool available to disambiguate mathematics sufficiently to date.

The DLMF contains numerous relations between functions and many other properties. It is written in LaTeX but uses specific semantic macros when applicable [40]. These semantic macros represent a unique function or polynomial defined in the DLMF. Hence, the semantic LaTeX used in the DLMF is often unambiguous. For a successful evaluation via CAS, we also need to utilize all requirements of an equation, such as constraints, domains, or substitutions. The DLMF provides this additional data too and generally in a machine-readable form [40]. This data is accessible via the i-boxes (information boxes next to an equation marked with the icon ⓘ). If the information is not given in the attached i-box or the information is incorrect, the translation via LaCASt would fail. The i-boxes, however, do not contain information about branch cuts (see Section B) or constraints. Constraints are accessible if they are directly attached to an equation. If they appear in the text (or even a title), LaCASt cannot utilize them. The test dataset, we are using, was generated from DLMF Version 1.1.3 (2021-09-15) and contained 9,977 formulae with 1,505 defined symbols, 50,590 used symbols, 2,691 constraints, and 2,443 warnings for non-semantic expressions, i.e., expressions without semantic macros [40]. Note that the DLMF does not provide access to the underlying LaTeX source. Therefore, we added the source of every equation to our result dataset.

3 Semantic LaTeX to CAS translation

The aforementioned translator LaCASt was developed by Cohl and Greiner-Petter et al. [8, 9, 19]. They reported a coverage of 58.8% translations for a manually selected part of the DLMF to the CAS Maple. This version of LaCASt serves as a baseline for our improvements. In order to verify their translations, they used symbolic and numeric evaluations and reported a success rate of $\sim$16% for symbolic and $\sim$12% for numeric verifications.

Evaluating the baseline on the entire DLMF result in a coverage of only 31.6%. Hence, we first want to increase the coverage of LaCASt on the DLMF. To achieve this goal, we first increasing the number of translatable semantic macros by manually defining more translation patterns for special functions and orthogonal polynomials. For Maple, we increased the number from 201 to 261. For Mathematica, we define 279 new translation patterns which enables LaCASt to perform translations to Mathematica. Even though the DLMF uses 675 distinguished semantic macros, we cover $\sim$70% of all DLMF equations with our extended list of translation patterns (see Zipf's law for mathematical notations [17]). In addition, we implemented rules for translations that are applicable in the context of the DLMF, e.g., ignore ellipsis following floating-point values or `\choose` always refers to a binomial expression. Finally, we tackle the remaining issues outlined by Cohl et al. [8] which can be categorized into three groups: (i) expressions of which the arguments of operators are not clear, namely sums, products, integrals, and limits; (ii) expressions with prime symbols indicating differentiation; and (iii) expressions that contain ellipsis. While we solve some of the cases in Group (iii) by ignoring ellipsis following floating-point values, most of these cases remain unresolved. In the following, we elaborate our solutions for (i) in Section 3.1 and (ii) in Section 3.2.

3.1 Parse sums, products, integrals, and limits

Here we consider common notations for the sum, product, integral, and limit operators. For these operators, one may consider mathematically essential operator metadata (MEOM). For all these operators, the MEOM includes *argument(s)* and *bound variable(s)*. The operators act on the arguments, which are themselves functions of the bound variable(s). For sums and products, the bound variables are referred to as *indices*. The bound variables for integrals[14] are called *integration variables*. For limits, the bound variables are continuous variables (for limits of continuous functions) and indices (for limits of sequences). For integrals, MEOM include precise descriptions of regions of integration (e.g., piecewise continuous paths/intervals/regions). For limits, MEOM include limit points (e.g., points in $\mathbb{R}^n$ or $\mathbb{C}^n$ for $n \in \mathbb{N}$), as well as information related to whether the limit to the limit point is independent or dependent on the direction in which the limit is taken (e.g., one-sided limits).

[14] The notion of integrals includes: antiderivatives (indefinite integrals), definite integrals, contour integrals, multiple (surface, volume, etc.) integrals, Riemannian volume integrals, Riemann integrals, Lebesgue integrals, Cauchy principal value integrals, etc.

For a translation of mathematical expressions involving the LaTeX commands `\sum`, `\int`, `\prod`, and `\lim`, we must extract the MEOM. This is achieved by (a) determining the argument of the operator and (b) parsing corresponding subscripts, superscripts, and arguments. For integrals, the MEOM may be complicated, but certainly contains the argument (function which will be integrated), bound (integration) variable(s) and details related to the region of integration. Bound variable extraction is usually straightforward since it is usually contained within a differential expression (infinitesimal, pushforward, differential 1-form, exterior derivative, measure, etc.), e.g., $\mathrm{d}x$. Argument extraction is less straightforward since even though differential expressions are often given at the end of the argument, sometimes the differential expression appears in the numerator of a fraction (e.g., $\int \frac{f(x)\mathrm{d}x}{g(x)}$). In which case, the argument is everything to the right of the `\int` (neglecting its subscripts and superscripts) up to and including the fraction involving the differential expression (which may be replaced with 1). In cases where the differential expression is fully to the right of the argument, then it is a *termination symbol*. Note that some scientists use an alternate notation for integrals where the differential expression appears immediately to the right of the integral, e.g., $\int \mathrm{d}x f(x)$. However, this notation does not appear in the DLMF. If such notations are encountered, we follow the same approach that we used for sums, products, and limits (see Section 3.1).

Extraction of variables and corresponding MEOM The subscripts and superscripts of sums, products, limits, and integrals may be different for different notations and are therefore challenging to parse. For integrals, we extract the bound (integration) variable from the differential expression. For sums and products, the upper and lower bounds may appear in the subscript or superscript. Parsing subscripts is comparable with the problem of parsing constraints [8] (which are often not consistently formulated). We overcame this complexity by manually defining patterns of common constraints and refer to them as blueprints. This blueprint pattern approach allows LaCASt to identify the MEOM in the sub- and superscripts. A more detailed explanations with examples about the blueprints is available in the Appendix A.

Identification of operator arguments Once we have extracted the bound variable for sums, products, and limits, we need to determine the end of the argument. We analyzed all sums in the DLMF and developed a heuristic that covers all the formulae in the DLMF and potentially a large portion of general mathematics. Let x be the extracted bound variable. For sums, we consider a summand as a part of the argument if (I) it is the very first summand after the operation; or (II) x is an element of the current summand; or (III) x is an element of the following summand (subsequent to the current summand) and there is no termination symbol between the current summand and the summand which contains x with an equal or lower depth according to the parse tree (i.e., closer to the root). We consider a summand as a single logical construct since addition and subtraction are granted a lower operator precedence than multiplication in mathematical expressions. Similarly, parentheses are granted higher precedence and, thus, a sequence wrapped in parentheses is part of the argument if it obeys the rules (I-III). Summands, and such sequences, are always entirely part of sums, products, and limits or entirely not.

A termination symbol always marks the end of the argument list. Termination symbols are relation symbols, e.g., $=$, $\neq$, $\leq$, closing parentheses or brackets, e.g.,),], or $>$, and other operators with MEOMs, if and only if, they define the same bound variable. If x is part of a subsequent operation, then the following operator is considered as part of the argument (as in (II)). However, a special condition for termination symbols is that it is only a termination symbol for the current chain of arguments. Consider a sum over a fraction of sums. In that case, we may reach a termination symbol within the fraction. However, the termination symbol would be deeper inside the parse tree as compared to the current list of arguments. Hence, we used the depth to determine if a termination symbol should be recognized or not. Consider an unusual notation with the binomial coefficient as an example

$$\sum_{k=0}^{n}\binom{n}{k} = \sum_{k=0}^{n}\frac{\prod_{m=1}^{n} m}{\prod_{m=1}^{k} m \prod_{m=1}^{n-k} m}. \tag{1}$$

This equation contains two termination symbols, marked red and green. The red termination symbol $=$ is obviously for the first sum on the left-hand side of the equation. The green termination symbol $\prod$ terminates the product to the left because both products run over the same bound variable m. In addition, none of the other $=$ signs are termination symbols for the sum on the right-hand side of the equation because they are deeper in the parse tree and thus do not terminate the sum.

Note that `varN` in the blueprints also matches multiple bound variable, e.g., $\sum_{m,k\in A}$. In such cases, x from above is a list of bound variables and a summand is part of the argument if one of the elements of x is within this summand. Due to the translation, the operation will be split into two preceding operations, i.e., $\sum_{m,k\in A}$ becomes $\sum_{m\in A}\sum_{k\in A}$. Figure 1 shows the extracted arguments for some example sums. The same rules apply for extraction of arguments for products and limits.

$$\boxed{\sum_{n=1}^{N} c}+2$$

$$\boxed{\sum_{n=1}^{N} c+\frac{c}{n}}$$

$$\boxed{\sum_{n=1}^{N} c+n^2}+N$$

$$\boxed{\sum_{n=1}^{N} n}+\boxed{\sum_{k=1}^{N} k}$$

$$\boxed{\sum_{n=1}^{N} n+\boxed{\sum_{k=1}^{n} k}}$$

$$\boxed{\sum_{n=1}^{N} c+\boxed{\sum_{k=1}^{N} k}+n}$$

Fig. 1: Example argument identifications for sums.

3.2 Lagrange's notation for differentiation and derivatives

Another remaining issue is the Lagrange (prime) notation for differentiation, since it does not outwardly provide sufficient semantic information. This notation presents two challenges. First, we do not know with respect to which variable the differentiation should be performed. Consider for example the Hurwitz zeta function $\zeta(s,a)$ [11, §25.11]. In the case of a differentiation $\zeta'(s,a)$, it is not clear if the function should be differentiated with respect to s or a. To remedy this issue, we analyzed all formulae in the DLMF which use prime notations and determined which variables (slots) for which functions represent the variables of the differentiation. Based on our analysis, we extended the translation patterns by meta information for semantic macros according to the slot of differentiation. For instance, in the case of the Hurwitz zeta function, the first slot is the slot for prime differentiation, i.e., $\zeta'(s,a) = \frac{\mathrm{d}}{\mathrm{d}s}\zeta(s,a)$. The identified variables of differentiations for the special functions in the DLMF can be considered to be the standard slots of differentiations, e.g., in other DML, $\zeta'(s,a)$ most likely refers to $\frac{\mathrm{d}}{\mathrm{d}s}\zeta(s,a)$.

The second challenge occurs if the slot of differentiation contains complex expressions rather than single symbols, e.g., $\zeta'(s^2,a)$. In this case, $\zeta'(s^2,a) = \frac{\mathrm{d}}{\mathrm{d}(s^2)}\zeta(s^2,a)$ instead of $\frac{\mathrm{d}}{\mathrm{d}s}\zeta(s^2,a)$. Since CAS often do not support derivatives with respect to complex expressions, we use the inbuilt substitution functions[15] in the CAS to overcome this issue. To do so, we use a temporary variable `temp` for the substitution. CAS perform substitutions from the inside to the outside. Hence, we can use the same temporary variable `temp` even for nested substitutions. Table 1 shows the translation performed for $\zeta'(s^2,a)$. CAS may provide optional arguments to calculate the derivatives for certain special functions, e.g., `Zeta(n,z,a)` in `Maple` for the n-th derivative of the Hurwitz zeta function. However, this shorthand notation is generally not supported (e.g., `Mathematica` does not define such an optional parameter). Our substitution approach is more lengthy but also more reliable. Unfortunately, lengthy expressions generally harm the performance of CAS, especially for symbolic manipulations. Hence, we have a genuine interest in keeping translations short, straightforward and readable. Thus, the substitution translation pattern is only triggered if the variable of differentiation is not a single identifier. Note that this substitution only triggers on semantic macros. Generic functions, including prime notations, are still skipped.

A related problem to MEOM of sums, products, integrals, limits, and differentiations are the notations of derivatives. The semantic macro for derivatives `\deriv{w}{x}` (rendered as $\frac{\mathrm{d}w}{\mathrm{d}x}$) is often used with an empty first argument to render the function behind the derivative notation, e.g., `\deriv{}{x}\sin@{x}` for $\frac{\mathrm{d}}{\mathrm{d}x}\sin x$. This leads to the same problem we faced above for identifying MEOMs. In this case, we use the same heuristic as we did for sums, products, and limits. Note that derivatives may be written following the function argument, e.g., $\sin(x)\frac{\mathrm{d}}{\mathrm{d}x}$. If we are unable to identify any following summand that contains the variable of differentiation before we reach a termination symbol, we look for arguments prior to the derivative according to the heuristic (I-III).

Table 1: Example translations for the prime derivative of the Hurwitz zeta function with respect to s^2.

System	$\zeta'(s^2,a)$
DLMF	`\Hurwitzzeta'@{s^2}{a}`
`Maple`	`subs(temp=(s)^(2),diff( Zeta(0,temp,a),temp$(1)))`
`Mathematica`	`D[HurwitzZeta[temp,a], {temp,1}]/.temp->(s)^(2)`

[15] Note that `Maple` also support an evaluation substitution via the two-argument `eval` function. Since our substitution only triggers on semantic macros, we only use `subs` if the function is defined in `Maple`. In turn, as far as we know, there is no practical difference between `subs` and the two-argument `eval` in our case.

Wronskians With the support of prime differentiation described above, we are also able to translate the Wronskian [11, (1.13.4)] to `Maple` and `Mathematica`. A translation requires one to identify the variable of differentiation from the elements of the Wronskian, e.g., z for $\mathscr{W}\{\mathrm{Ai}(z),\mathrm{Bi}(z)\}$ from [11, (9.2.7)]. We analyzed all Wronskians in the DLMF and discovered that most Wronskians have a special function in its argument—such as the example above. Hence, we can use our previously inserted metadata information about the slots of differentiation to extract the variable of differentiation from the semantic macros. If the semantic macro argument is a complex expression, we search for the identifier in the arguments that appear in both elements of the Wronskian. For example, in $\mathscr{W}\{\mathrm{Ai}(z^a),\zeta(z^2,a)\}$, we extract z as the variable since it is the only identifier that appears in the arguments z^a and z^2 of the elements. This approach is also used when there is no semantic macro involved, i.e., from $\mathscr{W}\{z^a,z^2\}$ we extract z as well. If LaCASt extracts multiple candidates or none, it throws a translation exception.

4 Evaluation of the DLMF using CAS

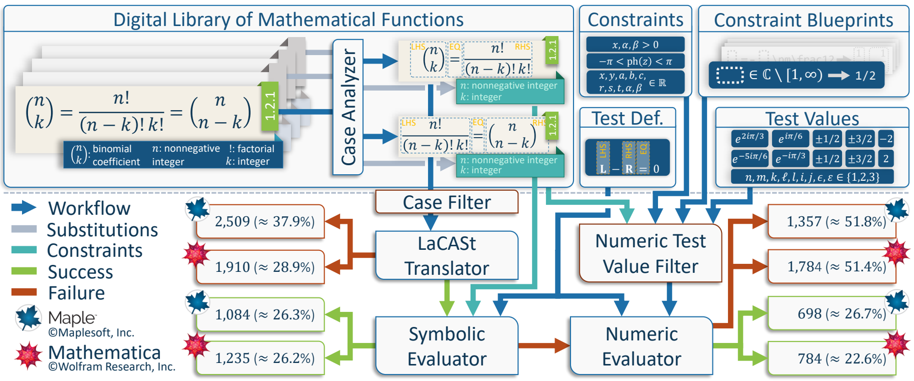

Digital Library of Mathematical Functions
$\binom{n}{k} = \frac{n!}{(n-k)!\,k!} = \binom{n}{n-k}$
1.2.1
$\binom{n}{k}$: binomial coefficient
n: nonnegative integer
!: factorial
k: integer
Case Analyzer
LHS
EQ
RHS
$\binom{n}{k} = \frac{n!}{(n-k)!\,k!}$
n: nonnegative integer
k: integer
$\frac{n!}{(n-k)!\,k!} = \binom{n}{n-k}$
Constraints
$x,\alpha,\beta > 0$
$-\pi < \mathrm{ph}(z) < \pi$
$x,y,a,b,c,r,s,t,\alpha,\beta \in \mathbb{R}$
Constraint Blueprints
$\in \mathbb{C} \setminus [1,\infty) \rightarrow 1/2$
Test Def.
$\mathbf{L} - \mathbf{R} = 0$
Test Values
$e^{2i\pi/3}$ $e^{i\pi/6}$ $\pm 1/2$ $\pm 3/2$ -2
$e^{-5i\pi/6}$ $e^{-i\pi/3}$ $\pm 1/2$ $\pm 3/2$ 2
$n,m,k,\ell,l,i,j,\epsilon,\varepsilon \in \{1,2,3\}$
Workflow
Substitutions
Constraints
Success
Failure
Maple ©Maplesoft, Inc.
Mathematica ©Wolfram Research, Inc.
Case Filter
LaCASt Translator
Symbolic Evaluator
Numeric Test Value Filter
Numeric Evaluator
2,509 (≈ 37.9%)
1,910 (≈ 28.9%)
1,084 (≈ 26.3%)
1,235 (≈ 26.2%)
1,357 (≈ 51.8%)
1,784 (≈ 51.4%)
698 (≈ 26.7%)
784 (≈ 22.6%)

Fig. 2: The workflow of the evaluation engine and the overall results. Errors and abortions are not included. The generated dataset contains 9,977 equations. In total, the case analyzer splits the data into 10,930 cases of which 4,307 cases were filtered. This sums up to a set of 6,623 test cases in total.

For evaluating the DLMF with `Maple` and `Mathematica`, we follow the same approach as demonstrated in [8], i.e., we symbolically and numerically verify the equations in the DLMF with CAS. If a verification fails, symbolically and numerically, we identified an issue either in the DLMF, the CAS, or the verification pipeline. Note that an issue does not necessarily represent errors/bugs in the DLMF, CAS, or LaCASt (see the discussion about branch cuts in Section B). Figure 2 illustrates the pipeline of the evaluation engine. First, we analyze every equation in the DLMF (hereafter referred to as test cases). A case analyzer splits multiple relations in a single line into multiple test cases. Note that only the adjacent relations are considered, i.e., with $f(z)=g(z)=h(z)$, we generate two test cases $f(z)=g(z)$ and $g(z)=h(z)$ but not $f(z)=h(z)$. In addition, expressions with $\pm$ and $\mp$ are split accordingly, e.g., $i^{\pm i}=\mathrm{e}^{\mp\pi/2}$ [11, (4.4.12)] is split into $i^{+i}=\mathrm{e}^{-\pi/2}$ and $i^{-i}=\mathrm{e}^{+\pi/2}$. The analyzer utilizes the attached additional information in each line, i.e., the URL in the DLMF, the used and defined symbols, and the constraints. If a used symbol is defined elsewhere in the DLMF, it performs substitutions. For example, the multi-equation [11, (9.6.2)] is split into six test cases and every ζ is replaced by $\frac{2}{3}z^{3/2}$ as defined in [11, (9.6.1)]. The substitution is performed on the parse tree of expressions [19]. A definition is only considered as such, if the defining symbol is identical to the equation's left-hand side. That means, $z = (\frac{3}{2}\zeta)^{3/2}$ [11, (9.6.10)] is not considered as a definition for ζ. Further, semantic macros are never substituted by their definitions. Translations for semantic macros are exclusively defined by the authors. For example, the equation [11, (11.5.2)] contains the Struve $\mathbf{K}_\nu(z)$ function. Since `Mathematica` does not contain this function, we defined an alternative translation to its definition $\mathbf{H}_\nu(z) - Y_\nu(z)$ in [11, (11.2.5)] with the Struve function $\mathbf{H}_\nu(z)$ and the Bessel function of the second kind $Y_\nu(z)$, because both of these functions are supported by `Mathematica`. The second entry in Table 3 in the Appendix D shows the translation for this test case.

Next, the analyzer checks for additional constraints defined by the used symbols recursively. The mentioned Struve $\mathbf{K}_\nu(z)$ test case [11, (11.5.2)] contains the Gamma function. Since the definition of the Gamma function [11, (5.2.1)] has a constraint $\Re z > 0$, the numeric evaluation must respect this constraint too. For this purpose, the case analyzer first tries to link the variables in constraints to the arguments of the functions. For example, the constraint $\Re z > 0$ sets a constraint for the first argument z of the Gamma function. Next, we check all arguments in the actual test case at the same position. The test case contains $\Gamma(\nu + 1/2)$. In turn, the variable z in the constraint of the definition of the Gamma function $\Re z > 0$ is replaced by the actual argument used in the test case. This adds the constraint $\Re(\nu + 1/2) > 0$ to the test case. This process is performed recursively. If a constraint does not contain any variable that is used in the final test case, the constraint is dropped.

In total, the case analyzer would identify four additional constraints for the test case [11, (11.5.2)]. Table 3 in the Appendix D shows the applied constraints (including the directly attached constraint $\Re z > 0$ and the manually defined global constraints from Figure 3). Note that the constraints may contain variables that do not appear in the actual test case, such as $\Re \nu + k + 1 > 0$. Such constraints do not have any effect on the evaluation because if a constraint cannot be computed to true or false, the constraint is ignored. Unfortunately, this recursive loading of additional constraints may generate impossible conditions in certain cases, such as $|\Gamma(iy)|$ [11, (5.4.3)]. There are no valid real values of y such that $\Re(iy) > 0$. In turn, every test value would be filtered out, and the numeric evaluation would not verify the equation. However, such cases are the minority and we were able to increase the number of correct evaluations with this feature.

To avoid a large portion of incorrect calculations, the analyzer filters the dataset before translating the test cases. We apply two filter rules to the case analyzer. First, we filter expressions that do not contain any semantic macros. Due to the limitations of LACAST, these expressions most likely result in wrong translations. Further, it filters out several meaningless expressions that are not verifiable, such as $z = x$ in [11, (4.2.4)]. The result dataset flag these cases with '*Skipped - no semantic math*'. Note that the result dataset still contains the translations for these cases to provide a complete picture of the DLMF. Second, we filter expressions that contain ellipsis[16] (e.g., `\cdots`), approximations, and asymptotics (e.g., $\mathcal{O}(z^2)$) since those expressions cannot be evaluated with the proposed approach. Further, a definition is skipped if it is not a definition of a semantic macro, such as [11, (2.3.13)], because definitions without an appropriate counterpart in the CAS are meaningless to evaluate. Definitions of semantic macros, on the other hand, are of special interest and remain in the test set since they allow us to test if a function in the CAS obeys the actual mathematical definition in the DLMF. If the case analyzer (see Figure 2) is unable to detect a relation, i.e., split an expression on $<$, $\leq$, $\geq$, $>$, $=$, or $\neq$, the line in the dataset is also skipped because the evaluation approach relies on relations to test. After splitting multi-equations (e.g., $\pm$, $\mp$, $a=b=c$), filtering out all non-semantic expressions, non-semantic macro definitions, ellipsis, approximations, and asymptotics, we end up with 6,623 test cases in total from the entire DLMF.

After generating the test case with all constraints, we translate the expression to the CAS representation. Every successfully translated test case is then symbolically verified, i.e., the CAS tries to simplify the difference of an equation to zero. Non-equation relations simplifies to Booleans. Non-simplified expressions are verified numerically for manually defined test values, i.e., we calculate actual numeric values for both sides of an equation and check their equivalence.

4.1 Symbolic Evaluation

The symbolic evaluation was performed for `Maple` as in [8]. However, we use the newer version `Maple` 2020. Another feature we added to LaCASt is the support of packages in `Maple`. Some functions are only available in modules (packages) that must be preloaded, such as `QPochhammer` in the package `QDifferenceEquations`[17]. The general `simplify` method in `Maple` does not cover q-hypergeometric functions. Hence, whenever LaCASt loads functions from the q-hyper-geometric package, the better performing `QSimplify` method is used. With the WED and the new support for `Mathematica` in LaCASt, we perform the symbolic and numeric tests for `Mathematica` as well. The symbolic evaluation in `Mathematica` relies on the full simplification[18]. For `Maple` and `Mathematica`, we defined the global assumptions $x,y \in \mathbb{R}$ and $k,n,m \in \mathbb{N}$. Constraints of test cases are added to their assumptions to support simplification. Adding more global assumptions for symbolic computation generally harms the performance since CAS internally uses assumptions for simplifications. It turned out that by adding more custom assumptions, the number of successfully simplified expressions decreases.

4.2 Numerical Evaluation

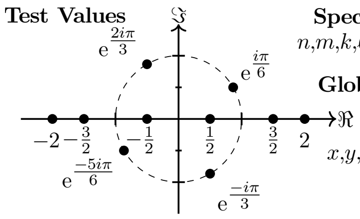
Defining an accurate test set of values to analyze an equivalence can be an arbitrarily complex process. It would make sense that every expression is tested on specific values according to the containing functions. However, this laborious process is not suitable

[16] Note that we filter out ellipsis (e.g., `\cdots`) but not single dots (e.g., `\cdot`).

[17] https://jp.maplesoft.com/support/help/Maple/view.aspx?path=QDifferenceEquations/QPochhammer [accessed 09/01/2021]

[18] https://reference.wolfram.com/language/ref/FullSimplify.html [accessed 09/01/2021]

for evaluating the entire DML and CAS. It makes more sense to develop a general set of test values that (i) generally covers interesting domains and (ii) avoid singularities, branch cuts, and similar problematic regions. Considering these two attributes, we come up with the ten test points illustrated in Figure 3. It contains four complex values on the unit circle and six points on the real axis. The test values cover the general area of interest (complex values in all four quadrants, negative and positive real values) and avoid the typical singularities at $\{0,\pm1,\pm i\}$. In addition, several variables are tied to specific values for entire sections. Hence, we applied additional global constraints to the test cases.



Special Test Values
$n,m,k,\ell,l,i,j,\epsilon,\varepsilon \in \{1,2,3\}$

Global Constraints
$x,\alpha,\beta > 0$
$-\pi < \mathrm{ph}(z) < \pi$
$x,y,a,b,c,r,s,t,\alpha,\beta \in \mathbb{R}$

Fig. 3: The ten numeric test values in the complex plane for general variables. The dashed line represents the unit circle $|z| = 1$. At the right, we show the set of values for special variable values and general global constraints. On the right, i is referring to a generic variable and not to the imaginary unit.

The numeric evaluation engine heavily relies on the performance of extracting free variables from an expression. Unfortunately, the inbuilt functions in CAS, if available, are not very reliable. As the authors explained in [8], a custom algorithm within `Maple` was necessary to extract identifiers. `Mathematica` has the undocumented function ``Reduce`FreeVariables`` for this purpose. However, both systems, the custom solution in `Maple` and the inbuilt `Mathematica` function, have problems distinguishing free variables of entire expressions from the bound variables in MEOMs, e.g., integration and continuous variables. `Mathematica` sometimes does not extract a variable but returns the unevaluated input instead. We regularly faced this issue for integrals. However, we discovered one example without integrals. For `EulerE[n,0]` from [11, (24.4.26)], we expected to extract $\{n\}$ as the set of free variables but instead received a set of the unevaluated expression itself {`EulerE[n,0]`}[19]. Since the extended version of LaCASt handles operators, including bound variables of MEOMs, we drop the use of internal methods in CAS and extend LaCASt to extract identifiers from an expression. During a translation process, LaCASt tags every single identifier as a variable, as long as it is not an element of a MEOM. This simple approach proves to be very efficient since it is implemented alongside the translation process itself and is already more powerful as compared to the existing inbuilt CAS solutions. We defined subscripts of identifiers as a part of the identifier, e.g., z_1 and z_2 are extracted as variables from z_1+z_2 rather than z.

The general pipeline for a numeric evaluation works as follows. First, we replace all substitutions and extract the variables from the left- and right-hand sides of the test expression via LaCASt. For the previously mentioned example of the Struve function [11, (11.5.2)], LaCASt identifies two variables in the expression, ν and z. According to the values in Figure 3, ν and z are set to the general ten values. A numeric test contains every combination of test values for all variables. Hence, we generate 100 test calculations for [11, (11.5.2)]. Afterward, we filter the test values

[19] The bug was reported to and confirmed by Wolfram Research Version 12.0.

that violate the attached constraints. In the case of the Struve function, we end up with 25 test cases.

In addition, we apply a limit of 300 calculations for each test case and abort a computation after 30 seconds due to computational limitations. If the test case generates more than 300 test values, only the first 300 are used. Finally, we calculate the result for every remaining test value, i.e., we replace every variable by their value and calculate the result. The replacement is done by `Mathematica`'s `ReplaceAll` method because the more appropriate method `With`, for unknown reasons, does not always replace all variables by their values. We wrap test expressions in `Normal` for numeric evaluations to avoid conditional expressions, which may cause incorrect calculations (see Section 5.1 for a more detailed discussion of conditional outputs). After replacing variables by their values, we trigger numeric computation. If the absolute value of the result (i.e., the difference between left- and right-hand side of the equation) is below the defined threshold of 0.001 or true (in the case of inequalities), the test calculation is considered successful. A numeric test case is only considered successful if and only if every test calculation was successful. If a numeric test case fails, we store the information on which values it failed and how many of these were successful.

5 Results

The translations to `Maple` and `Mathematica`, the symbolic results, the numeric computations, and an overview PDF of the reported bugs to `Mathematica` are available online on our demopage. In the following, we mainly focus on `Mathematica` because of page limitations and because `Maple` has been investigated more closely by [8]. The results for `Maple` are also available online. Compared to the baseline ($\approx$31%), our improvements doubled the amount translations ($\approx$62%) for `Maple` and reach $\approx$71% for `Mathematica`. The majority of expressions that cannot be translated contain macros that have no adequate translation pattern to the CAS, such as the macros for interval Weierstrass lattice roots [11, §23.3(i)] and the multivariate hypergeometric function [11, (19.16.9)]. Other errors (6% for `Maple` and `Mathematica`) occur for several reasons. For example, out of the 418 errors in translations to `Mathematica`, 130 caused an error because the MEOM of an operator could not be extracted, 86 contained prime notations that do not refer to differentiations, 92 failed because of the underlying LaTeX parser [39], and in 46 cases, the arguments of a DLMF macro could not be extracted.

Out of 4,713 translated expressions, 1,235 (26.2%) were successfully simplified by `Mathematica` (1,084 of 4,114 or 26.3% in `Maple`). For `Mathematica`, we also count results that are equal to 0 under certain conditions as successful (called `ConditionalExpression`). We identified 65 of these conditional results: 15 of the conditions are equal to constraints that were provided in the surrounding text but not in the info box of the DLMF equation; 30 were produced due to branch cut issues (see Section B); and 20 were the same as attached in the DLMF but reformulated, e.g., $z\in\mathbb{C}\backslash(1,\infty)$ from [11, (25.12.2)] was reformulated to $\Im z\neq 0\vee\Re z<1$. The remaining translated but not symbolically verified expressions were numerically evaluated for the test values in Figure 3. For the 3,474 cases, 784 (22.6%) were successfully verified numerically by `Mathematica` (698 of 2,618 or 26.7% by `Maple`[20]). For 1,784 the numeric evaluation failed. In the evaluation process, 655 computations timed out and 180 failed due to errors in `Mathematica`. Of the 1,784 failed cases, 691 failed partially, i.e., there was at least one successful calculation among the tested values. For 1,091 all test values failed. Table 3 in the Appendix D shows the results for three sample test cases. The first case is a false positive evaluation because of a wrong translation. The second case is valid, but the numeric evaluation failed due to a bug in Mathematica (see next subsection). The last example is valid and was verified numerically but was too complex for symbolic verifications.

5.1 Error Analysis

The numeric tests' performance strongly depends on the correct attached and utilized information. The first example in Table 3 in the Appendix D illustrates the difficulty of the task on a relatively easy case. Here, the argument of f was not explicitly given, such as in $f(x)$. Hence, LACASt translated f as a variable. Unfortunately, this resulted in a false verification symbolically and numerically. This type of error mostly appears in the first three chapters of the DLMF because they use generic functions frequently. We hoped to skip such cases by filtering expressions without semantic macros. Unfortunately, this derivative notation uses the semantic macro `deriv`. In the future, we filter expressions that contain semantic macros that are not linked to a special function or orthogonal polynomial.

As an attempt to investigate the reliability of the numeric test pipeline, we can run numeric evaluations on symbolically verified test cases. Since `Mathematica` already approved a translation symbolically, the numeric test should be successful if the pipeline is reliable. Of the 1,235 symbolically successful tests, only 94 (7.6%) failed numerically. None of the failed test cases failed entirely, i.e., for every test case, at least one test value was verified. Manually investigating the failed cases reveal 74 cases that failed due to an `Indeterminate` response from `Mathematica` and 5 returned `infinity`, which clearly indicates that the tested numeric values were invalid, e.g., due to testing on singularities. Of the remaining 15 cases, two were identical: [11, (15.9.2)] and [11, (18.5.9)]. This reduces the remaining failed cases to 14. We evaluated invalid values for 12 of these because the constraints for the values were given in the surrounding text but not in the info boxes. The remaining 2 cases revealed a bug in `Mathematica` regarding conditional outputs (see below). The results indicate that the numeric test pipeline is reliable, at least for relatively simple cases that were previously symbolically verified. The main reason for the high number of failed numerical cases in the entire DLMF (1,784) are due to missing constraints in the i-boxes and branch cut issues (see Section B in the Appendix), i.e., we evaluated expressions on invalid values.

Bug reports `Mathematica` has trouble with certain integrals, which, by default, generate conditional outputs if applicable. With the method `Normal`, we can suppress

[20] Due to computational issues, 120 cases must have been skipped manually. 292 cases resulted in an error during symbolic verification and, therefore, were skipped also for numeric evaluations. Considering these skipped cases as failures, decreases the numerically verified cases to 23% in `Maple`.

conditional outputs. However, it only hides the condition rather than evaluating the expression to a non-conditional output. For example, integral expressions in [11, (10.9.1)] are automatically evaluated to the Bessel function $J_0(|z|)$ for the condition[21] $z\in\mathbb{R}$ rather than $J_0(z)$ for all $z\in\mathbb{C}$. Setting the `GenerateConditions`[22] option to `None` does not change the output. `Normal` only hides $z\in\mathbb{R}$ but still returns $J_0(|z|)$. To fix this issue, for example in (10.9.1) and (10.9.4), we are forced to set `GenerateConditions` to false.

Setting `GenerateConditions` to false, on the other hand, reveals severe errors in several other cases. Consider $\int_z^\infty t^{-1}\mathrm{e}^{-t}\mathrm{d}t$ [11, (8.4.4)], which gets evaluated to $\Gamma(0,z)$ but (condition) for $\Re z>0\wedge\Im z=0$. With `GenerateConditions` set to false, the integral incorrectly evaluates to $\Gamma(0,z)+\ln(z)$. This happened with the 2 cases mentioned above. With the same setting, the difference of the left- and right-hand sides of [11, (10.43.8)] is evaluated to 0.398942 for $x,\nu=1.5$. If we evaluate the same expression on x, $\nu=\frac{3}{2}$ the result is `Indeterminate` due to `infinity`. For this issue, one may use `NIntegrate` rather than `Integrate` to compute the integral. However, evaluating via `NIntegrate` decreases the number of successful numeric evaluations in general. We have revealed errors with conditional outputs in (8.4.4), (10.22.39), (10.43.8-10), and (11.5.2) (in [11]). In addition, we identified one critical error in `Mathematica`. For [11, (18.17.47)], WED (`Mathematica`'s kernel) ran into a *segmentation fault (core dumped)* for $n>1$. The kernel of the full version of `Mathematica` gracefully died without returning an output[23].

Besides `Mathematica`, we also identified several issues in the DLMF. None of the newly identified issues were critical, such as the reported sign error from the previous project [8], but generally refer to missing or wrong attached semantic information. With the generated results, we can effectively fix these errors and further semantically enhance the DLMF. For example, some definitions are not marked as such, e.g., $Q(z)=\int_0^\infty \mathrm{e}^{-zt}q(t)\mathrm{d}t$ [11, (2.4.2)]. In [11, (10.24.4)], ν must be a real value but was linked to a *complex parameter* and x should be positive real. An entire group of cases [11, (10.19.10-11)] also discovered the incorrect use of semantic macros. In these formulae, $P_k(a)$ and $Q_k(a)$ are defined but had been incorrectly marked up as Legendre functions going all the way back to DLMF Version 1.0.0 (May 7, 2010). In some cases, equations are mistakenly marked as definitions, e.g., [11, (9.10.10)] and [11, (9.13.1)] are annotated as local definitions of n. We also identified an error in LaCASt, which incorrectly translated the exponential integrals $E_1(z)$, $\mathrm{Ei}(x)$ and $\mathrm{Ein}(z)$ (defined in [11, §6.2(i)]). A more explanatory overview of discovered, reported, and fixed issues in the DLMF, `Mathematica`, and `Maple` is provided in the Appendix C.

6 Conclusion

We have presented a novel approach to verify the theoretical digital mathematical library DLMF with the power of two major general-purpose computer algebra systems `Maple` and `Mathematica`. With LaCASt, we transformed the semantically enhanced

[21] $J_0(x)$ with $x\in\mathbb{R}$ is even. Hence, $J_0(|z|)$ is correct under the given condition.

[22] https://reference.wolfram.com/language/ref/GenerateConditions.html [accessed 09/01/2021]

[23] All errors were reported to and partially confirmed by Wolfram Research. See Appendix C for more information.

LaTeX expressions from the DLMF to each CAS. Afterward, we symbolically and numerically evaluated the DLMF expressions in each CAS. Our results are auspicious and provide useful information to maintain and extend the DLMF efficiently. We further identified several errors in `Mathematica`, `Maple` [8], the DLMF, and the transformation tool LaCASt, proving the profit of the presented verification approach. Further, we provide open access to all results, including translations and evaluations[24]. and to the source code of LaCASt[25].

The presented results show a promising step towards an answer for our initial research question. By translating an equation from a DML to a CAS, automatic verifications of that equation in the CAS allows us to detect issues in either the DML source or the CAS implementation. Each analyzed failed verification successively improves the DML or the CAS. Further, analyzing a large number of equations from the DML may be used to finally verify a CAS. In addition, the approach can be extended to cover other DML and CAS by exploiting different translation approaches, e.g., via MathML [36] or OpenMath [22].

Nonetheless, the analysis of the results, especially for an entire DML, is cumbersome. Minor missing semantic information, e.g., a missing constraint or not respected branch cut positions, leads to a relatively large number of false positives, i.e., unverified expressions correct in the DML and the CAS. This makes a generalization of the approach challenging because all semantics of an equation must be taken into account for a trustworthy evaluation. Furthermore, evaluating equations on a small number of discrete values will never provide sufficient confidence to verify a formula, which leads to an unpredictable number of true negatives, i.e., erroneous equations that pass all tests. A more sophisticated selection of critical values or other numeric tools with automatic results verification (such as variants of Newton's interval method) potentially mitigates this issue in the future. After all, we conclude that the approach provides valuable information to complement, improve, and maintain the DLMF, `Maple`, and `Mathematica`. A trustworthy verification, on the other hand, might be out of reach.

6.1 Future Work

The resulting dataset provides valuable information about the differences between CAS and the DLMF. These differences had not been largely studied in the past and are worthy of analysis. Especially a comprehensive and machine-readable list of branch cut positioning in different systems is a desired goal [10]. Hence, we will continue to work closely together with the editors of the DLMF to improve further and expand the available information on the DLMF. Finally, the numeric evaluation approach would benefit from test values dependent on the actual functions involved. For example, the current layout of the test values was designed to avoid problematic regions, such as branch cuts. However, for identifying differences in the DLMF and CAS, especially for analyzing the positioning of branch cuts, an automatic evaluation of these particular values would be very beneficial and can be used to collect a comprehensive, inter-system library of branch cuts. Therefore, we will further study the possibility of linking semantic macros with numeric regions of interest.

[24] https://lacast.wmflabs.org [accessed 01/01/2022]
[25] https://github.com/ag-gipp/LaCASt [accessed 04/01/2022]

Acknowledgements

We thank Jürgen Gerhard from Maplesoft for providing access and support for Maple. We also thank the DLMF editors for their assistance and support. This work was supported by the German Research Foundation (DFG grant no.: GI 1259/1) and the German Academic Exchange Service (DAAD grant no.: 57515245).

References

[1] Juan M. Aguirregabiria, Anibal M. Hernández, and Martin Rivas. “Are We Careful Enough when Using Computer Algebra?” In: *Computers in Physics* 8.1 (1994), pp. 56–61. DOI: 10.1063/1.4823260.

[2] Mike Barnett et al. “Boogie: A Modular Reusable Verifier for Object-Oriented Programs”. In: *Formal Methods for Components and Objects*. Springer Berlin Heidelberg, 2006, pp. 364–387. DOI: 10.1007/11804192_17.

[3] Eric Temple Bell. “Exponential Polynomials”. In: *The Annals of Mathematics* 35.2 (Apr. 1934), p. 258. ISSN: 0003486X. DOI: 10.2307/1968431. JSTOR: 1968431.

[4] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq´Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer Berlin Heidelberg, 2004. ISBN: 978-3-642-05880-6.

[5] François Bobot et al. “Why3: Shepherd Your Herd of Provers”. In: *Boogie 2011: First International Workshop on Intermediate Verification Languages* (May 2011), pp. 53–64. URL: https://hal.inria.fr/hal-00790310/document.

[6] Sylvain Boulmé et al. “On the way to certify Computer Algebra Systems”. In: *Electronic Notes in Theoretical Computer Science* 23.3 (1999). CALCULEMUS 99, Systems for Integrated Computation and Deduction (associated to FLoC’99, the 1999 Federated Logic Conference), pp. 370–385. ISSN: 1571-0661. DOI: 10.1016/S1571-0661(05)80609-7.

[7] Jacques Carette and Michael Kucera. “Partial evaluation of Maple”. In: *Science of Computer Programming* 76.6 (June 2011), pp. 469–491. DOI: 10.1016/j.scico.2010.12.001.

[8] Howard S. Cohl, André Greiner-Petter, and Moritz Schubotz. “Automated Symbolic and Numerical Testing of DLMF Formulae Using Computer Algebra Systems”. In: *Intelligent Computer Mathematics CICM*. Vol. 11006. Springer, 2018, pp. 39–52. DOI: 10.1007/978-3-319-96812-4_4.

[9] Howard S. Cohl et al. “Semantic Preserving Bijective Mappings of Mathematical Formulae Between Document Preparation Systems and Computer Algebra Systems”. In: *Intelligent Computer Mathematics CICM*. Springer, 2017, pp. 115–131. DOI: 10.1007/978-3-319-62075-6_9.

[10] Robert M. Corless et al. “"According to Abramowitz and Stegun" or arccoth needn’t be uncouth”. In: *SIGSAM Bulletin* 34.2 (2000), pp. 58–65. DOI: 10.1145/362001.362023.

[11] DLMF. *NIST Digital Library of Mathematical Functions*. https://dlmf.nist.gov/, Release 1.1.4 of 2022-01-15. F. W. J. Olver, A. B. Olde Daalhuis, D. W. Lozier, B. I. Schneider, R. F. Boisvert, C. W. Clark, B. R. Miller, B. V. Saunders, H. S. Cohl, and M. A. McClain, eds.

[12] Antonio J. Durán, Mario Pérez, and Juan L. Varona. “The Misfortunes of a Trio of Mathematicians Using Computer Algebra Systems. Can We Trust in Them?” In: *Notices of the AMS* 61.10 (2014), pp. 1249–1252.

[13] Daniel Elphick, Michael Leuschel, and Simon Cox. “Partial Evaluation of MATLAB”. In: *Gen. Prog. and Component Eng.* Springer, 2003, pp. 344–363. DOI: 10.1007/978-3-540-39815-8_21.

[14] Matthew England et al. “Branch cuts in Maple 17”. In: *ACM Comm. Comp. Algebra* 48.1/2 (2014), pp. 24–27. DOI: 10.1145/2644288.2644293.

[15] F. W. J. Olver et al. *NIST Handbook of Mathematical Functions*. New York, NY, USA: Cambridge University Press, 2010. ISBN: 9780521140638.

[16] André Greiner-Petter et al. “Comparative Verification of the Digital Library of Mathematical Functions and Computer Algebra Systems”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Cham: Springer International Publishing, Apr. 2022, pp. 87–105. DOI: 10.1007/978-3-030-99524-9_5.

[17] André Greiner-Petter et al. “Discovering Mathematical Objects of Interest - A Study of Mathematical Notations”. In: *WWW*. ACM, 2020, pp. 1445–1456. DOI: 10.1145/3366423.3380218.

[18] André Greiner-Petter et al. “Making Presentation Math Computable: Proposing a Context Sensitive Approach for Translating LaTeX to Computer Algebra Systems”. In: *International Congress of Mathematical Software (ICMS)*. Vol. 12097. Lecture Notes in Computer Science. Springer, 2020, pp. 335–341. DOI: 10.1007/978-3-030-52200-1_33.

[19] André Greiner-Petter et al. “Semantic Preserving Bijective Mappings for Expressions Involving Special Functions between Computer Algebra Systems and Document Preparation Systems”. In: *Aslib Journal of Information Management* 71.3 (2019), pp. 415–439. DOI: 10.1108/AJIM-08-2018-0185.

[20] John Harrison. “HOL Light: A Tutorial Introduction”. In: *Formal Methods in Computer-Aided Design (FMCAD)*. Ed. by Mandayam Srivas and Albert Camilleri. Vol. 1166. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 265–269. ISBN: 978-3-540-49567-3. DOI: 10.1007/BFb0031814.

[21] John R. Harrison and Laurent Théry. “A Skeptic’s Approach to Combining HOL and Maple”. In: *Journal of Automated Reasoning* 21.3 (1998), pp. 279–294. DOI: 10.1023/A:1006023127567.

[22] Jónathan Heras, Vico Pascual, and Julio Rubio. “Using Open Mathematical Documents to Interface Computer Algebra and Proof Assistant Systems”. In: *Intelligent Computer Mathematics MKM at CICM*. Vol. 5625. Lecture Notes in Computer Science. Springer, 2009, pp. 467–473. DOI: 10.1007/978-3-642-02614-0_37.

[23] Thomas Hickman, Christian Pardillo Laursen, and Simon Foster. “Certifying Differential Equation Solutions from Computer Algebra Systems in Isabelle/HOL”. In: (Feb. 4, 2021). arXiv: 2102.02679 `[cs, math]`. URL: http://arxiv.org/abs/2102.02679.

[24] David J. Jeffrey and Arthur C. Norman. “Not Seeing the Roots for the Branches: Multivalued Functions in Computer Algebra”. In: *SIGSAM Bulletin* 38.3 (Sept. 2004), pp. 57–66. DOI: 10.1145/1040034.1040036.

[25] Cezary Kaliszyk and Freek Wiedijk. “Certified computer algebra on top of an interactive theorem prover”. In: *Towards Mechanized Math. Assist.* Springer, 2007, pp. 94–105. DOI: 10.1007/978-3-540-73086-6_8.

[26] Muhammad Taimoor Khan. “Formal Specification and Verification of Computer Algebra Software”. PhD thesis. Johannes Kepler University Linz, Apr. 2014.

[27] Giovanni Yoko Kristianto, Goran Topić, and Akiko Aizawa. “Utilizing dependency relationships between math expressions in math IR”. In: *Information Retrieval Journal* 20.2 (Mar. 2017), pp. 132–167. DOI: 10.1007/s10791-017-9296-8.

[28] Laureano Lambán et al. “Verifying the bridge between simplicial topology and algebra: the Eilenberg-Zilber algorithm”. In: *Logic Journal of IGPL* 22.1 (Aug. 2013), pp. 39–65.

DOI: 10.1093/jigpal/jzt034.

[29] Wonyeol Lee, Rahul Sharma, and Alex Aiken. “On automatically proving the correctness of math.h implementations”. In: *Proc. ACM on Prog. Lang. (POPL)* 2.47 (2018), pp. 1–32. DOI: 10.1145/3158135.

[30] K. Rustan M. Leino. “Program proving using intermediate verification languages (IVLs) like Boogie and Why3”. In: *ACM SIGAda Ada Letters* 32.3 (Nov. 2012), pp. 25–26. DOI: 10.1145/2402709.2402689.

[31] Robert H. Lewis and Michael Wester. “Comparison of Polynomial-Oriented Computer Algebra Systems”. In: *SIGSAM Bull.* 33.4 (Dec. 1999), pp. 5–13. DOI: 10.1145/500457.500459.

[32] Bruce R. Miller and Abdou Youssef. “Technical Aspects of the Digital Library of Mathematical Functions”. In: *Ann. Math. Artif. Intell.* 38.1-3 (2003), pp. 121–136. DOI: 10.1023/A:1022967814992.

[33] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic.* Vol. 2283. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2002. ISBN: 3-540-43376-7. DOI: 10.1007/3-540-45949-9.

[34] Bernard Parisse. “Compiling LATEX to Computer Algebra-Enabled HTML5”. In: (July 5, 2017). arXiv: 1707.01271 [cs]. URL: http://arxiv.org/abs/1707.01271.

[35] Hélène Prieto, Stéphane Dalmas, and Yves Papegay. “Mathematica as an OpenMath Application”. In: *ACM SIGSAM Bulletin* 34.2 (June 2000), pp. 22–26. ISSN: 0163-5824. DOI: 10.1145/362001.362016.

[36] Moritz Schubotz et al. “Improving the Representation and Conversion of Mathematical Formulae by Considering their Textual Context”. In: *ACM/IEEE JCDL.* ACM, 2018, pp. 233–242. DOI: 10.1145/3197026.3197058.

[37] Moritz Schubotz et al. “Semantification of Identifiers in Mathematics for Better Math Information Retrieval”. In: *ACM SIGIR’16.* ACM Press, 2016, pp. 135–144. DOI: 10.1145/2911451.2911503.

[38] Ruocheng Shan and Abdou Youssef. “Towards Math Terms Disambiguation Using Machine Learning”. In: *Proceedings of the International Conference on Intelligent Computer Mathematics (CICM).* Ed. by Fairouz Kamareddine and Claudio Sacerdoti Coen. Vol. 12833. Lecture Notes in Computer Science. Timisoara, Romania: Springer, 2021, pp. 90–106. ISBN: 978-3-030-81096-2. DOI: 10.1007/978-3-030-81097-9_7.

[39] Abdou Youssef. “Part-of-Math Tagging and Applications”. In: *Intelligent Computer Mathematics CICM.* Vol. 10383. Lecture Notes in Computer Science. Springer, 2017, pp. 356–374. DOI: 10.1007/978-3-319-62075-6_25.

[40] Abdou Youssef and Bruce R. Miller. “A Contextual and Labeled Math-Dataset Derived from NIST’s DLMF”. In: *Intelligent Computer Mathematics CICM.* Vol. 12236. Lecture Notes in Computer Science. Springer, 2020, pp. 324–330. DOI: 10.1007/978-3-030-53518-6_25.

[41] Richard Zanibbi et al. “Overview of ARQMath 2020: CLEF Lab on Answer Retrieval for Questions on Math”. In: *CLEF.* Vol. 12260. Lecture Notes in Computer Science. Springer, 2020, pp. 169–193. DOI: 10.1007/978-3-030-58219-7_15.

Appendix

A MEOM Blueprints

In this section, we briefly explain the MEOM blueprints. Those blueprints are mathematical expressions with wild cards which are tied to a specific rule-based interpretations. If one of our blueprints matches an expression, we identified the necessary MEOM elements, i.e., the argument(s) and bound variable(s) of the mathematical operators.

For our MEOM blueprints, we define three placeholders (wild cards): `varN` for single identifiers or a list of identifiers (delimited by commas), `numL1`, and `numU1`, representing lower and upper bound expressions, respectively. In addition, for sums and products, we need to distinguish between including and excluding boundaries, e.g., $1<k$ and $1\leq k$. An excluding relation, such as $0<k<10$, must be interpreted as a sum from 1 to 9. Table 2 shows the final set of sum/product subscript blueprints.

Table 2: The table contains examples of the blueprints for subscripts of sums/products including an example expression that matches the blueprint.

Blueprints	Example
`numL1 \leq var1 < var2 \leq numU1`	$0\leq n<k\leq 10$
`-\infty < varN < \infty`	$-\infty<n<\infty$
`numL1 < varN < numU1`	$0<n,k<10$
`numL1 \leq varN < numU1`	$0\leq k<10$
`numL1 < varN \leq numU1`	$0<n,k\leq 10$
`varN \leq numU1`	$n,k\leq N+5$
`varN \in numL1`	$n\in\{1,2,3\}$
`varN = numL1`	$n,k,l=1$

Standard notations may not explicitly show infinity boundaries. Hence, we set the default boundaries to infinity. For limit expressions we need different blueprints to capture the limit direction. We cover the standard notations with '`var1 \to numL*`', where `*` is either `+`, `-`, `^+`, `^-` or absent and the different arrow-notations where `\to` can be either `\downarrow`, `\uparrow`, `\searrow`, or `\nearrow`, specifying one-sided limits. Note that the arrow-notation (besides `\to`) is not used in the DLMF and thus, has no effect on the performance of LACAST in our evaluation.

The blueprint approach can be easily extended to new patterns, which helps to maintain LACAST and support more expressions. In fact, the blueprint approach is flexible enough to parse more complex situations, such as multi-line subscript expressions. However, there are scenarios in which the blueprint approach is not enough to perform a translation. Consider the divisor sum $\sum_{(p-1)|2n} 1/p$ [11, (24.10.1)], where the sum is over all p such that $p-1$ divides $2n$. A proper translation needs to acknowledge that $p-1$ rather than p divides $2n$. Hence, a translation to `Mathematica` potentially manipulates the p in the argument of the sum, to adjust this. A proper translation could be `Sum[1/(p+1), {p, Divisors[2*n]}]`. However, such manipulations quickly increase in complexity and require symbolic computation when reaching a certain level. This is currently out of scope for LACAST. Note that blueprints could also cover several scenarios with ellipsis, such as in $\sum_{1<n_1<\cdots<n_k<m}$. However, a proper analysis of expressions with ellipsis is still an open issue for LACAST.

B Why Branch Cuts Matter

Problems that we regularly faced during evaluation are issues related to multi-valued functions. Multi-valued functions map values from a domain to multiple values in a codomain and frequently appear in the complex analysis of elementary and special functions. Prominent examples are the inverse trigonometric functions, the complex logarithm, or the square root. A proper mathematical description of multi-valued functions requires the complex analysis of Riemann surfaces. Riemann surfaces are one-dimensional complex manifolds associated with a multi-valued function. One usually multiplies the complex domain into a many-layered covering space. The correct properties of multi-valued functions on the complex plane may no longer be valid by their counterpart functions on CAS, e.g., $(1/z)^w$ and $1/(z^w)$ for $z,w \in \mathbb{C}$ and $z \neq 0$. For example, consider $z,w \in \mathbb{C}$ such that $z \neq 0$. Then mathematically, $(1/z)^w$ always equals $1/(z^w)$ (when defined) for all points on the Riemann surface with fixed w. However, this should certainly not be assumed to be true in CAS, unless very specific assumptions are adopted (e.g., $w \in \mathbb{Z}, z > 0$). For all modern CAS[26], this equation is not true. Try, for instance, $w = 1/2$. Then $(1/z)^{1/2} - 1/z^{1/2} \neq 0$ on CAS, nor for w being any other rational non-integer number.

The resulting ranges of multi-valued functions are referred to as branches, and the curves which separate these branches are called branch cuts. The restricted range which is associated with the range typically adopted using real numbers, is often referred to as the principal branch. In order to compute multi-valued functions, CAS choose branch cuts for these functions so that they may evaluate them on their principal branches. Branch cuts may be positioned differently among CAS [10], e.g., $\operatorname{arccot}(-\frac{1}{2}) \approx 2.03$ in `Maple` but is ≈ -1.11 in `Mathematica`. This is certainly not an error and is usually well documented for specific CAS [14, 24]. However, there is no central database that summarizes branch cuts in different CAS or DML. The DLMF as well, explains and defines their branch cuts carefully but does not carry the information within the info boxes of expressions. Due to complexity, it is rather easy to lose track of branch cut positioning and evaluate expressions on incorrect values. For example, consider the equation [11, (12.7.10)]. A path of $z(\phi) = e^{i\phi}$ with $\phi \in [0, 2\pi]$ would pass three different branch cuts. An accurate evaluation of the values of $z(\phi)$ in CAS require calculations on the three branches using analytic continuation. LACAST and our evaluation frequently fall into the same trap by evaluating values that are no longer on the principal branch used by CAS. To solve this issue, we need to utilize branch cuts not only for every function but also for every equation in the DLMF [19]. The positions of branch cuts are exclusively provided in the text but not in the i-boxes. Adding the information to each equation in the DLMF would be a laborious process because a branch cut position may change according to the used values (see the example [11, (12.7.10)] from above). Our result data, however, would provide beneficial information to update, extend, and maintain the DLMF, e.g., by adding the positions of the branch cuts for every function.

[26] The authors are not aware of any example of a CAS which treats multi-valued functions without adopting principal branches.

C Overview of Bug Reports and Discovered Issues

Throughout the development of LACAST and especially during the research on this paper, we identified several issues in the DLMF, `Maple`, and `Mathematica`. Some of these issues are severe while most of them are minor problems. With this section, we want to take the opportunity to conclude the progress of LACAST as a verification approach and summarize the more prominent issues we discovered over the time. Please note that some of these issues (especially in regard of the DLMF and `Maple`) have been reported before and even published in previous publications.

C.1 Digital Library of Mathematical Functions

Since LACAST was always developed in collaboration with developers of the DLMF, numerous of minor fixes, tweaks, and updates have been implemented over the time. Most of them are not worth noting with a few exceptions. The first error in the DLMF that we discovered with the help of LACAST [19] was the sign error in [11, (14.5.14)]

$$\mathrm{Q}_{\nu}^{-1/2}(\cos\theta) = -\left(\frac{\pi}{2\sin\theta}\right)^{1/2} \frac{\cos\left(\left(\nu+\frac{1}{2}\right)\theta\right)}{\nu+\frac{1}{2}}. \tag{2}$$

This error also appeared in the original *Handbook of Special Functions* [15, p. 359] and was fixed with DLMF version 1.0.16 in September 2017.

An entire group of equations [11, (10.19.10-11)] used semantic macros incorrectly and therefore yielded to wrong links and annotations visible in the attached information box next to the equation in the DLMF. In these formulae, $P_k(a)$ and $Q_k(a)$ are defined but had been incorrectly marked up as Legendre functions going all the way back to DLMF version 1.0.0. This error has been fixed due to our feedback with DLMF version 1.0.27 in June 2020.

Minor discovered issues include a missing comma in the constraint [11, (10.16.7)] $2\nu \neq -1, -2, -3, \ldots$ which was also missing in the DLMF book [15, p. 228] (fixed with v. 1.0.19), unmarked [11, (2.4.2)] or erroneously marked definitions in [11, (9.10.10)] and in [11, (9.13.1)] (all remain unsolved), and wrong annotations of ν as *complex parameter* and x as *real* while *real value* and *positive real*, respectively, would be correct in [11, (10.24.4)] (remain unsolved). Additionally, due to LACAST, the ambiguous semantic macro `\Wron` for Wronskians has been revised so that the variable which is differentiated against is precisely specified in 72 occasions [8].

C.2 Maple

Via LACAST, we discovered a bug in `Maple`'s 2016 `simplify` procedure. For the equation [11, (7.18.4)]

$$\frac{\mathrm{d}^n}{\mathrm{d}z^n}\left(e^{z^2}\operatorname{erfc} z\right) = (-1)^n 2^n n! e^{z^2} \mathrm{i}^n \operatorname{erfc}(z), \quad n = 0,1,2,\ldots, \tag{3}$$

where e is the base of the natural logarithm, $\operatorname{erfc}(z)$ is the complementary error function, and $\mathrm{i}^n\operatorname{erfc}(z)$ the repeated integrals of the complementary error function, LACAST correctly generated the following translation:

LaCASt translation of equation (3) to Maple

```
diff( exp(z^2)*erfc(z), [z$(n)] ) = (-1)^(n)*(2)^(n)*
    factorial(n)*exp(z^2)*erfc(n, z)
```

Redundant parentheses removed to improve readability.

Maple 2016 falsely returns 0 when we call the simplify procedure for the translated left-hand side of the equation. Maplesoft has confirmed this defect in Maple 2016 in private communications [19]. Although an updated behavior occurred in Maple 2018 and 2020, the error still persists today. Maple version 2020.2 automatically evaluates the left-hand side of equation (3) to the rather complex expression

$$\sum_{k=0}^{n}\frac{1}{\sqrt{\pi}}\binom{n}{k}\left(\sum_{m=0}^{k}e^{z^2}B_{k,m}\left((2)_1 z,\ldots,(2-k+m)_{k-m+1}z^{1-k+m}\right)\right)$$
$$\left(-G_{2,3}^{1,2}\left(z;{0,\frac{1}{2} \atop 0,-\frac{1}{2}+\frac{n}{2}-\frac{k}{2},\frac{n}{2}-\frac{k}{2}}\right)2^{n-k}+(1-n+k)_{n-k}\sqrt{\pi}z^{n-k-1}\right)z^{1-n+k}, \tag{4}$$

where $G_{p,q}^{m,n}\left(z;{a_1,\ldots,a_p \atop b_1,\ldots,b_q}\right)$ is the Meijer G-function [11, (16.17.1)], $B_{n,k}(x_1,\ldots,x_{n-k+1})$ the incomplete Bell polynomials [3], and $(x)_n$ the Pochhammer's symbol [11, (5.2.4)]. For small n and z, the difference of left- and right-hand side of equation (3) is indeed almost zero up to the machine accuracy. For large absolute values of z, however, the difference increases quickly. Figure 4 plots the difference of left- and right-hand side of equation (3) for $n=0$ and $z\in[-2,2]$. For $|z|>4$, the difference is already larger then 1.0. This might be produced by accumulated round-off errors because smaller values are calculated with greater precision in floating-point arithmetics.

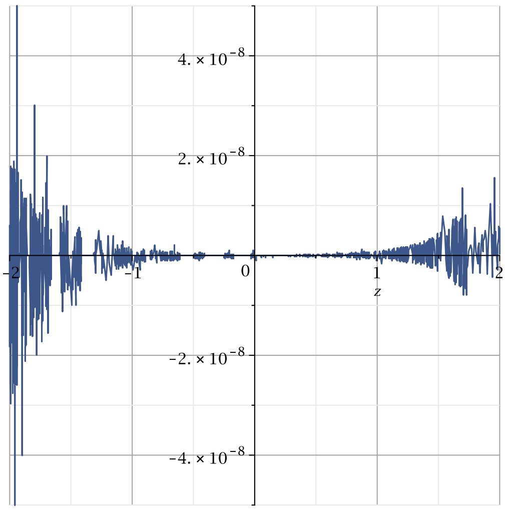


Fig. 4: The difference of the left- and right-hand side of equation (3) evaluated in Maple for $n=0$ and $z\in[-2,2]$.

C.3 Mathematica

As we pointed out in Section 5.1, we discovered some trouble with integrals in `Mathematica` and confusing behavior with rational numbers. After discussing these cases with `Mathematica` developers, some of them have been confirmed as bugs. Other cases, however, were the results of our testing methodology. First, we take a look at the confirmed errors. The most crucial report was about [11, (18.17.14)]

$$\frac{x^{\alpha+\mu} L_n^{(\alpha+\mu)}(x)}{\Gamma(\alpha+\mu+n+1)} = \int_0^x \frac{y^\alpha L_n^{(\alpha)}(y)}{\Gamma(\alpha+n+1)} \frac{(x-y)^{\mu-1}}{\Gamma(\mu)} \mathrm{d}y. \tag{5}$$

For this equation, we calculated the difference of the left- and right-hand side as usual

$$\frac{x^{\alpha+\mu} L_n^{(\alpha+\mu)}(x)}{\Gamma(\alpha+\mu+n+1)} - \int_0^x \frac{y^\alpha L_n^{(\alpha)}(y)}{\Gamma(\alpha+n+1)} \frac{(x-y)^{\mu-1}}{\Gamma(\mu)} \mathrm{d}y \tag{6}$$

and computed numerical test values for this difference. In particular, we identified the four variables x, n, α, and μ. As described in Figure 3 (Section 4.2) in the paper, n is defined as a special variable bind to the numeric values $\{1,2,3\}$, x and α are positive real values of our general test values, i.e., $x,\alpha \in \{\frac{1}{2},\frac{3}{2},2\}$, and μ is not further limited, i.e., $\mu \in \left\{\pm\frac{1}{2},\pm\frac{3}{2},\pm 2, e^{\frac{i\pi}{6}}, e^{\frac{2i\pi}{3}}, e^{\frac{-i\pi}{3}}, e^{\frac{-5i\pi}{6}}\right\}$. This resulted in 270 test value combinations which are further limited by the attached (local) constraints in the DLMF [11, (18.17.14)]: $\mu>0, x>0$. Since x was already constraint to positive real values with our global constraints, the second local constraint has no additional effect. For μ, we must note that we also removed invalid comparisons. For example, $e^{\frac{2i\pi}{3}}>0$ throws an error in `Mathematica` due to the invalid comparison with an imaginary number. Hence, $\mu>0$ filtered out imaginary numbers and negative real values. Here, we ended up with $\{\frac{1}{2},\frac{3}{2},2\}$. Additionally, we see equation 5 contains $\Gamma(\alpha+\mu+n+1)$, $\Gamma(\alpha+n+1)$, and $\Gamma(\mu)$. The definition of the Gamma function in the DLMF [11, (5.2.1)] contains the additional constraint $\Re z > 0$ where z is the argument of the Gamma function. Hence, we retrieved three additional constraints for equation 5: $\Re(\alpha+\mu+n+1)>0$, $\Re(\alpha+n+1)>0$, and $\Re(\mu)>0$. However, none of the constraints further reduced our test value set. Conclusively, we end up with 81 ($=3^4$) test value combinations: $n \in \{1,2,3\}$ and $x,\alpha,\mu \in \{\frac{1}{2},\frac{3}{2},2\}$.

Interestingly, for rational inputs (fractions) of test values, the difference was zero. For example, evaluating the difference on $x,a,\mu=\frac{3}{2}$ and $n=2$ returns zero. However, the same test with floating point numbers, i.e., $x,a,\mu=1.5$ and $n=2$, result in a fatal segmentation fault[27] causing `Mathematica` to crash.

[27] A *segmentation fault* is an access violation of protected memory. For example, an operating system can prevent a program `A` to change the memory of another program `B` and sends a *segmentation fault* to `A`. This signal generally causes program `A` to abnormally terminate, unless special error handling was implemented.

Segmentation Fault Example in Mathematica v. 12.1.1

```
In[1] := expr = -Integrate[((x - y)^(-1 + mu)*y^a*
         LaguerreL[n, a, y])/ (Gamma[1 + n + a]*Gamma[mu]
         ), {y, 0, x}] + (x^(a + mu)*LaguerreL[n, a + mu,
          x])/ Gamma[1 + n + a + mu];

In[2] := ReplaceAll[expr, {n -> 2, x -> 3/2, a -> 3/2, mu
          -> 3/2}]
Out[2] = 0

In[3] := ReplaceAll[expr, {n -> 2, x -> 1.5, a -> 1.5, mu
          -> 1.5}]
Segmentation fault (core dumped)
```

This issue was reported[28] and later fixed[29] with `Mathematica` version 12.2 (released November 2020). The developers told us this issue can be traced back to version 10.4 (released March 2016).

We further identified errors in the variable extraction procedure in `Mathematica`. For example, for [11, (24.4.26)]

$$E_n(0)=-E_n(1)=-\frac{2}{n+1}(2^{n+1}-1)B_{n+1}, \tag{7}$$

we expected to extract just n as the free variable. We reduced the issue to a minimal working example just for the most left-hand side of the equation.

False Variable Extraction in Mathematica v. 12.0

```
In[1] := Reduce`FreeVariables[EulerE[n, 0]]
Out[1] = {EulerE[n, 0]}
```

This particular error was confirmed and has been fixed[30]. However, since the procedure `Reduce`FreeVariables` is not a publicly documented function in `Mathematica`, the method remain unstable. Especially in mathematical operators with bounded variables, such as sums, products, integrals, and limits, the procedure tend to generate inaccurate results.

In regard of the outlined issues with the `GenerateConditions` flag in integrals, most problematic cases were the result of using `ReplaceAll` to set numeric values for variables. Consider, for example [11, (10.43.8)]

$$\int_0^x e^{\pm t}t^{-\nu}I_\nu(t)\mathrm{d}t=-\frac{e^{\pm x}x^{-\nu+1}}{2\nu-1}(I_\nu(x)\mp I_{\nu-1}(x))\mp\frac{2^{-\nu+1}}{(2\nu-1)\Gamma(\nu)}. \tag{8}$$

[28] Case ID: 4664157

[29] The fix was communicated to us via a new case ID: 4776927

[30] Case ID: 4373302

First, LACAST splits the expression in two test cases by resolving $\pm$ and $\mp$. For the first case, i.e., $\pm$ is replaced by $+$ and $\mp$ by $-$, `Mathematica` automatically evaluates the test expression, i.e., the difference of left- and right-hand side of the equation:

$$\left(\int_0^x e^t t^{-\nu} I_\nu(t) \mathrm{d}t\right) - \left(-\frac{e^x x^{-\nu+1}}{2\nu-1}(I_\nu(x) - I_{\nu-1}(x)) - \frac{2^{-\nu+1}}{(2\nu-1)\Gamma(\nu)}\right) \tag{9}$$

with the `GenerateConditions` set to `None` for the integral to

$$\begin{aligned}&\frac{e^x x^{-\nu}(-x+2\nu)I_\nu(x)}{-1+2\nu} + \frac{e^x x^{1-\nu}(-I_{-1+\nu}(x)+I_\nu(x))}{-1+2\nu} + \\ &\frac{e^x x^{1-\nu} I_{1+\nu}(x)}{-1+2\nu} + \frac{2^{1-\nu}}{(-1+2\nu)\Gamma(z)} + \frac{2^\nu\sqrt{\pi}\nu\sec(\pi\nu)}{\Gamma\left(\frac{3}{2}-\nu\right)\Gamma(1+2\nu)}.\end{aligned} \tag{10}$$

This happens because CAS automatically perform some computations on their inputs unless we prevent it (e.g., via `Hold`). However, evaluating this expression now on $x, \nu = 1.5$ returns 0.398942 rather than the expected zero. For $x, \nu = \frac{3}{2}$, the return value is infinity (i.e., indeterminate). The issue was acknowledged by the developers, who explained that the last term causes the behaviour, because

$$\frac{2^\nu\sqrt{\pi}\,\nu\sec(\pi\nu)}{\Gamma\left(\frac{3}{2}-\nu\right)\Gamma(1+2\nu)} \tag{11}$$

is `Infinity/Infinity` for $\nu = 3/2$. A workaround to this issue is to use `Limit` rather than `ReplaceAll` to evaluate the expression on specific values.

Limit Workaround for equation 9

```
In[2] := N[ReplaceAll[expr,{Rule[x,3/2], Rule[nu,3/2]}]]
Out[2] = Indeterminate

In[3] := Limit[ expr, {Rule[x,3/2], Rule[nu,3/2]} ]
Out[3] = 0
```

`expr` is the input of equation (9).

To the best of our knowledge, this *issue* still persists[31]. If this behavior is intended (or even desired) is up for debate. Yet, it is another characteristic of CAS to keep track of. The same workaround was suggested for [11, (11.5.2)] and [11, (11.5.8-10)]. In case of [11, (10.9.1)]

$$J_0(z) = \frac{1}{\pi}\int_0^\pi \cos(z\cos\theta)\mathrm{d}\theta, \tag{12}$$

the right-hand side of the equation was evaluated to `BesselJ[0,z]` by `Mathematica` for `GenerateConditions -> False`. Which is correct. However, without this flag (or set to `None`), `Mathematica` returns `BesselJ[0, Abs[z]]` if $z \in \mathbb{R}$.

[31] As of 9/9/2021

Conditional Flag Influence in Mathematica

```
In[1] := Divide[1,Pi]*Integrate[Cos[z*Sin[θ]], {θ, 0, Pi}
         ,GenerateConditions -> False]
Out[1] = BesselJ[0, z]

In[2] := Divide[1,Pi]*Integrate[Cos[z*Sin[θ]], {θ, 0, Pi}]
Out[2] = BesselJ[0, Abs[z]] if z∈ℝ
```

While confusing at the first glance, the output is not particularly wrong. Since $J_0(z)$ is even in the second argument and along the Real line, the absolute value is simply redundant.

In case of [11, (8.4.4)]

$$\Gamma(0,z) = \int_z^\infty t^{-1}e^{-t}\mathrm{d}t = E_1(z), \tag{13}$$

we have not received any feedback from the developers. For the difference between left- and right-hand side of the first equation

$$\Gamma(0,z) - \int_z^\infty t^{-1}e^{-t}\mathrm{d}t, \tag{14}$$

`Mathematica` conditionally returns 0 if $\Re(z) > 0$ and $\Im(z) = 0$. We would expect 0 without conditions. Setting the problematic `GenerateConditions` to `False` returns $-\ln(z)$.

Second Example of Conditional Flag Influence in Mathematica

```
In[1] := Gamma[0,z] - Integrate[(t)^(-1)*Exp[-t], {
         t,z,Infinity}]
Out[1] = 0 if Re[z] > 0 && Im[z] == 0

In[2] := Gamma[0,z] - Integrate[(t)^(-1)*Exp[-t], {
         t,z,Infinity}, GenerateConditions -> False]
Out[2] = -Log[z]
```

We noticed that another initial computation hook on the input could cause the issue. For example, if we prevent instant evaluations on the input via `Hold` and evaluate the expression on $z = \mathrm{i}$, `Mathematica` returns $0.+0.\mathrm{i}$. Without `Hold`, an evaluation on the same value returns `undefined`.

Hold Inputs in `Mathematica`

```
In[1] := expr = Gamma[0,z] - Integrate[(t)^(-1)*Exp[-t],
         {t,z,Infinity}]
Out[1] = 0 if Re[z] > 0 && Im[z] == 0

In[2] := N[ReplaceAll[expr, {z -> I}]]
Out[2] = Undefined

In[3] := expr = Hold[Gamma[0,z] - Integrate[(t)^(-1)*Exp[
         -t], {t,z,Infinity}]]
Out[3] = Hold[Gamma[0,z] - Integrate[(t)^(-1)*Exp[-t], {
         t,z,Infinity}]]

In[4] := N[ReleaseHold[ReplaceAll[expr, {z -> I}]]]
Out[4] = 0. + 0. i
```

All cases were handled by the `Mathematica` team with the case ID 4664157.

D Evaluation Tables

In this section, we provide three additional tables for our evaluation and translation results. Table 3, provides three examples of our evaluations on the DLMF with different degrees of complexity. The first entry [11, (1.4.8)], for example, illustrates the difficulty of translating formulae from LaTeX to CAS syntaxes even on a semantically enriched dataset like the DLMF. Often, the arguments of a function in derivative notations are omitted since they can be deduced from the variable of differentiation. For example, $\frac{\mathrm{d}^2 f}{\mathrm{d}x^2}$ the argument of the function $f(x)$ is omitted. However, in this case LaCASt is unable to correctly interpret f as a function and presumed it to be a variable. Unfortunately, not only caused this error a wrong translation but also produced a false positive evaluation because the symbolic simplification returned $0=0$ for

```
D[f, {x, 2}] == D[D[f, x], x].
```
(15)

The other two examples in Table 3, even though more complex, illustrate the capability of LaCASt and our evaluation pipeline.

Additionally, Table 5 and 6 show the number of translated and evaluated expressions for each chapter of the DLMF. For reference, Table 4 shows the full name of each chapter and the total number of displayed formulae according to the released dataset by Youssef and Miller [40]. The actual number of functions may vary compared to Table 5, because Youssef and Miller did not split multi-equations, and $\pm$ or $\mp$. Hence, our final dataset consists of 10,930 formulae. Additionally, as described in our paper, we filter out non-semantic expressions, non-semantic macro definitions, ellipsis, approximations, and asymptotics. We ended up with 6,623 test cases. A more comprehensive table and all data is available at https://lacast.wmflabs.org. For overview reasons, Table 6 only shows the results for translations to `Mathematica`. For `Maple`, see our website.

Table 3: The table shows three sample cases of our evaluation pipeline from the DLMF. The translation shows the performed translations to Mathematica. The numeric column contains the number of successfully computed test cases. The constraints column contains all applied constraints including global constraints from Figure 3.

[11, (1.4.8)]	$\frac{d^2 f}{dx^2}=\frac{d}{dx}\left(\frac{df}{dx}\right)$			
Translation ✘	`D[f, {x, 2}] == D[D[f, x], x]` (A correct translation requires the argument for f, such as `D[f[x], {x, 2}] == D[D[f[x], x], x]` .)			
Symbolic	Numeric	Variables	Constraints	Test Values
✓	✓(30/30)	f ✘, x ✓	$x\in\mathbb{R}$, $\Re(x)>0$	$x\in\left\{\frac{1}{2},\frac{3}{2},2\right\}$, $f\in\left\{\pm\frac{1}{2},\pm\frac{3}{2},\pm 2,e^{2i\pi/3},e^{i\pi/6},e^{-i\pi/3},e^{-5i\pi/6}\right\}$
[11, (11.5.2)]	$\mathbf{K}_{\nu}(z)=\frac{2(\frac{1}{2}z)^{\nu}}{\sqrt{\pi}\Gamma(\nu+\frac{1}{2})}\int_0^{\infty}e^{-zt}(1+t^2)^{\nu-\frac{1}{2}}dt$			
Translation ✓	`StruveH[\[Nu],z]-BesselY[\[Nu],z]==Divide[2*(Divide[1,2]*z)^\[Nu],Sqrt[Pi]*Gamma[\[Nu]+Divide[1,2]]]*Integrate[Exp[-z*t]*(1+(t)^(2))^(\[Nu]-Divide[1,2]),{t,0,Infinity}, GenerateConditions->None]`			
Symbolic	Numeric	Variables	Constraints	Test Values
✓	✘(10/25)	ν,z ✓	$-\pi<\mathrm{ph}(z)<\pi$, $\Re(z)>0$, $\Re(\nu+\frac{1}{2})>0$, $\Re(\nu+k+1)>0$, $\Re(-\nu+k+1)>0$, $\Re(n+\nu+\frac{3}{2})>0$	$\nu,z\in\left\{\frac{1}{2},\frac{3}{2},2,e^{i\pi/6},e^{-i\pi/3}\right\}$
[11, (18.5.8)]	$P_n^{(\alpha,\beta)}(x)=2^{-n}\sum_{\ell=0}^{n}\binom{n+\alpha}{\ell}\binom{n+\beta}{n-\ell}(x-1)^{n-\ell}(x+1)^{\ell}$			
Translation ✓	`JacobiP[n,\[Alpha],\[Beta],x]==(2)^(-n)*Sum[Binomial[n+\[Alpha],\[ScriptL]]*Binomial[n+\[Beta],n-\[ScriptL]]*(x-1)^(n-\[ScriptL])*(x+1)^\[ScriptL],{\[ScriptL],0,n}, GenerateConditions->None]`			
Symbolic	Numeric	Variables	Constraints	Test Values
✘	✓(81/81)	n,α,β,x ✓	$n\in\{1,2,3\}$, $x,\alpha,\beta\in\mathbb{R}$, $x,\alpha,\beta>0$	$n\in\{1,2,3\}$, $\alpha,\beta,x\in\left\{\frac{1}{2},\frac{3}{2},2\right\}$

Table 4: Summary of DLMF chapters with corresponding 2-letter codes (2C), chapter numbers (C#), chapter names of the DLMF chapters, and the total number of displayed formulas (i.e., without inline formulae) per chapter according to [40] (F).

2C	**C#**	**Chapter Name**	**F**
AL	1	Algebraic and Analytic Methods	571
AS	2	Asymptotic Approximations	349
NM	3	Numerical Methods	296
EF	4	Elementary Functions	513
GA	5	Gamma Function	161
EX	6	Exponential, Logarithmic, Sine, and Cosine Integrals	100
ER	7	Error Functions, Dawson's and Fresnel Integrals	137
IG	8	Incomplete Gamma and Related Functions	240
AI	9	Airy and Related Functions	230
BS	10	Bessel Functions	696
ST	11	Struve and Related Functions	149
PC	12	Parabolic Cylinder Functions	177
CH	13	Confluent Hypergeometric Functions	372
LE	14	Legendre and Related Functions	283
HY	15	Hypergeometric Function	198
GH	16	Generalized Hypergeometric Functions & Meijer G-Function	99
QH	17	q-Hypergeometric and Related Functions	181
OP	18	Orthogonal Polynomials	502
EL	19	Elliptic Integrals	464
TH	20	Theta Functions	113
MT	21	Multidimensional Theta Functions	59
JA	22	Jacobian Elliptic Functions	258
WE	23	Weierstrass Elliptic and Modular Functions	167
BP	24	Bernoulli and Euler Polynomials	188
ZE	25	Zeta and Related Functions	171
CM	26	Combinatorial Analysis	201
NT	27	Functions of Number Theory	132
MA	28	Mathieu Functions and Hill's Equation	353
LA	29	Lamé Functions	205
SW	30	Spheroidal Wave Functions	114
HE	31	Heun Functions	120
PT	32	Painlevé Transcendents	286
CW	33	Coulomb Functions	146
TJ	34	$3j$, $6j$, $9j$ Symbols	64
FM	35	Functions of Matrix Argument	62
IC	36	Integrals with Coalescing Saddles	137
		Σ	8,494

Table 5: Overview of translations for DLMF chapters. Table headings are 2C: 2-letter chapter codes; C#: chapter numbers; F: number of formulae; $\mathrm{T}_{\mathrm{old}}$: number of translated expressions using old translator; $\mathrm{T}_{\mathrm{Map}}$, $\mathrm{T}_{\mathrm{Math}}$: number of translations with improved translator—Map for Maple and Math for Mathematica; $\mathrm{M}_{\mathrm{Map}}$, $\mathrm{M}_{\mathrm{Math}}$: number of failed translations due to missing macro translation; E: number of other errors in the translation process. Best five performances are colored. Chapter codes are linked with our result page https://lacast.wmflabs.org.

2C	**C#**	**F**	$\mathbf{T}_{\mathbf{old}}$	$\mathbf{T}_{\mathbf{Map}}$	$\mathbf{T}_{\mathbf{Math}}$	$\mathbf{M}_{\mathbf{Map}}$	$\mathbf{M}_{\mathbf{Math}}$	**E**
AL	1	227	60 (26.4%)	102 (44.9%)	103 (45.4%)	79	78	46
AS	2	136	33 (24.3%)	65 (47.8%)	65 (47.8%)	51	51	20
NM	3	53	36 (67.9%)	40 (75.5%)	40 (75.5%)	8	8	5
EF	4	569	353 (62.0%)	494 (89.3%)	564 (99.1%)	58	4	1
GA	5	144	38 (26.4%)	130 (93.5%)	139 (96.5%)	7	3	2
EX	6	107	21 (19.6%)	56 (52.3%)	77 (72.0%)	50	29	1
ER	7	149	35 (23.5%)	101 (67.8%)	120 (80.5%)	47	28	1
IG	8	204	84 (41.2%)	160 (78.4%)	163 (79.9%)	39	36	5
AI	9	235	36 (15.3%)	180 (76.6%)	179 (76.2%)	46	47	9
BS	10	653	143 (21.9%)	392 (60.0%)	486 (74.4%)	243	135	32
ST	11	124	48 (38.7%)	121 (97.6%)	112 (90.3%)	3	12	0
PC	12	106	33 (31.1%)	79 (74.5%)	90 (84.9%)	23	9	7
CH	13	260	126 (48.5%)	252 (96.9%)	254 (97.7%)	3	1	5
LE	14	238	166 (69.7%)	230 (96.6%)	229 (96.2%)	5	6	3
HY	15	206	148 (71.8%)	198 (96.1%)	197 (95.6%)	3	4	5
GH	16	53	20 (37.7%)	23 (43.4%)	25 (47.2%)	27	1	27
QH	17	175	1 (0.6%)	53 (30.3%)	124 (70.8%)	112	35	16
OP	18	468	132 (28.2%)	235 (50.2%)	288 (61.5%)	203	149	31
EL	19	516	103 (20.0%)	252 (48.8%)	416 (80.6%)	250	84	16
TH	20	128	52 (40.6%)	98 (76.6%)	98 (76.6%)	8	8	22
MT	21	32	0 (0.0%)	0 (0.0%)	0 (0.0%)	30	30	2
JA	22	264	115 (43.6%)	232 (87.9%)	238 (90.2%)	27	21	5
WE	23	164	7 (4.3%)	19 (11.6%)	34 (20.7%)	125	112	18
BP	24	175	31 (17.7%)	117 (67.2%)	148 (84.6%)	35	15	12
ZE	25	154	28 (18.2%)	124 (80.5%)	120 (77.9%)	26	30	4
CM	26	136	31 (22.8%)	78 (57.3%)	87 (64.0%)	54	42	7
NT	27	79	5 (6.3%)	26 (32.9%)	15 (19.0%)	38	49	15
MA	28	267	52 (19.5%)	97 (36.3%)	110 (41.2%)	138	125	32
LA	29	111	11 (9.9%)	23 (20.7%)	22 (19.8%)	79	80	9
SW	30	71	14 (19.7%)	19 (26.8%)	26 (36.6%)	47	39	6
HE	31	35	29 (82.8%)	22 (62.8%)	15 (42.8%)	9	16	4
PT	32	67	43 (64.2%)	57 (85.1%)	57 (85.1%)	0	0	10
CW	33	108	21 (19.4%)	14 (13.0%)	11 (10.2%)	80	86	11
TJ	34	57	0 (0.0%)	1 (1.8%)	37 (64.9%)	46	4	16
FM	35	46	0 (0.0%)	0 (0.0%)	0 (0.0%)	36	36	10
IC	36	106	12 (11.3%)	24 (22.6%)	24 (22.6%)	79	79	3
Σ		6,623	2,067 (31.2%)	4,114 (62.1%)	4,713 (71.2%)	2,114	1,492	418

Table 6: Overview of symbolic and numeric evaluations for DLMF chapters as in Table 5. Table headings are $\mathrm{T}_{\mathrm{Math}}$: number of successfully translations to Mathetmatica; $\mathrm{S}_{\mathrm{success}}$, $\mathrm{S}_{\mathrm{fail}}$: number of successful and failed symbolic verifications (for translated expressions only) respectively; $\mathrm{N}_{\mathrm{success}}$, $\mathrm{N}_{\mathrm{fail}}$: number of successful and remaining failed numeric (for failed symbolical tests only) respectively. P, T: number of partial (at least one test was successful) and total failed numeric tests. Best five performances are colored. Chapter codes are linked with our result page https://lacast.wmflabs.org.

2C	**C#**	$\mathbf{T}_{\mathbf{Math}}$	$\mathbf{S}_{\mathbf{success}}$	$\mathbf{S}_{\mathbf{fail}}$	$\mathbf{N}_{\mathbf{success}}$	$\mathbf{N}_{\mathbf{fail}}$	**[P / T]**	**A**	**E**
AL	1	103	34 (33.0%)	69	14 (20.3%)	40	[9 / 31]	11	4
AS	2	65	6 (9.2%)	59	4 (6.8%)	38	[6 / 32]	7	9
NM	3	40	5 (12.5%)	35	0 (0.0%)	29	[8 / 21]	6	0
EF	4	564	304 (53.9%)	260	110 (42.3%)	146	[55 / 91]	2	0
GA	5	139	65 (46.8%)	74	30 (40.5%)	20	[9 / 11]	13	9
EX	6	77	18 (23.4%)	59	23 (39.0%)	32	[6 / 26]	4	0
ER	7	120	45 (37.5%)	75	21 (28.0%)	43	[13 / 30]	9	1
IG	8	163	65 (39.9%)	98	22 (22.4%)	44	[19 / 25]	16	15
AI	9	179	69 (38.5%)	110	30 (27.3%)	58	[38 / 20]	14	7
BS	10	486	115 (23.7%)	371	90 (24.2%)	151	[57 / 94]	92	18
ST	11	112	36 (32.1%)	76	21 (27.6%)	33	[8 / 25]	10	11
PC	12	90	18 (20.0%)	72	13 (18.0%)	43	[15 / 28]	12	3
CH	13	254	69 (27.2%)	185	23 (12.4%)	95	[59 / 36]	45	21
LE	14	229	30 (13.1%)	199	59 (29.6%)	92	[54 / 38]	41	5
HY	15	197	53 (26.9%)	144	23 (16.0%)	77	[52 / 25]	29	6
GH	16	25	2 (8.0%)	23	1 (4.3%)	10	[7 / 3]	9	2
QH	17	124	6 (4.8%)	118	13 (11.0%)	57	[52 / 5]	39	5
OP	18	288	101 (35.1%)	185	45 (24.3%)	68	[31 / 37]	52	12
EL	19	416	51 (12.2%)	365	18 (4.9%)	264	[49 /215]	61	15
TH	20	98	1 (1.0%)	97	33 (34.0%)	40	[25 / 15]	24	0
MT	21	0	-	-	-		-	-	-
JA	22	238	30 (12.6%)	206	22 (10.7%)	131	[39 / 92]	51	0
WE	23	34	4 (11.8%)	30	2 (6.7%)	23	[9 / 14]	2	3
BP	24	148	23 (15.5%)	125	78 (62.4%)	33	[22 / 11]	14	0
ZE	25	120	48 (40.0%)	72	22 (30.5%)	22	[6 / 16]	22	3
CM	26	87	19 (21.8%)	68	44 (64.7%)	18	[10 / 8]	5	1
NT	27	15	6 (40.0%)	9	3 (33.3%)	6	[3 / 3]	0	0
MA	28	110	7 (6.4%)	103	3 (2.9%)	48	[13 / 35]	33	17
LA	29	22	0 (0.0%)	22	0 (0.0%)	21	[1 / 20]	0	1
SW	30	26	0 (0.0%)	26	0 (0.0%)	19	[2 / 17]	5	1
HE	31	15	2 (13.3%)	13	0 (0.0%)	8	[0 / 8]	5	0
PT	32	57	3 (5.3%)	54	0 (0.0%)	41	[2 / 39]	8	5
CW	33	11	0 (0.0%)	11	0 (0.0%)	11	[2 / 9]	0	0
TJ	34	37	0 (0.0%)	37	14 (37.8%)	10	[5 / 5]	13	0
FM	35	0	-	-	-		-	-	-
IC	36	24	0 (0.0%)	24	3 (12.5%)	13	[1 / 12]	1	6
Σ		4,713	1,235 (26.2%)	3,474	784 (22.6%)	1,784	[687 / 1,097]	655	180

Listing 1.1: Use the following BibTeX code to cite this article

```
@inproceedings{GreinerPetter2022,
  author       = {Andr\'{e} Greiner-Petter and Howard S.~Cohl
                  and Abdou Youssef and Moritz Schubotz
                  and Avi Trost and Rajen Dey
                  and Akiko Aizawa and Bela Gipp},
  title        = {Comparative Verification of the Digital
                  Library of Mathematical Functions and
                  Computer Algebra Systems},
  year         = {2022},
  month        = {Apr.},
  booktitle    = {International Conference on Tools and
                  Algorithms for the Construction and Analysis
                  of Systems (TACAS)},
  publisher    = {Springer International Publishing},
  address      = {Cham},
  pages        = {87--105},
  doi          = {10.1007/978-3-030-99524-9_5}
}
```