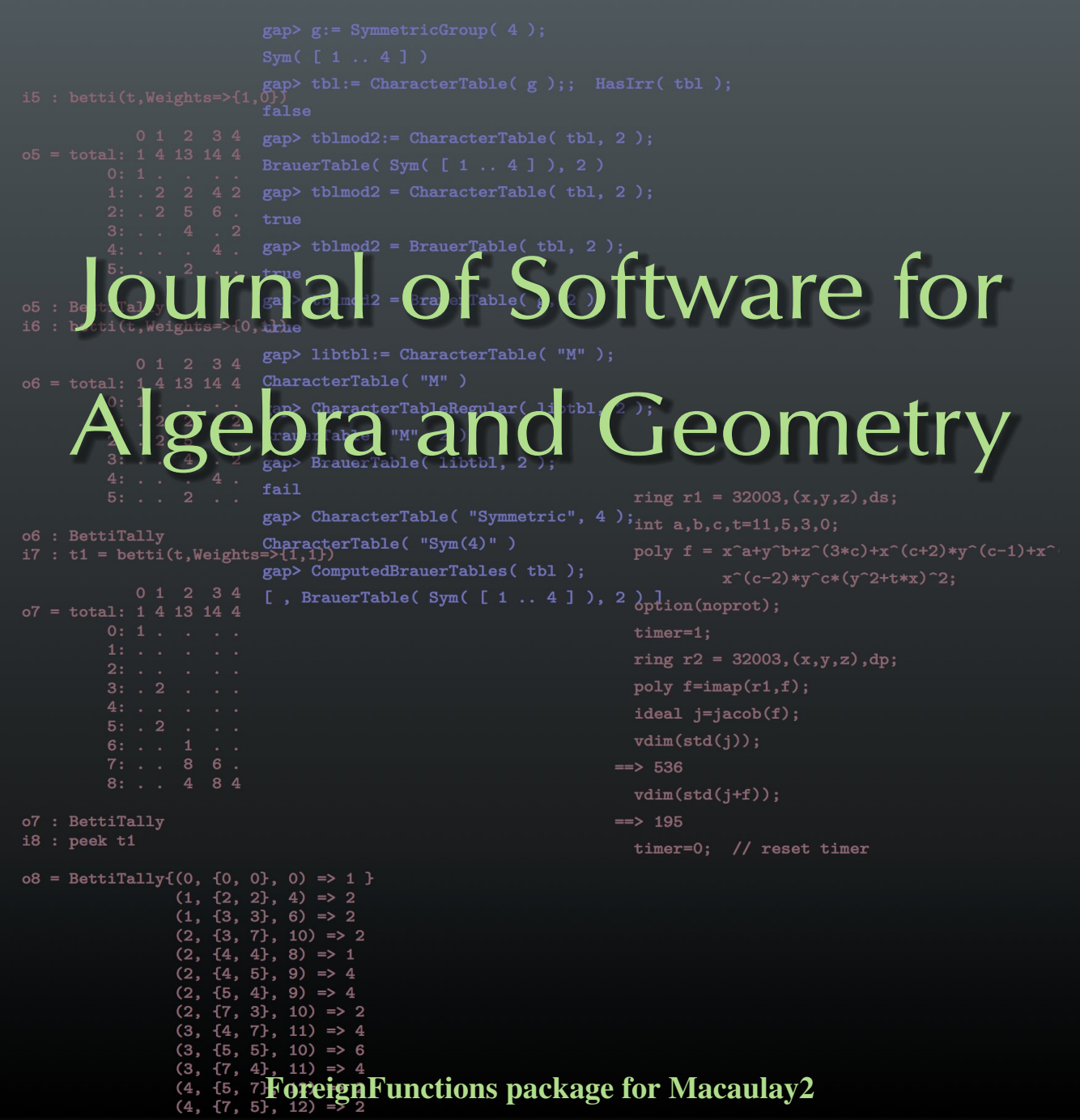

Journal of Software for Algebra and Geometry

ForeignFunctions package for Macaulay2

DOUGLAS A. TORRANCE

vol 15 2025

JSAG **15** (2025), 1–9
https://doi.org/10.2140/jsag.2025.15.1

The Journal of Software for
Algebra and Geometry

ForeignFunctions package for Macaulay2

DOUGLAS A. TORRANCE

ABSTRACT: We introduce the `ForeignFunctions` package for MACAULAY2, which uses LIBFFI to provide the ability to call functions from external libraries without needing to link against them at compile time. As examples, we use the library FFTW to multiply polynomials using fast Fourier transforms, call a LAPACK function to solve a general Gauss–Markov linear model problem, and use JIT compilation to compute Fibonacci numbers.

1. INTRODUCTION. Traditionally, there have been two ways to call functions from external libraries in MACAULAY2 [11], a software platform widely used by algebraic geometers and commutative algebraists.

One is to write some code wrapping the external library's functions. This code might involve C++ if the library is to be used by the MACAULAY2 engine, and it would also very likely involve the D language used by the interpreter and the top-level MACAULAY2 language. The library would then be statically or dynamically linked at compile time. For example, the `roots` function for computing the zeros of a polynomial is implemented in this way using the library MPSOLVE [2; 3]. The addition of a new library using this method requires rebuilding MACAULAY2 from source.

The other is to use the shell to communicate with an external program's command line interface, often creating several temporary files containing the program's input and output in the process. For example, the `Normaliz` package [4] communicates in this way with NORMALIZ, a software platform for discrete convex geometry [5].

However, using the library LIBFFI [12], it is possible to create a "foreign function interface" where users may open a shared library and call functions from it at run time, without needing to link against the library at compile time or rely on the overhead of communicating with a command line interface via the shell. Beginning with version 1.21, MACAULAY2 has been distributed with the `ForeignFunctions` package, written by author, which provides such an interface.

This paper is organized as follows. In Section 2, we describe the interface. In Section 3, we discuss how it was implemented. The last three sections are devoted to examples. In Section 4, we call functions from the FFTW library [9] to compute fast Fourier transforms to multiply polynomials. In Section 5, we use LAPACK to solve a general Gauss–Jordan linear model problem. In Section 6, we use the `ForeignFunctions` package for JIT compilation to quickly compute Fibonacci numbers.

MSC2020: primary 13-04, 14-04; secondary 65T50, 62J05.
Keywords: Macaulay2, foreign function interface.
`ForeignFunctions` version 0.4

```
i1 : needsPackage "ForeignFunctions"
o1 = ForeignFunctions
o1 : Package
i2 : openSharedLibrary "fftw3"
o2 = fftw3
o2 : SharedLibrary
```

Listing 1. Constructing a `SharedLibrary` object.

2. FOREIGN FUNCTION INTERFACE. The `ForeignFunctions` package introduces several new types that we outline in this section.

The first of these is `SharedLibrary`. Each `SharedLibrary` object contains a pointer to a shared library "handle" as returned by the C standard library function `dlopen`. The constructor method `openSharedLibrary`, which takes a string representing a library name or filename as input, calls `dlopen` and returns an instance of this class. For an example, see Listing 1, where we load the FFTW library [9] that will be used in Section 4.

There are also two related abstract classes: `ForeignType` and `ForeignObject`. MACAULAY2 users may be familiar with `PolynomialRing`/`RingElement` and `Module`/`Vector`, the abstract classes for polynomial rings/modules and their elements, respectively. These classes work similarly. There are a number of instances of `ForeignType` representing various C types such as `int` (32-bit signed integers), `double` (double-precision floating point numbers) and `char*` (pointers to null-terminated strings). Each `ForeignType` object is a class, and `ForeignObject` is the abstract class for instances of these classes. Each `ForeignType` object is a self-initializing class that serves as a constructor method for the appropriate `ForeignObject` class, taking a MACAULAY2 object as input and converting it to the appropriate C type; see, for example, Listing 2. There are a number of built-in instances of `ForeignType`, and users may create additional types for arrays, structs, and unions.

Finally, the `ForeignFunction` class, which gives the package its name, is a subclass of `FunctionClosure`. Instances of this class convert their input from MACAULAY2 objects to the corresponding

```
i3 : int 5
o3 = 5
o3 : ForeignObject of type int32
i4 : double pi
o4 = 3.14159265358979
o4 : ForeignObject of type double
i5 : charstar "Hello, world!"
o5 = Hello, world!
o5 : ForeignObject of type char*
```

Listing 2. Each instance of a `ForeignType` is a `ForeignObject`.

```
i6 : puts = foreignFunction("puts", int, charstar)
o6 = puts
o6 : ForeignFunction
i7 : puts "Hello, world!"
Hello, world!
o7 = 14
o7 : ForeignObject of type int32
```

Listing 3. Printing "Hello, world!" with `puts` as a `ForeignFunction`.

C objects (using the appropriate `ForeignType` classes), call a C function, and then return their output as a `ForeignObject`. `ForeignFunction` objects are created using the `foreignFunction` method, which takes a `SharedLibrary` object (this is optional if the C function is available in shared library MACAULAY2 is already linked against), a string with the name of the function, the output type, and input type (or a list of input types). See Listing 3 for an example `ForeignFunction` object wrapping the `puts` function from the C standard library for outputting strings. It is possible to call functions from C++ libraries as well, but their names are often mangled by the compiler, so this must be accounted for.

3. IMPLEMENTATION DETAILS. Several additions to the MACAULAY2 interpreter were made to support the `ForeignFunctions` package, which we describe in this section.

First, `Pointer`, a new subclass of `Thing`, was added to represent pointers to memory. Although users of the package often do not need to deal with these objects directly, they are fundamental to the inner workings of the package. For example, all instances of the new classes mentioned in Section 2 are basic lists or hash tables containing `Pointer` objects.

Next, a number of low-level routines for converting MACAULAY2 objects into pointers to C objects and back again were added. In particular, MACAULAY2 integers, which belong to the class `ZZ`, use GMP, the GNU Multiple Precision Arithmetic Library [10]. Functions were added to convert these to and from pointers to 8-bit, 16-bit, 32-bit, and 64-bit signed and unsigned integers. Similarly, MACAULAY2 reals, or `RR` objects, use MPFR, the GNU Multiple Precision Floating-Point Reliable Library [8]. Functions were added to convert these to and from pointers to single- and double-precision floating point numbers (`float` and `double` objects, respectively). The constructor methods for each `ForeignType` object use these functions to convert from MACAULAY2 objects to pointers to C objects, and the `value(ForeignObject)` method uses them to convert back.

Everything is then driven by the LIBFFI library [12]. When a `ForeignFunction` object is created using the `foreignFunction` constructor method, the function is found using the C standard library function `dlsym` and an `ffi_cif` struct is created with information about the function to be called, its input types, and its output types. Then when it is called, its inputs are converted to pointers to C objects, and `ffi_call` is called to actually call the function.

Much of this code may be found in the file `M2/Macaulay2/d/ffi.d` at the MACAULAY2 GitHub repository (https://github.com/Macaulay2/M2).

4. FAST FOURIER TRANSFORM EXAMPLE. Consider $\boldsymbol{a} = (a_0, \ldots, a_n) \in \mathbb{C}^{n+1}$ and let

$$f = \sum_{k=0}^{n} a_k x^k \in \mathbb{C}[x].$$

Recall that the *discrete Fourier transform* of $\boldsymbol{a}$ is $\mathscr{F}\{\boldsymbol{a}\} = (f(1), f(\omega^{-1}), \ldots, f(\omega^{-n}))$ and the *inverse discrete Fourier transform* of $\boldsymbol{a}$ is $\mathscr{F}^{-1}\{\boldsymbol{a}\} = (f(1), f(\omega), \ldots, f(\omega^n))$, where $\omega = \exp(2\pi i/(n+1))$ is a principal $(n+1)$-th root of unity. In particular, $\mathscr{F}\{\mathscr{F}^{-1}\{\boldsymbol{a}\}\} = \mathscr{F}^{-1}\{\mathscr{F}\{\boldsymbol{a}\}\} = (n+1)\boldsymbol{a}$. Naïvely, computing a discrete Fourier transform or its inverse using their definitions has time complexity $O(n^2)$, but the famous divide-and-conquer *fast Fourier transform* (FFT) algorithm of Cooley and Tukey [6] has time complexity of only $O(n \log n)$.

A well-known implementation of the FFT algorithm is the C library FFTW [9]. Using the `ForeignFunctions` package, it is possible to create wrappers for its functions in MACAULAY2; see Listing 4. Note that FFTW works with arrays of `fftw_complex` objects, but each of these is just an array containing

```
needsPackage "ForeignFunctions"

libfftw = openSharedLibrary "fftw3"
fftwPlanDft1d = foreignFunction(libfftw, "fftw_plan_dft_1d", voidstar,
   int, voidstar, voidstar, int, uint)
fftwExecute = foreignFunction(libfftw, "fftw_execute", void, voidstar)
fftwDestroyPlan = foreignFunction(libfftw, "fftw_destroy_plan", void, voidstar)

fftHelper = (x, sign) -> (
   n := #x;
   inptr := getMemory(2 * n * size double);
   outptr := getMemory(2 * n * size double);
   p := fftwPlanDft1d(n, inptr, outptr, sign, 64);
   registerFinalizer(p, fftwDestroyPlan);
   dbls := splice apply(x, y -> (realPart numeric y, imaginaryPart numeric y));
   apply(2 * n, i -> *(value inptr + i * size double) = double dbls#i);
   fftwExecute p;
   r := ((2 * n) * double) outptr;
   apply(n, i -> value r_(2 * i) + ii * value r_(2 * i + 1)))

fastFourierTransform = method()
fastFourierTransform List := x -> fftHelper(x, -1)

inverseFastFourierTransform = method()
inverseFastFourierTransform List := x -> fftHelper(x, 1)
```

Listing 4. MACAULAY2 wrappers around FFTW functions.

```
fftMultiply = (f, g) -> (
    m := first degree f;
    n := first degree g;
    a := fastFourierTransform splice append(
        apply(m + 1, i -> coefficient(x^i, f)), n:0);
    b := fastFourierTransform splice append(
        apply(n + 1, i -> coefficient(x^i, g)), m:0);
    c := inverseFastFourierTransform apply(a, b, times);
    sum(m + n + 1, i -> c#i * x^i)/(m + n + 1))
```

Listing 5. MACAULAY2 implementation of FFT polynomial multiplication.

two `double`'s. So some time is spent unpacking the real and imaginary parts from a list of n `CC` objects (MACAULAY2's complex number type) into a list of $2n$ `RR` objects that can be converted into an array of `double`'s for use by FFTW.

One application of the FFT algorithm of possible interest to MACAULAY2 users is $O(n \log n)$ polynomial multiplication. Consider $f = \sum_{k=0}^{m} a_k x^k$, $g = \sum_{k=0}^{n} b_k x^k \in \mathbb{C}[x]$ and let $\boldsymbol{a} = (a_0, \ldots, a_m, 0, \ldots, 0)$ and $\boldsymbol{b} = (b_0, \ldots, b_n, 0, \ldots, 0)$ be $(m+n+1)$-tuples. Define $\boldsymbol{c} = \mathscr{F}\{\boldsymbol{a}\} \cdot \mathscr{F}\{\boldsymbol{b}\}$ using componentwise multiplication, and then it follows that $1/(m+n+1)\mathscr{F}^{-1}\{\boldsymbol{c}\}$ contains the coefficients of fg, see [7, Chapter 30] for more details and Listing 5 for a MACAULAY2 implementation using the FFTW wrappers defined above.

The MACAULAY2 engine multiplies polynomials using the naïve $O(n^2)$ algorithm, multiplying each pair of monomials and then summing the results. However, it is written in C++, and for small polynomials it is superior to `fftMultiply`, which has the overhead of the top-level MACAULAY2 language and calls to LIBFFI and FFTW. But for larger polynomials, `fftMultiply` can be faster. For example, in Listing 6, we see that it computed the product of two degree 6000 polynomials over a second more quickly than MACAULAY2's native polynomial multiplication on a system with an AMD Ryzen 5 2600 processor running Ubuntu 22.04.

```
i12 : R = CC[x];
i13 : n = 6000;
i14 : f = (random(CC^1, CC^(n + 1)) * matrix apply(n + 1, i -> x^i))_(0,0);
i15 : g = (random(CC^1, CC^(n + 1)) * matrix apply(n + 1, i -> x^i))_(0,0);
i16 : elapsedTime f * g;
 -- 16.5132 seconds elapsed
 i17 : elapsedTime fftMultiply(f, g);
 -- 15.2714 seconds elapsed
```

Listing 6. Polynomial multiplication running time comparisons.

5. GENERAL GAUSS–MARKOV LINEAR MODEL PROBLEM EXAMPLE. Statisticians consider the *general Gauss–Markov linear model*

$$\boldsymbol{d} = A\boldsymbol{\beta} + \boldsymbol{\varepsilon},$$

with response vector $\boldsymbol{d} \in \mathbb{R}^n$, $n \times m$ design matrix A, parameter vector $\boldsymbol{\beta} \in \mathbb{R}^m$, and noise vector $\boldsymbol{\varepsilon} \in \mathbb{R}^n$ with mean $\boldsymbol{\Gamma}$ and covariance matrix $\sigma^2 W$, where $\sigma^2 \in \mathbb{R}$ is unknown and W is a known $n \times n$ symmetric nonnegative definite matrix of rank p. As shown in [13], a *best linear unbiased estimator* (BLUE) of $\boldsymbol{\beta}$ is the vector $\boldsymbol{x} \in \mathbb{R}^m$ that, alongside $\boldsymbol{y} \in \mathbb{R}^p$, minimizes $\|\boldsymbol{y}\|$ (using the usual L^2-norm) subject to the constraint $\boldsymbol{d} = A\boldsymbol{x} + B\boldsymbol{y}$, where B is an $n \times p$ matrix of full rank for which $W = BB^T$.

Constrained least squares problems such as this can be solved using the function `dggglm` from the linear algebra library LAPACK [1]. MACAULAY2 already uses a number of LAPACK functions for its linear algebra computations. However, `dggglm` is not one of these.

Using the `ForeignFunctions` package, it is possible to call this function from MACAULAY2; see Listing 7 for the code. Note the helper method `toLAPACK`, which converts MACAULAY2 matrices and

```
needsPackage "ForeignFunctions"

toLAPACK = method()
toLAPACK Matrix := A -> (
    T := (numRows A * numColumns A) * double;
    T flatten entries transpose A)
toLAPACK Vector := toLAPACK @@ matrix

dggglm = foreignFunction("dggglm_", void, toList(13:voidstar))

generalLinearModel= method()
generalLinearModel(Matrix, Matrix, Vector) := (A, B, d) -> (
    if numRows A != numRows B
    then error "expected first two arguments to have the same number of rows";
    n := numRows A;
    m := numColumns A;
    p := numColumns B;
    x := getMemory(m * size double);
    y := getMemory(p * size double);
    lwork := n + m + p;
    work := getMemory(lwork * size double);
    i := getMemory int;
    dggglm(address int n, address int m, address int p, toLAPACK A,
        address int n, toLAPACK B, address int n, toLAPACK d, x, y, work,
        address int lwork, i);
    if value(int * i) != 0 then error("call to dggglm failed");
    (vector value (m * double) x, vector value (p * double) y))
```

Listing 7. MACAULAY2 wrapper around `dggglm` from LAPACK.

```
i2 : A = matrix 1, 2, 3, 4, 1, 2, 5, 6, 7, 3, 4, 6;
                4      3
o2 : Matrix ZZ  <-- ZZ
i3 : B = matrix 1, 0, 0, 0, 2, 3, 0, 0, 4, 5, 1e-5, 0, 7, 8, 9, 10;
                4      4
o3 : Matrix RR   <-- RR
             53        53
i4 : d = vector 1, 2, 3, 4;
      4
o4 : ZZ
i5 : generalLinearModel(A, B, d)
o5 = (|   .3141  |, | .0296627 |)
      | -.334417 |  | .0451036 |
      |  .441691 |  | .0585128 |
                    | .0650142 |
o5 : Sequence
```

Listing 8. Example of general Gauss–Markov linear model problem using LAPACK.

vectors, which are stored using row-major order, into arrays in column-major order as expected by LAPACK. Note also that since MACAULAY2 is already linked against LAPACK, it is not necessary to call `openSharedLibrary` or to include a shared library object as input to `foreignFunction`.

As an example, we use MACAULAY2, via a foreign function call to LAPACK, to reproduce the computation from [13, Section 4] in Listing 8.

6. JUST-IN-TIME COMPILATION EXAMPLE. Another possible use of the `ForeignFunctions` package is *just-in-time*, or *JIT*, compilation. At runtime, C code can be compiled into a shared library and called from Macaulay2. This can result in significant performance increases.

As an example, we consider the naïve computation of Fibonacci numbers using their usual recurrence relation definition, i.e., for all nonnegative integers n,

$$F_n = \begin{cases} n & \text{if } n < 2, \\ F_{n-1} + F_{n-2} & \text{otherwise.} \end{cases}$$

It is a bad idea in general to use this definition to compute Fibonacci numbers because its running time is $\Theta(\varphi^n)$, where $\varphi = (1+\sqrt{5})/2$ is the golden ratio [7, Section 27.1]. However, it will be useful to illustrate our point.

In Listing 9, we define two functions. The first, `fibonacci1`, is a standard top-level Macaulay2 implementation of the above definition. The second, `fibonacci2`, uses JIT compilation. In particular, it writes a C implementation of the definition to a file, calls GCC [14] to compile this code into a shared library object, opens this shared library, creates a foreign function, calls it, and finally converts the return value from a C `int` to a Macaulay2 `ZZ` object.

```
needsPackage "ForeignFunctions"

fibonacci1 = n -> if n < 2 then n else fibonacci1(n - 1) + fibonacci1(n - 2)

fibonacci2 = n -> (
    dir := temporaryFileName();
    makeDirectory dir;
    dir | "/libfib.c" << ///int fibonacci2(int n)
{
    if (n < 2)
        return n;
    else
        return fibonacci2(n - 1) + fibonacci2(n - 2);
}
/// << close;
    run("gcc -c -fPIC " | dir | "/libfib.c -o " | dir | "/libfib.o");
    run("gcc -shared " | dir | "/libfib.o -o " | dir | "/libfib.so");
    lib := openSharedLibrary("libfib", FileName => dir | "/libfib.so");
    f = foreignFunction(lib, "fibonacci2", int, int);
    value f n)
```

Listing 9. Definitions of Macaulay2 and JIT functions implementing Fibonacci number computation.

```
i2 : elapsedTime fibonacci1 35
 -- 10.5189s elapsed
o2 = 9227465
i3 : elapsedTime fibonacci2 35
 -- .129873s elapsed
o3 = 9227465
```

Listing 10. Comparison of Macaulay2 and JIT Fibonacci number computation performances.

As can be seen in Listing 10, despite the additional overhead of compiling a shared library and making a foreign function call, the JIT implementation is significantly quicker, running over 80 times faster than the native Macaulay2 implementation in this particular test.

ACKNOWLEDGEMENTS. We would like to the thank the anonymous referee for helpful comments, and in particular for suggesting the examples in Sections 5 and 6.

SUPPLEMENT. The online supplement contains version 0.4 of `ForeignFunctions`.

REFERENCES.

[1] V. A. Barker, L. S. Blackford, J. Dongarra, J. Du Croz, S. Hammarling, M. Marinova, J. Waśniewski, and P. Yalamov, *LAPACK95 users' guide*, Software, Environments, and Tools **13**, SIAM, Philadelphia, PA, 2001. MR Zbl

[2] D. A. Bini and G. Fiorentino, "Design, analysis, and implementation of a multiprecision polynomial rootfinder", *Numer. Algorithms* **23**:2-3 (2000), 127–173. MR Zbl

[3] D. A. Bini and L. Robol, "Solving secular and polynomial equations: a multiprecision algorithm", *J. Comput. Appl. Math.* **272** (2014), 276–292. MR Zbl

[4] W. Bruns and G. Kämpf, "A Macaulay2 interface for Normaliz", *J. Softw. Algebra Geom.* **2** (2010), 15–19. MR Zbl

[5] W. Bruns, C. S. B. Ichim, and U. von der Ohe, *Normaliz. algorithms for rational cones and affine monoids*, 2024, available at https://normaliz.uos.de.

[6] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series", *Math. Comp.* **19** (1965), 297–301. MR Zbl

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*, 3rd ed., MIT Press, Cambridge, MA, 2009. MR Zbl

[8] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "MPFR: a multiple-precision binary floating-point library with correct rounding", *ACM Trans. Math. Software* **33**:2 (2007), art. id. 13. MR Zbl

[9] M. Frigo and S. Johnson, "The design and implementation of FFTW3", *Proceedings of the IEEE* **93**:2 (2005), 216–231. Zbl

[10] T. Granlund, "GNU multiple precision arithmetic library", electronic reference, 2024, available at https://gmplib.org/.

[11] D. R. Grayson and M. E. Stillman, "Macaulay2, a software system for research in algebraic geometry", 1992, available at https://macaulay2.com/.

[12] A. Green, "libffi, a portable foreign function interface library", electronic reference, 2024, available at https://sourceware.org/libffi.

[13] S. Kourouklis and C. C. Paige, "A constrained least squares approach to the general Gauss–Markov linear model", *J. Amer. Statist. Assoc.* **76**:375 (1981), 620–625. MR Zbl

[14] R. Stallman, "Using GCC: the GNU compiler collection", reference manual, 2003.

RECEIVED: 21 May 2024 REVISED: 10 Dec 2024 ACCEPTED: 17 Dec 2024

DOUGLAS A. TORRANCE:

dtorrance@piedmont.edu

Department of Mathematical Sciences, Piedmont University, Demorest, GA, United States