Julia: A fresh approach to numerical computing

Jeff Bezanson MIT

Alan Edelman MIT

Stefan Karpinski MIT

Viral B. Shah

November 7, 2014

Abstract

The Julia programming language is gaining enormous popularity. Julia was designed to be easy and fast. Most importantly, Julia shatters deeply established notions widely held in the applied community:

1. High-level, dynamic code has to be slow by some sort of law of nature
2. It is sensible to prototype in one language and then recode in another language
3. There are parts of a system for the programmer, and other parts best left untouched as they are built by the experts.

Julia began with a deep understanding of the needs of the scientific programmer and the needs of the computer in mind. Bridging cultures that have often been distant, Julia combines expertise from computer science and computational science creating a new approach to scientific computing.

This note introduces the programmer to the language and the underlying design theory. It invites the reader to rethink the fundamental foundations of numerical computing systems.

In particular, there is the fascinating dance between specialization and abstraction. Specialization allows for custom treatment. We can pick just the right algorithm for the right circumstance and this can happen at runtime based on argument types (code selection via multiple dispatch). Abstraction recognizes what remains the same after differences are stripped away and ignored as irrelevant. The recognition of abstraction allows for code reuse (generic programming). A simple idea that yields incredible power.

The Julia design facilitates this interplay in many explicit and subtle ways for machine performance and, most importantly, human convenience.

Contents

1 Scientific computing languages: The Julia innovation

The original numerical computing language was Fortran, short for "Formula Translating System", released in 1957. Since those early days, scientists have dreamed of writing high-level, generic formulas and having them translated automatically into low-level, efficient machine code, tailored to the particular data types they need to apply the formulas to. Fortran made historic strides towards realization of this dream, and its dominance in so many areas of high-performance computing is a testament to its remarkable success.

The landscape of computing has changed dramatically over the years. Modern scientific computing environments such as MATLAB [13], R [7], Mathematica [12], Octave [15], SciPy [19], and SciLab [5] have grown in popularity and fall under the general category known as *dynamic languages* or *dynamically typed languages*. In these programming languages, programmers write simple, high-level code without any mention of types like `int`, `float` or `double` that pervade *statically typed languages* such as C and Fortran.

Julia is dynamically typed as well, but it is different as it approaches statically typed performance. New users can begin working with Julia as they did in the traditional technical computing languages, and work their way up when ready. In Julia, types are implied by the computation itself together with input values. As a result, Julia programs are often completely generic and compute with data of different input types without modification—a feature known as "polymorphism."

It is a fact that compilers need type information to emit efficient code for high performance. It is also true that the convenience of not typing gives dynamic systems a major advantage in productivity. Accordingly, many researchers today do their day-to-day work in dynamic languages. Still, C and Fortran remain the gold standard for computationally-intensive problems because high-level dynamic languages have lacked performance.

An unfortunate outcome of the currently popular languages is that the most challenging areas of numerical computing have benefited the least from the increased abstraction and productivity offered by higher level languages. A pervasive idea that Julia wishes to shatter is that technical computing languages are "prototyping" languages and one must switch to C or Fortran for performance. The consequences to scientific computing have been more serious than many realize.

We recommend signing up at `juliabox.org` to use Julia conveniently through the Julia prompt or the IJulia notebook. (We like to say "download,next,next,next" is three clicks too many!) If you wish to, you can download Julia at `www.julialang.org`, along with the IJulia notebook.

We wish to show how users can be coaxed into writing better software that is comfortably familiar, easier to work with, easier to maintain, and can perform very well. This paper brings together

- Examples to orient the new user and
- The theory and principles that underly Julia's innovative design

specifically so that everyday users of numerical computing languages may understand why Julia represents a truly fresh approach.

2 A taste of Julia

2.1 A brief tour

Since Julia is a young language, new users may worry that it is somehow immature or lacking some functionality. Our experience shows that Julia is rapidly heading towards maturity. Most new users are surprised to learn that Julia has much of the functionality of the traditional established languages, and a great deal of functionality not found in the older languages.

New users also want a quick explanation as to why Julia is fast, and whether somehow the same "magic dust" could also be sprinkled on their traditional scientific computing language. Why Julia is fast is a combination of many technologies some of which are introduced in Sections 4 through 6 in this paper. Julia is fast because we, the designers, developed it that way for us, the users. Performance is fragile, like accuracy, one arithmetic error can ruin an entire otherwise correct computation. We do not believe that a language can be designed for the human, and retrofitted for the computer. Rather a language must be designed from the start for the human and the computer. Section 3.1 explores this issue.

Before we get into the details of Julia, we provide a brief tour for the reader to get a beginners' feel for the language. Users of other dynamic numerical computing environments may find some of the syntax familiar. In language design, there is always a trade-off between making users feel comfortable in a new language vs. clean language design. In Julia, we maintain superficial similarity to existing environments to ease new users, but not at the cost of sacrificing good language design or performance.

```
In[1]:   A = rand(3,3) + eye(3) # Familiar Syntax
         inv(A)

Out[1]:  3x3 Array{Float64,2}:
          0.698106 -0.393074 -0.0480912
         -0.223584  0.819635 -0.124946
         -0.344861  0.134927  0.601952
```

The output from the Julia prompt says that A is a two dimensional matrix of size 3×3, and contains double precision floating point numbers.

Indexing of arrays is performed with brackets, and is 1-based. It is also possible to compute an entire array expression and then index into it, without assigning the expression to a variable:

```
In[2]:   x = A[1,2]
         y = (A+2I)[3,3] # The [3,3] entry of A+2I

Out[2]:  2.601952
```

In Julia, `I` is a built-in representation of the identity matrix, which can be scaled and combined with other matrix operations, without explicitly forming the identity matrix as is commonly done using commands such as "eye" which unnecessarily requires $O(n^2)$ storage and unnecessary computation in traditional languages and in the end is a cute but ultimately wrong word for the identity matrix. The built-in representation is fully implemented in Julia code in the Julia base library.

Julia has symmetric tridiagonal matrices as a special type. For example, we may define Gil Strang's favorite matrix (the second order difference matrix) in a way that uses only $O(n)$ memory.

Figure 1: Gil Strang's favorite matrix is `strang(n) = SymTridiagonal(2*ones(n),-ones(n-1))` Julia only stores the diagonal and off-diagonal. (Picture taken in Gil Strang's classroom.)

```
In [3]: strang(n) = SymTridiagonal(2*ones(n),-ones(n-1))
        strang(7)
```

```
Out[3]: 7x7 SymTridiagonal{Float64}:
          2.0  -1.0   0.0   0.0   0.0   0.0   0.0
         -1.0   2.0  -1.0   0.0   0.0   0.0   0.0
          0.0  -1.0   2.0  -1.0   0.0   0.0   0.0
          0.0   0.0  -1.0   2.0  -1.0   0.0   0.0
          0.0   0.0   0.0  -1.0   2.0  -1.0   0.0
          0.0   0.0   0.0   0.0  -1.0   2.0  -1.0
          0.0   0.0   0.0   0.0   0.0  -1.0   2.0
```

Julia calls an efficient $O(n)$ tridiagonal solver:

```
In [4]: strang(8)\ones(8)
```

```
Out[4]: 8-element Array{Float64,1}:
          4.0
          7.0
          9.0
         10.0
         10.0
          9.0
          7.0
          4.0
```

Consider the sorting of complex numbers. Sometimes it is handy to have a sort that generalizes the real sort. This can be done by sorting first by the real part, and where there are ties, sort by the imaginary part. Other times it is handy to use the polar representation, which sorts by radius than angle.

If a technical computing language "hard-wires" its sort to be one or the other, it misses a wonderful opportunity. A sorting algorithm need not depend on details of what is being compared or how it is being compared. One can abstract away these details thereby reusing a sorting algorithm for many different situations. One can specialize later. Thus alphabetizing strings, sorting real numbers, or sorting complex numbers in two or more ways all run with the same code.

In Julia, one can turn a complex number `w` into an ordered pair of real numbers (a tuple of length 2) such as the Cartesian form `(real(w),imag(w))` or the polar form `(abs(w),angle(w))`. Tuples are then compared lexicographically in Julia. The sort command takes an optional "less-than" operator, `lt`, which is used to compare elements when sorting.

```
In [5]: # Cartesian comparison sort of complex numbers
        complex_compare1(w,z) = (real(w),imag(w)) < (real(z),imag(z))
        sort([-2,2,-1,im,1], lt = complex_compare1 )
```

```
Out[5]: 5-element Array{Complex{Int64},1}:
        -2+0im
        -1+0im
         0+1im
         1+0im
         2+0im
```

In[6]:
```
# Polar comparison sort of complex numbers
complex_compare2(w,z) = (abs(w),angle(w)) < (abs(z),angle(z))
sort([-2,2,-1,im,1], lt = complex_compare2)
```

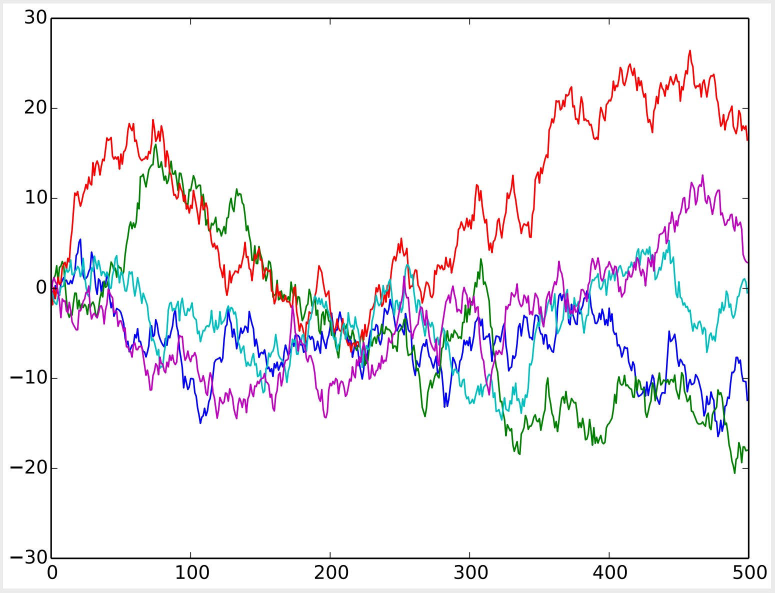
Out[6]:
```
5-element Array{Complex{Int64},1}:
  1+0im
  0+1im
 -1+0im
  2+0im
 -2+0im
```

Note the `Array{ElementType,dims}` syntax. It shows up everywhere. In the above example, the elements are complex numbers whose parts are `Int64`'s. The `1` indicates it is a one dimensional vector.

The last example that we have chosen for the introductory taste of Julia is a quick plot of Brownian motion:

In[7]:
```
Pkg.add("PyPlot") # Download the PyPlot package
using PyPlot # load the functionality into Julia

for i=1:5
    y=cumsum(randn(500))
    plot(y)
end
```



Julia has been in development since 2009; a public release was announced in February of 2012. It is an active open source project with over 250 contributors and is available under the MIT License [10] for open source software. Nurtured at the Massachusetts Institute of Technology, but with contributors from around the world, Julia is increasingly seen as a high-performance alternative to R, Matlab, Octave, SciPy and SciLab, or a high-productivity alternative to C, C++ and Fortran. It is also recognized as being better suited to general purpose computing tasks than traditional technical computing systems, allowing it to be used not only to prototype numerical algorithms, but also to deploy those algorithms, and even serve results of numerical computations to the rest of the world.[1] Perhaps most significantly, a rapidly growing ecosystem of high-quality, open source, composable numerical packages (over 300 in number) written in Julia has emerged, including libraries for linear algebra, statistics, optimization, data analysis, machine learning and many other applications of numerical computing.

2.2 An invaluable tool for numerical integrity

One popular feature of Julia is that it gives the user the ability to "kick the tires" of a numerical computation. We thank Velvel Kahan for the sage advice[2] concerning the importance of this feature.

The idea is simple: a good engineer tests his or her code for numerical stability. In Julia this can be done by changing IEEE rounding modes. There are five modes to choose from, yet most engineers silently only choose the `RoundNearest` mode default available in many technical computing systems. If a difference is detected, one can also run the computation in higher precision.

We round a 15x15 Hilbert-like matrix, and take the [1,1] entry of the inverse computed in various round off modes. The radically different answers dramatically indicates the numerical sensitivity to roundoff. We even noticed that slight changes to LAPACK give radically different answers. Very likely you will see different numbers when you run this code due to the very high sensitivity to roundoff errors.

```
In[8]:  h(n)=[1/(i+j+1) for i=1:n,j=1:n]

Out[8]: h (generic function with 1 method)
```

```
In[9]:  H=h(15);
        with_rounding(Float64,RoundNearest) do
            inv(H)[1,1]
        end

Out[9]: 154410.55589294434
```

```
In[10]:  with_rounding(Float64,RoundUp) do
             inv(H)[1,1]
         end

Out[10]: -49499.606132507324
```

```
In[11]:  with_rounding(Float64,RoundDown) do
             inv(H)[1,1]
         end

Out[11]: -841819.4371948242
```

With 300 bits of precision we obtain

[1]Sudoku as a service, by Iain Dunning, `http://iaindunning.com/2013/sudoku-as-a-service.html`, is a wonderful first example where a Sudoku puzzle is solved using the optimization capabilities of the JUMP.jl Julia package and made available as a web service.

[2]Personal communication, January 2013, in the Kahan home, Berkeley, California

In[12]:
```
with bigfloat_precision(300) do
    inv(big(H))[1,1]
end
```

Out[12]:
```
-2.0939717925074627012828017421448951616270885770371495976323268904715
3507658824910549983762 52e+03
```

Note this is the [1,1] entry of the inverse of the rounded Hilbert-like matrix, not the inverse of the exact Hilbert-like matrix. Also, the results are senstive to the BLAS and LAPACK, and the results may differ on different machines with different versions of Julia.

2.3 Julia architecture and language design philosophy

Many popular dynamic languages were not designed with the goal of high performance. After all, if you wanted really good performance you would use a static language, or so the popular wisdom would say. Only with the increased need in the day-to-day life of scientific programmers for simultaneous productivity and performance in a single system has the need for high-performance dynamic languages become pressing. Unfortunately, retrofitting an existing slow dynamic language for high performance is almost impossible *specifically* in numerical computing ecosystems. This is because numerical computing requires performance-critical numerical libraries, which invariably depend on the details of the internal implementation of the high-level language, thereby locking in those internal implementation details. The best path to a fast, high-level system for scientific and numerical computing is to make the system fast enough that all of its libraries can be written in the high-level language in the first place. The JUMP.jl library [11] for mathematical optimization written by Miles Lubin and Iain Dunning is a great example of the success of this approach—the entire library is written in Julia and uses many language features such as metaprogramming, user-defined parametric types, and multiple dispatch extensively to provide a seamless interface to describe various kinds of optimization problems and solve them with any number of commercially or freely available solvers.

This is an essential part of the design philosophy of Julia: all basic functionality must be possible to implement in Julia—never force the programmer to resort to using C or Fortran. In particular, basic functionality must be fast: integer arithmetic, for loops, recursion, floating-point operations, calling C functions, manipulating C-like structs. While these are not only important for numerical programs, without them, you certainly cannot write fast numerical code. “Vectorization languages” like Matlab, R, and Python+NumPy hide their for loops and integer operations, but they are still there, in C and Fortran, lurking behind just a thin veneer. Julia removes this separation entirely, allowing high-level code to “just write a for loop” if that happens to be the best way to solve a problem.

We believe that the Julia programming language fulfills much of the Fortran dream: automatic translation of formulas into efficient executable code. It allows programmers to write clear, high-level, generic and abstract code that closely resembles mathematical formulas, as they have grown accustomed to in dynamic systems, yet produces fast, low-level machine code that has traditionally only been generated by static languages.

Julia’s ability to combine these levels of performance and productivity in a single language stems from the choice of a number of features that work well with each other:

1. An expressive parametric type system, allowing optional type annotations; (See Section 3.3 for parametric types and Section 4.1 for optional type annotations.)
2. Multiple dispatch using those types to select implementations (Section 4);
3. A dynamic dataflow type inference algorithm allowing types of most expressions to be inferred (Section 3.5);
4. Careful design of the language and standard library to be amenable to type analysis (Section 6);

5. Aggressive code specialization against run-time types (Section 5);
6. Metaprogramming for sophisticated code generation (Section 5);
7. Just-In-Time (JIT) compilation using the Low-level Virtual Machine (LLVM) compiler framework [9], which is also used by a number of other compilers such as Clang [3] and Apple's Swift [18] (Section 5).

Although a sophisticated type system is made available to the programmer, it remains unobtrusive in the sense that one is never required to specify types, nor are type annotations necessary for performance. Type information flows naturally from having actual values (and hence types) during code generation, and from the multiple dispatch paradigm: by expressing polymorphic behavior of functions declaratively with multiple dispatch, the programmer provides the compiler with extensive type information, while also enjoying increased expressiveness.

In what follows, we describe the benefits of Julia's language design for numerical computing, allowing programmers to more readily express themselves and obtain performance at the same time.

3 Writing programs with and without types

3.1 The balance between human and the computer

Graydon Hoare, author of the Rust programming language [17], in an essay on "Interactive Scientific Computing" [6] defined programming languages succinctly:

> Programming languages are mediating devices, interfaces that try to strike a balance between human needs and computer needs. Implicit in that is the assumption that human and computer needs are equally important, or need mediating.

Catering to the needs of both the human and the computer is a repeating theme in this paper. A program consists of data and operations on data. Data is not just the input file, but everything that is held —an array, a list, a graph, a constant—during the life of the program. The more the computer knows about this data, the better it is at executing operations on that data. Types are exactly this metadata. Describing this metadata, the types, takes real effort for the human. Statically typed languages such as C and Fortran are at one extreme, where all types must be defined and are statically checked during the compilation phase. The result is excellent performance. Dynamically typed languages dispense with type definitions, which leads to greater productivity, but lower performance as the compiler and the runtime cannot benefit from the type information that is essential to produce fast code. Can we strike a balance between the human's preference to avoid types and the computer's need to know?

3.2 Julia's recognizable types

Julia's design allows for gradual learning of concepts, where users start in a manner that is familiar to them and over time, learn to structure programs in the "Julian way" (a term that captures well-structured readable high performance Julia code). Julia users coming from other numerical computing environments have a notion that data may be represented as matrices that may be dense, sparse, symmetric, triangular, or of some other kind. They may also, though not always, know that elements in these data structures may be single precision floating point numbers, double precision, or integers of a specific width. In more general cases, the elements within data structures may be other data structures. We introduce Julia's type system using matrices and their number types:

```
In[13]:  rand(1,2,1)
```

```
Out[13]: 1x2x1 Array{Float64,3}:
         [ :, :, 1] =
           0.789166 0.652002
```

```
In[14]:  [1 2; 3 4]
```

```
Out[14]: 2x2 Array{Int64,2}:
           1 2
           3 4
```

```
In[15]:  [true; false]
```

```
Out[15]: 2-element Array{Bool,1}:
           true
           false
```

We see a pattern in the examples above. `Array{T,ndims}` is the general form of the type of a dense array with `ndims` dimensions, whose elements themselves have a specific type `T`, which is of type double precision floating point in the first example, a 64-bit signed integer in the second, and a boolean in the third example. Therefore `Array{T,1}` is a 1-d vector (first class objects in Julia) with element type `T` and `Array{T,2}` is the type for 2-d matrices.

It is useful to think of arrays as a generic N-d object that may contain elements of any type `T`. Thus `T` is a type parameter for an array that can take on many different values. Similarly, the dimensionality of the array `ndims` is also a parameter for the array type. This generality makes it possible to create arrays of arrays. For example, Using Julia's array comprehension syntax, we create a 2-element vector containing 2×2 identity matrices.

```
In[16]:  a = [eye(2) for i=1:2]
```

```
Out[16]: 2-element Array{Array{Float64,2},1}:
```

Many first time users are already familiar with basic floating point number types such as `Float32` (single precision) and `Float64` (double precision). In addition, Julia also provides a built-in `BigFloat` type that may interest a new Julia user. With `BigFloats`, users can run computations in arbitrary precision arithmetic, thereby exploring the numerical properties of their computation.

3.3 User's own types are first class too

Many dynamic languages for numerical computing have traditionally had an asymmetry, where built-in types have much higher performance than any user-defined types. This is not the case with Julia, where there is no meaningful distinction between user-defined and "built-in" types.

We have mentioned so far a few number types and two matrix types, `Array{T,2}` the dense array, with element type `T` and `SymTridiagonal{T}`, the symmetric tridiagonal with element type `T`. There are also other matrix types, for other structures including SparseMatrixCSC (Compressed Sparse Columns), Hermitian, Triangular, Bidiagonal, and Diagonal. Julia's sparse matrix type has an added flexibility that it can go beyond storing just numbers as nonzeros, and instead store any other Julia type as well. The indices in SparseMatrixCSC can also be represented as integers of any width (16-bit, 32-bit or 64-bit). All these different matrix types, although available as built-in types to a user downloading Julia, are implemented completely in Julia, and are in no way any more or less special than any other types one may define in their own program.

For demonstration, we create a symmetric arrow matrix type that contains a diagonal and the first row `A[1,2:n]`.

In[17]:
```
# Parametric Type Example (Parameter T)
# Define a Symmetric Arrow Matrix Type with elements of type T

type SymArrow{T}
    dv::Vector{T} # diagonal
    ev::Vector{T} # 1st row[2:n]
end

# Create your first Symmetric Arrow Matrix
S = SymArrow([1,2,3,4,5],[6,7,8,9])
```

Out[17]:
```
SymArrow{Int64}([1,2,3,4,5],[6,7,8,9])
```

Our purpose here is to introduces “Parametric Types”. Later in Section 4.4.2, we develop this example much further. The `SymArrow` matrix type contains two vectors, one each for the diagonal and the first row, and these vector contain elements of type `T`. In the type definition, the type `SymArrow` is parametrized by the type of the storage element `T`. By doing so, we have created a generic type, which refers to a universe of all arrow matrices containing elements of all types. The matrix `S`, is an example where `T` is `Int64`. When we write functions in Section 4.4.2 that operate on arrow matrices, those functions themselves will be generic and applicable to the entire universe of arrow matrices we have defined here.

Julia’s type system allows for abstract types, concrete “bits” types, composite types, and immutable composite types. All of these can be parametric and users may even write programs using unions of these different types. We refer the reader to read all about Julia’s type system in the types chapter in the Julia manual[3].

3.4 Vectorization: Key Strengths and Serious Weaknesses

Users of traditional high level computing languages know that vectorization improves performance. Do most users know exactly why vectorization is so useful? It is precisely because, by vectorizing, the user has promised the computer that the type of an entire vector of data matches the very first element. This is an example where users are willing to provide type information to the computer without even knowing exactly that is what they are doing. Hence, it is an example of a strategy that balances the computer’s needs with the human’s.

From the computer’s viewpoint, vectorization means that operations on data happen largely in sections of the code where types are known to the runtime system. When the runtime is operating on arrays, it has no idea about the data contained in an array until it encounters the array. Once encountered, the type of the data within the array is known, and this knowledge is used to execute an appropriate high performance kernel. Of course what really occurs at runtime is that the system figures out the type, and gets to reuse that information through the length of the array. As long as the array is not too small, all the extra work in gathering type information and acting upon it at runtime is amortized over the entire operation.

The downside of this approach is that the user can achieve high performance only with built-in types, and user defined types end up being dramatically slower. The restructuring for vectorization is often unnatural, and at times not possible. We illustrate this with an example of the cumulative

[3] See the chapter on types in the Julia manual: `http://docs.julialang.org/en/latest/manual/types/`

sum computation.

```
In[18]: # Sum prefix (cumsum) on vector w with elements of type T
        function prefix{T}(w::Vector{T})
            for i=2:size(w,1)
               w[i]+=w[i-1]
            end
            return w
        end
```

We execute this code on a vector of double precision numbers and double-precision complex numbers and observe something that may seem remarkable: similar running times.

```
In[19]: x = ones(1_000_000)
        @time prefix(x)

        y = ones(1_000_000) + im*ones(1_000_000)
        @time prefix(y);
```

```
Out[19]: elapsed time:  0.003243692 seconds (80 bytes allocated)
         elapsed time:  0.003290693 seconds (80 bytes allocated)
```

This simple example is difficult to vectorize, and hence is often built into many numerical computing systems. In Julia, the implementation is very similar to the snippet of code above, and runs at speeds similar to C. While Julia users can write vectorized programs like in any other dynamic language, vectorization is not a pre-requisite for performance. This is because Julia strikes a different balance between the human and the computer when it comes to specifying types. Julia allows optional type annotations. If the programmer annotates their program with types, the Julia compiler will use that information. But for the most part, user code often includes minimal or no type annotations, and the Julia compiler automatically infers the types.

3.5 Type inference rescues "for loops" and so much more

A key component of Julia's ability to combine performance with productivity in a single language is its implementation of dynamic dataflow type inference [14],[8],[2]. Unlike type inference algorithms for static languages, this algorithm is tailored to the way dynamic languages work: the typing of code is determined by the flow of data through it. The algorithm works by walking through a program, starting with the types of its input values, and "abstractly interpreting" it: instead of applying the code to values, it applies the code to types, following all branches concurrently and tracking all possible states the program could be in, including all the types each expression could assume.

Since programs can iterate arbitrarily long and recur to arbitrary depths, this process must be iterated until it reaches a fixed point. The design of the type lattice used ensures that the process terminates. Once that point is reached, the program is annotated with upper bounds on the types that each variable and expression can assume.

The dynamic dataflow type inference algorithm allows programs to be automatically annotated with type bounds without forcing the programmer to explicitly specify types. Yet, in dynamic languages it is possible to write programs which inherently cannot be concretely typed. In such cases, dataflow type inference provides what bounds it can, but these may be trivial and useless—i.e. they may not narrow down the set of possible types for an expression at all. However, the design of Julia's programming model and standard library are such that a majority of expressions in typical programs *can* be concretely typed. Moreover, there is a positive correlation between the ability to concretely type code and that code being performance-critical.

This type system supports productive interactions between the programmer and compiler. For example, in mathematical environments users often want a square root function defined for reals that returns either a real or complex number depending on the sign of the argument. In Julia this can be defined as

```
In[20]:  # Define xsqrt for real inputs
         # return sqrt(complex(x)) if x<0
         xsqrt(x::Real) = x < 0 ?  sqrt(complex(x)) :  sqrt(x)
```

This definition works, but will not perform as well as a real-only `sqrt` due to the overhead of tracking whether the result is real or complex. However, in a particular use the programmer might know that a real result is expected, and annotate the call as `xsqrt(y)::Real`. If at runtime, the result is not real, an error is generated.

`Real` is an abstract type, so this code is still generic and works for any real numeric type. (For example, a real integer or a real float or a real rational.) The compiler will combine this declaration with the inferred type of `xsqrt(y)`, recovering exact type information as a result.

A lesson of the numerical computing languages is that one must learn to vectorize to get performance. The mantra is "for loops" are bad, vectorization is good. Indeed one can find the mantra on p.72 of the "1998 Getting Started with Matlab manual" (and other editions):

> Experienced Matlab users like to say "Life is too short to spend writing for loops."

It is not that "for loops" are inherently slow by themselves. The slowness comes from the fact that in the case of most dynamic languages, the system does not have access to the types of the various variables within a loop. Since programs often spend much of their time doing repeated computations, the slowness of a particular operation due to lack of type information is magnified inside a loop. This leads to users often talking about "slow for loops" or "loop overhead".

Consider the example of the cumulative sum earlier. The function definition tells the compiler that the input will be a vector, containing elements of the same type. Within the loop, the compiler is able to infer that the input vector is indexed with scalar integers and the partial sums are stored in the same array as the computation progresses. The sums are computed using the "+" operator. The correct version of `+` is chosen by the system from the 123 `+` methods that are available in Julia at the time of this writing, which define how various types may be added — combinations of integers, floating point numbers, rational numbers, dense arrays, sparse matrices, etc.

```
In[21]:  +

Out[21]:  + (generic function with 123 methods)
```

In statically typed languages, full type information is always available at compile time, allowing compilation of a loop into a few machine instructions. This is not the case in most dynamic languages, where the types are discovered at run time, and the cost of determining the types and selecting the right operation can run into hundreds or thousands of instructions.

Julia has a transparent performance model. For example a `Vector{Float64}` as in our example here, always has the same in-memory representation as it would in C or Fortran; one can take a pointer to the first array element and pass it to a C library function using `ccall` and it will just work. The programmer knows exactly how the data is represented and can reason about it. They know that a `Vector{Float64}` does not require any additional heap allocation besides the `Float64` values and that arithmetic operations on these values will be machine arithmetic operations. In the case of say, `Complex128`, Julia stores complex numbers in the same way as C or Fortran. Thus complex arrays are actually arrays of complex values, where the real and imaginary values are stored consecutively. Some systems have taken the path of storing the real and imaginary parts separately, which leads to some convenience for the user, at the cost of performance and interoperability. With the `immutable` keyword, a programmer can also define immutable data types, and enjoy the same benefits of performance for composite types as for the more primitive number types (bits types). This approach is being used to define many interesting data structures such as small arrays of fixed sizes, which can have much higher performance than the more general array data structure.

The transparency of the C data and performance models has been one of the major reasons for C's long-lived success. One of the design goals of Julia is to have similarly transparent data and performance models. With a sophisticated type system and type inference, Julia achieves both.

4 Code Selection

Code selection or code specialization from one point of view is the opposite of code reuse enabled by abstraction. Ironically, viewed another way, it enables abstraction. Julia allows users to overload function names, and select code based on argument types. This can happen at the highest and lowest levels of the software stack. Code specialization lets us optimize for the details of the case at hand. Code abstraction lets calling codes, probably those not yet even written or perhaps not even imagined, work all the way through on structures that may not have been envisioned by the original programmer.

We see this as the ultimate realization of the famous 1908 quip that

Mathematics is the art of giving the same name to different things.

by noted mathematician Henri Poincaré.[4]

In this upcoming section we provide examples of how plus can apply to so many objects. Some examples are floating point numbers, or integers. It can also apply to sparse and dense matrices. Another example is the use of the same name, "det", for determinant, for the very different algorithms that apply to very different matrix structures. The use of overloading not only for single argument functions, but for multiple argument functions is already a powerful abstraction.

4.1 Dispatch by Argument Type

In Julia one can define a function of two arguments without argument types:

```
In[22]:  g(x,y)=sqrt(x^2+y^2)
         # x,y of any type
```

or with argument types. (This is known as optional type annotation.):

```
In[23]:  f(x::Real,    y::Real)    = sqrt(x^2+y^2)
         f(x::Complex,y::Complex) = sqrt(abs(x)^2-abs(y)^2)
```

[4] A few versions of this quote are very relevant to Julia's power of abstractions and technical computing. They are worth pondering:

It is the harmony of the different parts, their symmetry, and their happy adjustment; it is, in a word, all that introduces order, all that gives them unity, that enables us to obtain a clear comprehension of the whole as well as of the parts. Elegance may result from the feeling of surprise caused by the unlooked-for occurrence of objects not habitually associated. In this, again, it is fruitful, since it discloses thus relations that were until then unrecognized. **Mathematics is the art of giving the same names to different things.**

http://www.nieuwarchief.nl/serie5/pdf/naw5-2012-13-3-154.pdf. and

One example has just shown us the importance of terms in mathematics; but I could quote many others. It is hardly possible to believe what economy of thought, as Mach used to say, can be effected by a well-chosen term. I think I have already said somewhere that **mathematics is the art of giving the same name to different things**. It is enough that these things, though differing in matter, should be similar in form, to permit of their being, so to speak, run in the same mould. When language has been well chosen, one is astonished to find that all demonstrations made for a known object apply immediately to many new objects: nothing requires to be changed, not even the terms, since the names have become the same.

`http://www-history.mcs.st-andrews.ac.uk/Extras/Poincare_Future.html`

The function $g(x, y)$ will compute $\sqrt{x^2 + y^2}$ no matter what the type of x or y. By contrast, the function $f(x, y)$ amounts to the separation of cases:

```
if x is Real and y is Real compute sqrt(x^2+y^2}
If x is Complex and y is Complex compute sqrt(abs(x)^2-abs(y)^2)
```

If x and y are not both real or not both complex, then $g(x, y)$ is an error.

The notation

```
In[24]:  # Function definitions for the four possibilities of x,y having Type1/Type2
         f(x::Type1, y::Type1) =  code specialized for both arguments Type1
         f(x::Type1, y::Type2) =  code specialized for arguments Type1, Type2 respectively
         f(x::Type2, y::Type1) =  code specialized for arguments Type2, Type1 respectively
         f(x::Type2, y::Type2) =  code specialized for both arguments Type2
```

is a convenient way of expressing code selection (known as dispatch) based on the types of the first two arguments. Other technical computing libraries might use Case statements, or If/ElseIf constructs. Julia avoids the time that can be wasted during this code selection process which degrades performance.

In Julia, the dispatch by type goes straight to the compiler leading to remarkable efficiencies. These efficiencies can be measured by small numbers of lines of assembler or by short execution times.

We have not seen this in the literature but it seems worthwhile to point out four possibilities:

1. static single dispatch (not done)
2. static multiple dispatch (frequent in static languages, e.g. C++ overloading)
3. dynamic single dispatch (MATLAB's object oriented system might fall in this category though it has its own special characteristics)
4. dynamic multiple dispatch (usually just called multiple dispatch).

In Section 4.3 we discuss the comparison with traditional object oriented approaches. Class-based object oriented programming could reasonably be called dynamic single dispatch, and overloading could reasonably be called static multiple dispatch. Julia's (dynamic!) multiple dispatch approach is more flexible and adaptable while still retaining powerful performance capabilities. Julia programmers often find that dynamic multiple dispatch makes it easier to structure their programs in ways that are closer to the underlying science.

4.2 Code selection from bits to matrices

Julia uses the same mechanism for code selection at all levels, from the top to the bottom. In Sections 4.2.1, 4.2.2, and 4.2.3 we consider code selection problems that have the common general format defined in In[24] above.

f	Function	Operand Types
Low Level "+"	Add Numbers	{Float , Int}
High Level "+"	Add Matrices	{Dense Matrix , Sparse Matrix}
Number/Function "*11	Scale or Compose	{Function , Number }

4.2.1 Summing Numbers: Floats and Ints

We begin at the lowest level. Mathematically, integers are thought of as being special real numbers, but on a computer, an Int and a Float have two very different representations. Ignoring for a moment that there are even many choices of Int and Float representations, if we add two numbers, code selection based on numerical representation is taking place at a very low level. Most users are blissfully unaware of this code selection, because it is hidden somewhere that is usually off-limits to the user. Nonetheless, one can follow the evolution of the high level code all the way down to the assembler level which ultimately would reveal an ADD instruction for integer addition, and, for example, the AVX[5] instruction VADDSD[6] for floating point addition in the language of x86 assembly level instructions. The point being these are ultimately two different algorithms being called, one for a pair of Ints and one for a pair of Floats.

Figure 2 takes a close look at what a computer must do to perform `x+y` depending on whether (x,y) is (Int,Int), (Float,Float), or (Int,Float) respectively. In the first case, an integer add is called, while in the second case a float add is called. In the last case, a promotion of the int to float is called through the x86 instruction VCVTSI2SD[7], and then the float add follows.

It is instructive to build a Julia simulator in Julia itself. Let us define the aforementioned assembler instructions using Julia.

In[25]:
```
# Simulate the assembly level add, vaddsd, and vcvtsi2sd commands

add(x::Int        ,y::Int)     = x+y
vaddsd(x::Float64,y::Float64) = x+y
vcvtsi2sd(x::Int)              = float(x)
```

In[26]:
```
# Simulate Julia's definition of + using ⊕
# To type ⊕, type as in TeX, \oplus and hit the <tab> key
⊕(x::Int,    y::Int)     = add(x,y)
⊕(x::Float64,y::Float64) = vaddsd(x,y)
⊕(x::Int,    y::Float64) = vaddsd(vcvtsi2sd(x),y)
⊕(x::Float64,y::Int)     = y ⊕ x
```

In[27]:
```
methods(⊕)
```

Out[27]:
```
4 methods for generic function ⊕:
⊕ (x::Int64,y::Int64) at In[26]:3
⊕ (x::Float64,y::Float64) at In[26]:4
⊕ (x::Int64,y::Float64) at In[26]:5
⊕ (x::Float64,y::Int64) at In[26]:6
```

4.2.2 Summing Matrices: Dense and Sparse

We now move to a much higher level: matrix addition. The versatile "+" symbol lets us add matrices. Mathematically, sparse matrices are thought of as being special matrices with enough zero entries. On a computer, dense matrices are (usually) contiguous blocks of data with a few parameters attached, while sparse matrices (which may be stored in many ways) require storage of index information one way or another. If we add two matrices, code selection must take place depending on whether the summands are (dense,dense), (dense,sparse), (sparse,dense) or (sparse,sparse).

While this is at a much higher level, the basic pattern is unmistakably the same as that of Section 4.2.1. While including all the bells and whistles is not appropriate in this paper, we can show one way of implementing and algorithms that works with dense matrices if either A or B are dense, but

[5]AVX: Adanced Vector eXtension to the x86 instruction set
[6]VADDSD: Vector ADD Scalar Double-precision
[7]VCVTSI2SD: Vector ConVerT Doubleword (Scalar) Integer to(2) Scalar Double Precision Floating-Point Value

Figure 2: While assembly code may seem intimidating, Julia disassembles readily. Armed with the `code_native` command in Julia and perhaps a good list of assembler commands such as may be found on `http://docs.oracle.com/cd/E36784_01/pdf/E36859.pdf` one can really learn to see the details of code selection in action at the lowest levels.

```
In[28]:  f(a,b) = a + b

Out[28]: f (generic function with 1 method)

In[29]:  # Ints add with the x86 add instruction
         @code_native f(2,3)

Out[29]: push RBP
         mov RBP, RSP
         add RDI, RSI
         mov RAX, RDI
         pop RBP
         ret

In[30]:  # Floats add, for example, with the x86 vaddsd instruction
         @code_native f(1.0,3.0)

Out[30]: push RBP
         mov RBP, RSP
         vaddsd XMM0, XMM0, XMM1
         pop RBP
         ret

In[31]:  # Int + Float requires a convert to scalar double precision, hence
         # the x86 vcvtsi2sd instruction
         @code_native f(1.0,3)

Out[31]: push RBP
         mov RBP, RSP
         vcvtsi2sd XMM1, XMM0, RDI
         vaddsd XMM0, XMM1, XMM0
         pop RBP
         ret
```

uses a sparse algorithm if both are sparse.

In[32]:
```
# Dense + Dense
⊕(A::Matrix, B::Matrix) =
      [A[i,j]+B[i,j] for i in 1:size(A,1),j in 1:size(A,2)]
# Dense + Sparse
⊕(A::Matrix, B::AbstractSparseMatrix) = A ⊕ full(B)
# Sparse + Dense
⊕(A::AbstractSparseMatrix,B::Matrix)  = B ⊕ A
# Sparse + Sparse is best written using the long form function definition:
function ⊕(A::AbstractSparseMatrix, B::AbstractSparseMatrix)
    C=copy(A)
    (i,j)=findn(B)
    for k=1:length(i)
        C[i[k],j[k]]+=B[i[k],j[k]]
    end
    C
end
```

We now have eight methods for ⊕, four for the low level sum, and four more for the high level sum. The most important point is that the mechanism for code selection, this dispatch notation which goes straight to the compiler, is the same.

In[33]:
```
methods(⊕)
```

Out[33]:
```
8 methods for generic function ⊕:
```
a listing of all eight follows: four low level, four matrix level

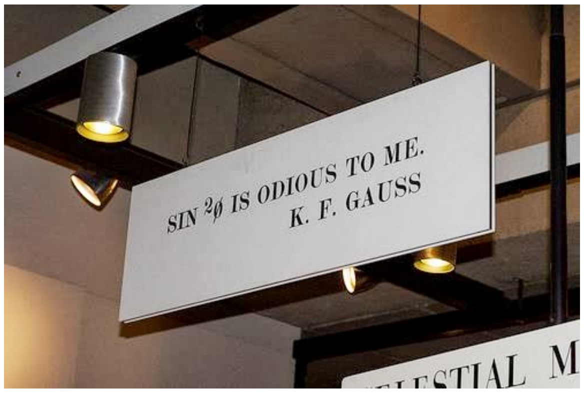


Figure 3: Gauss quote hanging from the ceiling of the Boston Museum of Science Mathematica Exhibit.

4.2.3 Further examples

We have a few further examples of the framework provided by In[24] at the beginning of this section of the paper.

It is possible to model mathematical notations that are often used in print, but are difficult to employ in programs. For example, we can teach the computer some natural ways to multiply numbers and functions. Suppose that a and t are scalars, and f and g are functions, and we wish to define

1. **Number x Function** = **scale output:** $a * g$ is the function that takes x to $a * g(x)$
2. **Function x Number** = **scale argument :** $f * t$ is the function that takes x to $f(tx)$ and
3. **Function x Function** = **composition of functions:** $f * g$ is the function that takes x to $f(g(x))$.

If you are a mathematician who does not program, you would not see the fuss. If you thought how you might implement this in your favorite computer language, you might immediately see the benefit. In Julia, multiple dispatch makes all three uses of `*` easy to express:

In[34]:
```
*(a::Number, g::Function)= x->a*g(x)   # Scale output
*(f::Function,t::Number) = x->f(t*x)   # Scale argument
*(f::Function,g::Function)= x->f(g(x)) # Function composition
```

Here, multiplication is dispatched by the type of its first and second arguments. It goes the usual way if both are numbers, but there are three new ways if one, the other, or both are functions.

These definitions exist as part of a larger system of generic definitions, which can be reused by later definitions. Consider the case of the mathematician Gauss' preference for $\sin^2\phi$ to refer to $\sin(\sin(\phi))$ and not $\sin(\phi)^2$ (writing "$\sin^2(\phi)$ is odious to me, even though Laplace made use of it."(Figure 3).) By defining `*(f::Function,g::Function)= x->f(g(x))`, `(f^2)(x)` automatically computes $f(f(x))$ as Gauss wanted. This is a consequence of a generic definition that evaluates `x^2` as `x*x` no matter how `x*x` is defined.

This paradigm is a natural fit for numerical computing, since so many important operations involve interactions among multiple values or entities. Binary arithmetic operators are obvious examples, but other uses abound. For example,[8] code performing collision detection among different

[8]From `http://assoc.tumblr.com/post/71454527084/cool-things-you-can-do-in-julia`.

shapes can define

```
In[35]: function collide(me::Circle, other::Rectangle)
        function collide(me::Polygon, other::Circle)
        function collide(me::Polygon, other::Rectangle)
```

When the call `collide(self, other)` takes place, the appropriate method is selected based on both arguments. This dispatch is dynamic, based on run-time types, unlike function overloading in C++. This design point is important, since it provides full flexibility — the expected method is called no matter how much run-time variation there is in the types of shape arguments, and independent of whether the compiler can prove anything about those types. Dynamic multiple dispatch encompasses both object-oriented techniques used in general-purpose programming, and the behaviors of mathematical objects compilers do not know how to reason about.

Note this example allows for three shape types: Circle, Rectangle, Polygon. There could thus be nine possible functions total of a function with two arguments and three types for each argument.

4.3 Is "code selection" just traditional object oriented programming?

It is worth crystallizing some key aspects of the code selection as described above.

1. The same name can be used for different functions. (See Footnote 5). (Method or function overloading.)
2. The collection of functions that might be called by the same name is thought of as an entity itself. (Generic functions.)
3. Code selection at the lowest and highest levels use one and the same mechanism
4. Which method is called is based entirely on the types of the arguments of the methods. (This is sometimes the emphasis of the term ad-hoc polymorphism.)
5. The types of the arguments of a method that is being called may or may not be knowable by a static compiler. Instead, the method may be chosen dynamically at runtime as the types become known. (Dynamic Dispatch) (See Figure 4.) In particular, one can recover from a loss of type information.
6. The method is not chosen by only one argument (Single Dispatch) but rather by all the arguments (Multiple Dispatch)
7. Julia is not encumbered by the encapsulation restrictions (class based methods) of most object oriented languages. The generic functions play a more important role than the types. (Some call this "verb" based languages as opposed to most object oriented languages being "noun" based. In numerical computing, it is the concept of "solve $Ax = b$" that often feels more primary, at the highest level, rather than whether the matrix A is full, sparse, or structured.)

Readers familiar with Java might think, so what? One can easily create methods based on the types of the arguments. An example is provided in Figure 4.

```
\* Polymorphic Java Example. Method defined by types of two arguments. *\

public class OverloadedAddable {

   public int    addthem(int i, int f} {
      return i+f;
   }

   public double addthem(int i, double f} {
      return i+f;
   }

   public double addthem(double i, int f} {
      return i+f;
   }

   public double addthem(double i,  double  f} {
      return i+f;
   }

}
```

Figure 4: Advantages of Julia: It is true that the above Java code is polymorphic based on the types of the two arguments. However, in Java if the method `addthem` is called, the types of the arguments must be known at compile time. This is static dispatch. Java is also encumbered by encapsulation: in this case `addthem` is encapsulated inside the `OverloadedAddable` class. While this is considered a safety feature in Java culture, it becomes a terrible burden for technical computing.

However a moment's thought shows that the following dynamic situation in Julia is impossible to express in Java:

In[36]:
```
# It is possible for a static compiler to know that x,y are Float
x = randbool() ?  1.0 :  2.0
y = randbool() ?  1.0 :  2.0
x+y

# It is impossible to know until runtime if x,y are Int or Float
x = randbool() ?  1 :  1.0
y = randbool() ?  1 :  1.0
x+y
```

In Julia as in mathematics, functions are as important as their arguments. Perhaps even more so. We saw in Out[21] that "+" already has 123 methods attached, while Out[8] and Out[28] are functions with one method. In Out[33] we took advantage of Julia's ability to name variables and methods with unicode to create the generic ⊕ with eight methods. We can created a new function `foo` and gave it six definitions depending on the combination of types. In the following example we introduce terms from computer science language research for the benefit of an audience of technical computing programmers.

In[37]:
```
# Define a generic function with 6 methods.  Each method is itself a
# function.  In Julia generic functions are far more convenient than the
# multitude of case statements seen in other languages.  When Julia sees
# foo, it decides which method to use, rather than first seeing and deciding
# based on the type.
foo() = "Empty input"
foo(x::Int) = x
foo(S::String) = length(S)
foo(x::Int, S::String) = "An Int and a String"
foo(x::Float64,y::Float64) = sqrt(x^2+y^2)
foo(a::Any,b::String)= "Something more general than an Int and a String"
# The function name foo is overloaded.  This is an example of polymorphism.
# In the jargon of computer languages this is called ad-hoc polymorphism.
# The multiple dynamic dispatch idea captures the notion that the generic
# function is deciphered dynamically at runtime.  One of the six choices
# will be made or an error will occur.
```

Out[37]:
```
foo (generic function with 6 methods)
```

Any one instance of `foo` is known as a method or function. The collection of six methods is referred to as a **generic function**. Contemplating the Poincaré quote in Footnote 5, it is handy to reason about everything that you are giving the same name. In real life coding, one tends to use the same name when the abstraction makes a great deal of sense. Humans often lose sight it is an abstraction. That we use "+" for ints,floats, dense matrices, and sparse matrices is the same name for different things. Methods are grouped into (generic) functions. A generic function can be applied to several arguments, and the method with the most specific signature matching the arguments is invoked.

Readers familiar with MATLAB may be familiar with MATLAB's single dispatch mechanism. It is unusual in that it is not completely class based, as the code selection is based on MATLAB's own custom hierarchy. In MATLAB the leftmost object has precedence, but user-defined classes have precedence over built-in classes. MATLAB also has a mechanism to create a custom hierarchy.

Julia generally shuns the notion of "built-in" vs. "user-defined" preferring to focus on the method to be performed based on the combination of types, and obtaining high performance as a byproduct. A high level library writer, which we do not distinguish from any user, has to match the best algorithm for the best input structure. A sparse matrix would match to a sparse routine, a dense matrix to a dense routine. A low level language designer has to make sure that integers are added with an integer adder, and floating points are added with a float adder. Despite the very different levels, the reader might recognize that deep down, these are both examples of code being selected to match the structure of the problem.

Readers familiar with object-oriented paradigms such as C++ or Java are most likely familiar with the approach of encapsulating methods inside classes. This is very similar to our single dispatch examples, where we have a `newdet` method for each type of matrix. Julia's more general multiple dispatch mechanism (also known as generic functions, or multi-methods) is a paradigm where methods are defined on combinations of data types (classes) (Figure ??). Julia has proven that this is remarkably well suited for technical computing.

A class based language might express the sum of a sparse matrix with a full matrix as follows: `A_sparse_matrix.plus(A_full_matrix)`. Similarly it might express indexing as `A_sparse_matrix.sub(A_full_matrix)` . If a tridiagonal were added to the system, one has to find the method `plus` or `subsref` which is encapsulated in the sparse matrix class, modify it and test it. Similarly, one has to modify every full matrix method, etc. We believe that class-based methods, which can be taken quite far, are not sufficiently powerful to express the full gamut of abstractions in scientific computing. Further, the burdens of encapsulation create a wall around objects and methods that are counterproductive for technical computing.

The generic function idea captures the notion that a method for a general operation on pairs of matrices may exist (e.g. "+") but if a more specific operation is possible (e.g. "+" on sparse matrices, or "+" on a special matrix structure like Bidiagonal), then the more specific operation is used. We also mention indexing as another example, Why should the indexee take precedence over the index?

4.3.1 Quantifying use of multiple dispatch

In [1] we performed an analysis to substantiate the claim that multiple dispatch, an esoteric idea from computer languages, finds its killer application in scientific computing. We wanted to answer for ourselves the question of whether there was really anything different about how Julia uses multiple dispatch.

Table 4.3.1 gives an answer in terms of Dispatch ratio (DR), Choice ratio (CR). and Degree of specialization (DoS). While multiple dispatch is an idea that has been circulating for some time, its application to technical computing appears to have significantly favorable characteristics compared to previous applications.

4.4 Case Study for Numerical Computing

For a reader wishing to get started with some of these powerful features in Julia we provide an introductory example.

4.4.1 Determinant: Simple Single Dispatch

In traditional technical computing there were people with special skills known as library writers. Most users were, well, just users of libraries. In this case study, we show how anybody can dispatch a new determinant function based solely on the type of the argument.

For triangular and diagonal structures the obvious formulas are used. For general matrices, the programmer will compute QR and use the diagonal elements of R.[9] For symmetric tridiagonals

[9]LU is more efficient. We simply wanted to illustrate other ways are possible.

Language	DR	CR	DoS
Gwydion	1.74	18.27	2.14
OpenDylan	2.51	43.84	1.23
CMUCL	2.03	6.34	1.17
SBCL	2.37	26.57	1.11
McCLIM	2.32	15.43	1.17
Vortex	2.33	63.30	1.06
Whirlwind	2.07	31.65	0.71
NiceC	1.36	3.46	0.33
LocStack	1.50	8.92	1.02
Julia	5.86	51.44	1.54
Julia operators	28.13	78.06	2.01

Table 1: A comparison of Julia (1208 functions exported from the `Base` library) to other languages with multiple dispatch. The "Julia operators" row describes 47 functions with special syntax (binary operators, indexing, and concatenation). Data for other systems are from Ref. [16]. The results indicate that Julia is using multiple dispatch far more heavily than previous systems.

the usual 3-term recurrence formula is used. (The first four are defined as one line functions; the symmetric tridiagonal uses the long form.)

In[38]:
```
# Simple determinants defined using the short form for functions
newdet(x::Number) = x
newdet(A::Diagonal ) = prod(diag(A))
newdet(A::Triangular) = prod(diag(A))
newdet(A::Matrix) = -prod(diag(qrfact(full(A))[:R]))*(-1)ŝize(A,1)

# Tridiagonal determinant defined using the long form for functions
function newdet(A::SymTridiagonal)
    # Assign c and d as a pair
    c,d = 1, A[1,1]
    for i=2:size(A,1)
        # temp=d, d=the expression, c=temp
        c,d = d, d*A[i,i]-c*A[i,i-1]^2
    end
    d
end
```

We have illustrated a mechanism to select a determinant formula at runtime based on the type of the input argument. If Julia knows an argument type early, it can make use of this information for performance. If it does not, code selection can still happen, at runtime. The reason why Julia can still perform well is that once code selection based on type occurs, Julia can return to performing well once insider the method.

4.4.2 A Symmetric Arrow Matrix Type

In the field of Matrix Computations, there are matrix structures and operations on these matrices. In Julia, these structures exist as Julia types. Julia has a number of predefined matrix structure types: (dense) `Matrix`, (compressed sparse column) `SparseMatrixCSC`, `Symmetric`, `Hermitian`, `SymTridiagonal`, `Bidiagonal`, `Tridiagonal`. `Diagonal`, and `Triangular` are all examples of Julia's matrix structures.

The operations on these matrices exist as Julia's methods. Familiar examples of operations are indexing, determinant, size, and matrix addition. Since matrix addition takes two arguments, it may be necessary to reconcile two different types when computing the sum.

Some languages do not allow you to extend their built in methods. This ability is known as external dispatch. In the following example, we illustrate how the user can add symmetric arrow matrices to the system, and then add a specialized `det` method to compute the determinant of a symmetric arrow matrix efficiently.

In[39]:
```
# Define a Symmetric Arrow Matrix Type
immutable SymArrow{T} <:  AbstractMatrix{T}
    dv::Vector{T} # diagonal
    ev::Vector{T} # 1st row[2:n]
  end
```

In[40]:
```
# Define its size
importall Base
size(A::SymArrow, dim::Integer) = size(A.dv,1)
size(A::SymArrow)= size(A,1), size(A,1)
```

Out[40]:
```
size (generic function with 52 methods)
```

In[41]:
```
# Index into a SymArrow
function getindex(A::SymArrow,i::Integer,j::Integer)
    if i==j; return A.dv[i]
      elseif i==1; return A.ev[j-1]
      elseif j==1; return A.ev[i-1]
      else return zero(typeof(A.dv[1]))
    end
end
```

Out[41]:
```
getindex (generic function with 168 methods)
```

In[42]:
```
# Dense version of SymArrow
full(A::SymArrow) =[A[i,j] for i=1:size(A,1), j=1:size(A,2)]
```

Out[42]:
```
full (generic function with 17 methods)
```

In[43]:
```
# An example
S=SymArrow([1,2,3,4,5],[6,7,8,9])
```

Out[43]:
```
5x5 SymArrow{Int64}:
1 6 7 8 9
6 2 0 0 0
7 0 3 0 0
8 0 0 4 0
9 0 0 0 5
```

```
In [44]: # det for SymArrow (external dispatch example)
         function exc_prod(v) # prod(v)/v[i]
             [prod(v[[1:(i-1),(i+1):end]]) for i=1:size(v,1)]
         end
         # det for SymArrow formula
         det(A::SymArrow) = prod(A.dv)-sum(A.ev.^2.*exc_prod(A.dv[2:end]))
```

```
Out[44]: det (generic function with 17 methods)
```

The above julia code uses the special formula

$$\det(A) = \prod_{i=1}^{n} d_i - \sum_{i=2}^{n} e_i^2 \prod_{2\le j\ne i\le n} d_j,$$

valid for symmetric arrow matrices with diagonal d and first row starting with the second entry e.

In Julia terminology `det` and `newdet` are *functions.* In some technical computing languages, a function might begin with a lot of argument checking to pick which algorithm to use. In Julia, one creates a number of *methods.* Thus `newdet` on a diagonal is one method for `newdet`, and `newdet` on a triangular matrix is a second method. `det` on a `SymArrow` is a new method for `det`. Code is selected, in advance if the compiler knows the type, otherwise the code is selected at run time. The selection of code is known as *dispatch.*

We have seen a number of examples of code selection for single dispatch. We can now turn to a very powerful feature, Julia's multiple dispatch mechanism. Now that we have created a symmetric arrow matrix, we might want to add it to all possible matrices of all types. However, we might notice that a symmetric arrow plus a diagonal does not require operations on full dense matrices.

The code below starts with the most general case, and then allows for specialization for the symmetric arrow and diagonal sum:

```
In [45]: # SymArrow + Any Matrix:  (Fallback:  add full dense arrays )
         +(A::SymArrow, B::Matrix) = full(A)+B
         +(B::Matrix, A::SymArrow) = A+B
         # SymArrow + Diagonal:  (Special case:  add diagonals, copy off-diagonal)
         +(A::SymArrow, B::Diagonal) = SymArrow(A.dv+B.diag,A.ev)
         +(B::Diagonal, A::SymArrow) = A+B
```

5 Code reuse: Code Generation is not only for the compiler

At the crossroads of code selection and code reuse is the linear algebra software engineer's dilemma. The complexity of the full solution has been nicely captured in the context of LAPACK and ScaLAPACK by Demmel and Dongarra, et.al., [4] and reproduced verbatim here:

```
(1) for all linear algebra problems
     (linear systems, eigenproblems, ...)
(2)     for all matrix types
            (general, symmetric, banded, ...)
(3)           for all data types
                  (real, complex, single, double, higher precision)
(4)                 for all machine architectures
                      and communication topologies
(5)                       for all programming interfaces
(6)                            provide the best algorithm(s) available in terms of
                                 performance and accuracy (``algorithms" is plural
                                 because sometimes no single one is always best)
```

Obviously (1) and (6) and perhaps (4) and even (5) are about code selection. (2) and (3) are about code reuse at the high level, and code selection at the lowest levels.

In the language of Computer Science, code reuse is about taking advantage of polymorphism. In the general language of mathematics it's about taking advantage of abstraction, or the sameness of two things. Either way, programs are efficient, powerful, and maintainable if programmers are given powerful mechanisms to reuse code.

Reusing code can be very tricky. Much of the development of modern programming language theory, both static and dynamic, can be understood by following the emergence of various forms of polymorphism. Saying the same notion another way, figuring out how to enable users to reuse code is perhaps far more difficult than training mathematicians to recognize that two different mathematical objects share sufficient commonality that they may be given the same name.

5.1 Example: Non-floating point linear algebra

Increasingly, the applicability of linear algebra has gone well beyond the LAPACK world of floating point numbers. These days linear algebra is being performed on, say, high precision numbers, integers, elements of finite fields, or rational numbers.

There will always be a special place for the BLAS, and the performance it provides for floating point numbers. Nonetheless, linear algebra operations like "inv" transcend any one data type. One can write a general "inv" and as long as the necessary operations are available, the code just works. That is the power of code reuse.

Here we show an example of rational types just working. A rational could be replaced by other constructs, which if they have well defined operations of algebra, will continue to work.

In [46]:
```
nHilbert = 8
```

Out[46]:
```
8
```

In [47]:
```
H = Rational{BigInt}[1//(i+j-1) for i=1:nHilbert, j=1:nHilbert]
```

```
Out[47]: 8x8 Array{Rational{BigInt},2}:
          1//1  1//2  1//3   1//4   1//5   1//6   1//7   1//8
          1//2  1//3  1//4   1//5   1//6   1//7   1//8   1//9
          1//3  1//4  1//5   1//6   1//7   1//8   1//9   1//10
          1//4  1//5  1//6   1//7   1//8   1//9   1//10  1//11
          1//5  1//6  1//7   1//8   1//9   1//10  1//11  1//12
          1//6  1//7  1//8   1//9   1//10  1//11  1//12  1//13
          1//7  1//8  1//9   1//10  1//11  1//12  1//13  1//14
          1//8  1//9  1//10  1//11  1//12  1//13  1//14  1//15
```

```
In[48]:  inv(H)
```

```
Out[48]: 8x8 Array{Rational{BigInt},2}:
              64//1      -2016//1  …  -288288//1       192192//1      -51480//1
           -2016//1      84672//1      15567552//1   -10594584//1     2882880//1
           20160//1    -952560//1    -204324120//1    141261120//1   -38918880//1
          -92400//1    4656960//1     1109908800//1   -776936160//1   216216000//1
          221760//1  -11642400//1    -2996753760//1   2118916800//1  -594594000//1
         -288288//1   15567552//1  …  4249941696//1  -3030051024//1   856215360//1
          192192//1  -10594584//1    -3030051024//1   2175421248//1  -618377760//1
          -51480//1    2882880//1      856215360//1  -618377760//1   176679360//1
```

5.2 Generic Programming for library development

Julia blurs the line between library developer and user. In the Julia philosophy anyone can contribute actively if they feel comfortable and ready.

The modern numerical computing world depends heavily on domain libraries for all areas of applied math, and specialized analyses for different branches of science. These libraries make end users more productive, but developing the libraries themselves is a difficult problem. A useful library needs to be both high performance and generic, providing enough flexibility to adapt to the varying problems of different users.

Meeting these requirements places a large burden on library developers. The need for code that is both generic and performant pushes programming languages to their limits, requiring use of advanced features such as C++ templates. Next, the popularity of dynamic languages means that interfacing work is needed to make these libraries callable from Python, R, and other systems. Overall, this workflow requires not only domain knowledge, but a high level of programming skill and knowledge of language internals. These requirements on time and expertise are unreasonable.

The design of Julia changes this situation. While being "high level" and "fast" are prerequisites to solving the problem, the true requirements involve subtleties that are not addressed just by taking any high level language and speeding it up. To obtain the best performance for complex library constructs, a compiler must understand code more deeply, and perform more computations at compile time. This is where a fresh language design is needed.

A good example is determining the ranks of multi-dimensional arrays. numerical computing systems typically have complex rules for how various functions operate on array dimensions, and knowing the ranks of arrays at compile time is crucial for performance. Therefore compilers for "array languages" like APL and MATLAB typically have built-in rules for such functions. However built-in rules do not address the flexibility needs of library developers. More general mechanisms like C++ templates are needed to "teach" compilers new rules. Unfortunately, templates are notoriously difficult to use.

We addressed the array-rank-inference problem in a recent paper [1]. It turns out that Julia's combination of multiple dispatch and dataflow type inference allows array rank rules to be defined in a natural way, with minimal code. Furthermore, this style of expression allows our compiler to infer array ranks without specialized array knowledge being built in. The following example of this technique is taken directly from Julia's standard library:

In[49]:
```
index_shape(I::Real...)   = ()
index_shape(i, I...)   = tuple(length(i), index_shape(I...)...)
```

The arguments are indices that might be used to index an array, and the output is the shape. These two lines define a generic function that computes the shape of the array I resulting from an indexing operation. It implements the rule that trailing dimensions indexed with scalars are dropped.

This code may look cryptic to the newcomer, so let's break it down.

In[50]:
```
# Indexing entirely by scalars returns an empty tuple
index_shape(I::Real...)   = ()
```

The `I::Real` indicates a real scalar number type. The `...` indicates a sequence of such types, which can be any length including 0. Thus calling `index_shape` with any sequence of scalars yields an empty array:

In[51]:
```
println( index_shape() )
println( index_shape(1,2,3) )
println( index_shape(1,2,3,4) )
```

Out[51]:
```
()
()
()
```

The method on the second line is dispatched if any one argument is not a scalar:

In[52]:
```
# Peel off the length of the first argument, and dispatch the remaining part
index_shape(i,I...)=tuple(length(i),index_shape(I...)...)
```

In[53]:
```
println( index_shape(7:9) )
println( index_shape(5,7:9) )
println( index_shape(7:9,1,2,3,4) )
```

Out[53]:
```
(3,)
(1,3)
(3,)
```

Different rules (e.g. dropping all dimensions indexed with scalars) can be obtained by adjusting the definitions slightly. When dataflow type inference is applied to these definitions for any particular invocation, a tuple type of the correct length emerges, telling the compiler the array rank.

We emphasize that this can be written in other ways in other technical computing languages, but doing it this way gives the information to the compiler. This is the moral of Julia, that the programmer can help provide information to the compiler so that code is not slowed down at run time.

5.3 Macros

Julia has a macro system that provides easy custom code generation, bringing a level of performance that is otherwise difficult to achieve. For example, a library developer implemented an `@evalpoly` macro that uses Horner's rule to evaluate polynomials efficiently. Consider

```
In[54]: @evalpoly(10,3,4,5,6)
```

which returns 6543 (the polynomial $3 + 4x + 5x + 6x^2$, evaluated at 10 with Horner's rule). Julia allows us to see the inline generated code with the command

```
In[55]: macroexpand(:@evalpoly(10,3,4,5,6))
```

We reproduce the key lines below

```
Out[55]: #471#t = 10 # Store 10 into a variable named #471#t
         Base.Math.+(3,Base.Math.*(#471#t,Base.Math.+(4,Base.Math.*
         (#471#t,Base.Math.+(5,Base.Math.*(#471#t,6)))) ))
```

This code-generating macro only needs to produce the correct symbolic structure, and Julia's compiler handles the remaining details of fast native code generation. Since polynomial evaluation is so important for numerical library software it is critical that users can evaluate polynomials without having to check if coefficients have reached an end of a stack or have to worry about whether the numbers are real or complex. We emphasize the key point that any technical language can evaluate a quartic, but in Julia the compiler generates specifically code to evaluate a quartic, and not a cubic or a quintic at runtime.

Recently it was reported[10] by Steve Johnson (a former Wilkinson prize winner for FFTW) who wanted to implement the digamma special function for complex arguments that:

> As described in Knuth TAOCP vol. 2, sec. 4.6.4, there is actually an algorithm even better than Horner's rule for evaluating polynomials p(z) at complex arguments (but with real coefficients): you can save almost a factor of two for high degrees. It is so complicated that it is basically only usable via code generation, so it would be especially nice to modify the @horner macro to switch to this for complex arguments.

No sooner than this was proposed, the macro was rewritten to allow for this case giving a factor of four performance improvement on all real polynomials evaluated at complex arguments.

5.4 Just-In-Time compilation and code specialization

Assuming that users want performance, Julia specializes code for run-time types by default. This technique is fairly effective, often yielding performance within a factor of 2 of C (Fig. 5).

Through specialization, our compiler can remove overhead for user defined types with sophisticated behaviors. The Units package written by Keno Fischer [11] is a good example. With this package installed, it is possible to ask for the sum of two lengths:

```
In[56]: Pkg.add("SIUnits") # Needed once to download the package
        using SIUnits
        1Meter + 2Meter
```

```
Out[56]: 3m
```

The `+` methods defined in the library get compiled to fast machine code, consisting only of an `add` instruction and stack operations, with the `Meter`s completely optimized away. It is possible to see the x86 assembler code any time with the `@code_native` command. (There are many online listings such as `http://en.wikipedia.org/wiki/X86_instruction_listings` for users to learn what all

[10] `https://github.com/JuliaLang/julia/issues/7033`
[11] `https://github.com/Keno/SIUnits.jl`

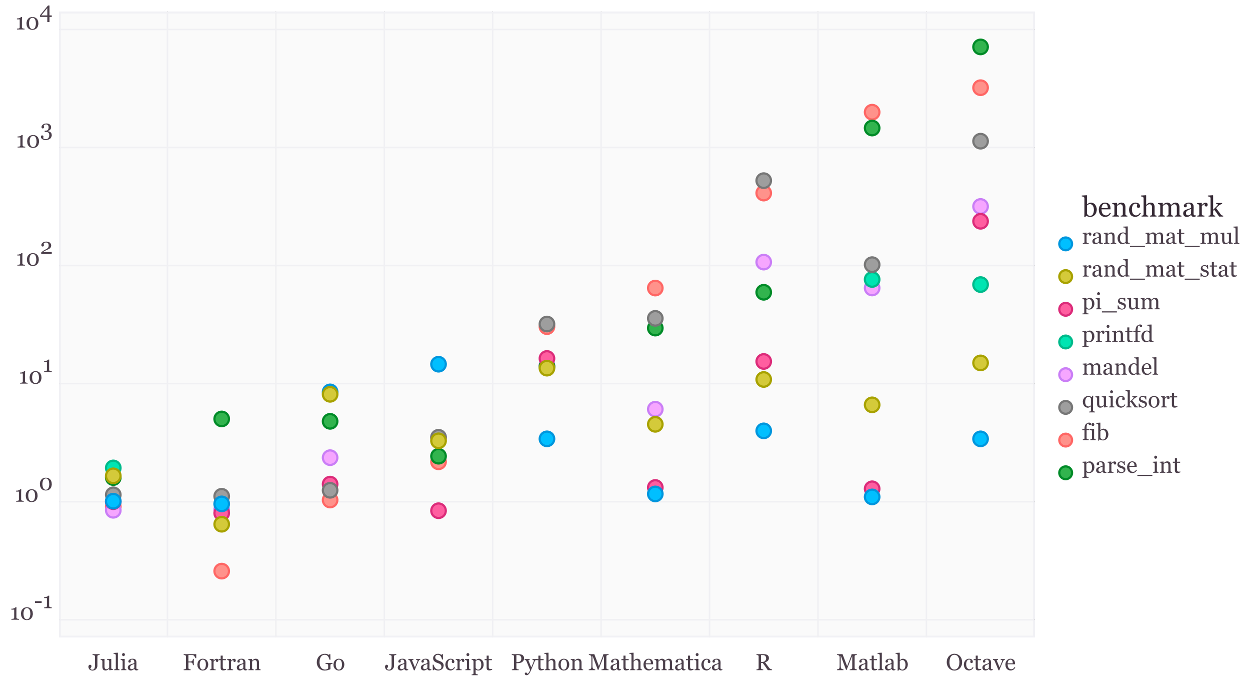


Figure 5: Performance comparison of various language performing simple micro-benchmarks. Benchmark execution time relative to C. (Smaller is better, C performance = 1.0).

the commands are doing. Julia users who never looked at assembler before, can get to understand assembler very quickly by simply trying out a few simple commands.)

In [57]:
```
@code_native 1Meter + 2Meter
```

Out[57]:
```
  .section __TEXT,__text,regular,pure_instructions
Filename:  /Users/viral/.julia/SIUnits/src/SIUnits.jl
Source line:  122
  push RBP
  mov RBP, RSP
Source line:  122
  add RDI, RSI
Source line:  123
  mov RAX, RDI
  pop RBP
  ret
```

The trained eye sees this as minimal code generated for the instruction. Specifically there is some register activity and simply one add instruction (highlighted here).

6 Language and standard library design

Seemingly innocuous design choices in a language can have profound, pervasive performance implications. These are often overlooked in languages that were not designed from the beginning to be able to deliver excellent performance. Other aspects of language and library design affect the usability, composability, and power of the provided functionality.

6.1 Integer arithmetic

A simple but crucial example of a performance-critical language design choice is integer arithmetic. Julia uses machine arithmetic for integer computations. This means that the range of Int values is bounded and wraps around at either end so that adding, subtracting and multiplying integers can overflow or underflow, leading to results that are familiar to C and Fortran programmers but which can be unsettling to users of systems like Mathematica or Python:

```
In[58]:  typemax(Int)

Out[58]:  9223372036854775807

In[59]:  ans+1

Out[59]: -9223372036854775808

In[60]:  -ans

Out[60]: -9223372036854775808

In[61]:  2*ans

Out[61]: 0
```

In both Mathematica and Python, integers behave like mathematical integers, growing forever larger or smaller, never overflowing. While this may be an acceptable choice if only end-users will ever write code with the system's integers, it becomes rapidly clear that it is too slow if every loop in every function uses these integers to count its loops and compute indices and offsets.

Since machine multiplication and addition are associative and distribute, Julia is free to aggressively optimize simple little functions like `f(k) = 5k-1`. The machine code for this function is just this:

```
In[62]:  code_native(f,(Int,))

Out[62]:   .section __TEXT,__text,regular,pure_instructions
         Filename:  none
         Source line:  1
           push RBP
           mov RBP, RSP
         Source line:  1
           lea RAX, QWORD PTR [RDI + 4*RDI - 1]
           pop RBP
           ret
```

The actual body of the function is a single `lea` instruction, which computes the integer multiply and add at once. (Assembler names such as "'Load Effective Address," have grown irrelevant and one may think of LEA as simply an integer operation.) The benefit of minimal code generation is even more beneficial when `f` gets inlined into the inner loop of another function:

In[63]:
```
function g(k,n)
    for i = 1:n
      k = f(k)
    end
    return k
end
```

Out[63]:
```
g (generic function with 2 methods)
```

In[64]:
```
code_native(g,(Int,Int))
```

Out[64]:
```
  .section __TEXT,__text,regular,pure_instructions
Filename:  none
Source line:  3
  push RBP
  mov RBP, RSP
  test RSI, RSI
  jle 22
  mov EAX, 1
Source line:  3
  lea RDI, QWORD PTR [RDI + 4*RDI - 1]
  inc RAX
  cmp RAX, RSI
Source line:  2
  jle -17
Source line:  5
  mov RAX, RDI
  pop RBP
ret
```

Since the call to `f` gets inlined, the loop body ends up being just a single `lea` instruction. Next, consider what happens if we make the number of loop iterations fixed:

In[65]:
```
# 10 Iterations of f(k)=5k-1 on integers
function g(k)
    for i = 1:10
      k = f(k)
    end
    return k
end
```

Out[65]:
```
g (generic function with 2 methods)
```

In[66]:
```
code_native(g,(Int,))
```

Out[66]:
```
  .section __TEXT,__text,regular,pure_instructions
Filename:  none
Source line:  3
  push RBP
  mov RBP, RSP
Source line:  3
  imul RAX, RDI, 9765625
  add RAX, -2441406
Source line:  5
  pop RBP
  ret
```

Because the compiler knows that integer addition and multiplication are associative and that multiplication distributes over addition it can optimize the entire loop down to just a multiply and an add. Indeed, if $f(k) = 5k - 1$, it is true that the tenfold iterate $f^{(10)}(k) = -2441406 + 9765625k$.

6.2 A powerful approach to linear algebra

6.2.1 Matrix factorizations

For decades, orthogonal matrices have been represented internally as products of Household matrices stored in terms of vectors, and displayed for humans as matrix elements. LU factorizations are often performed in place, storing the L and U information together in the data locations originally occupied by A. All this speaks to the fact that matrix factorizations deserve to be first class objects in a linear algebra system.

In Julia, thanks to the contributions of Andreas Noack Jensen and many others, these structures are indeed first class objects. The structure `QRCompactWY` holds a compact Q and an R in memory. Similarly an `LU` holds an L and U in packed form in memory. Through the magic of multiple dispatch, we can solve linear systems, extract the pieces, and do least squares directly on these structures.

The QR example is even more fascinating. Suppose one computes QR of a 5×3 matrix. What is the size of Q? The right answer, of course, is that it depends: it could be 5×5 or 5×3. The underlying representation is the same.

In Julia one can compute `Aqr = qrfact(rand(5,3))`. Then one can take `Q=Aqr[:Q]`. This Q retains its clever underlying structure and therefore is efficient and applicable when multiplying vectors of length 5 or length 3, contrary to the rules of freshman linear algebra, but welcome in numerical libraries for saving space and faster computations.

julia¿ Aqr = qrfact(rand(5,3));

In[67]:
```
Q = Aqr[:Q]
```

Out[67]:
```
5x5 QRCompactWYQ{Float64}:
 -0.251536   0.154937   0.77616
 -0.101711  -0.608041   0.512436
 -0.755597   0.217326  -0.0987686
 -0.014734  -0.714761  -0.210326
 -0.596021  -0.219468  -0.284593
```

In[68]:
```
Q*rand(5)
```

```
Out[68]:  5-element Array{Float64,1}:
            0.755784
           -1.10069
           -0.757399
           -0.272477
           -0.0510777
```

```
In[69]:  Q*rand(3)
```

```
Out[69]:  5-element Array{Float64,1}:
            0.152596
           -0.0562855
           -0.569295
           -0.303167
           -0.644554
```

6.2.2 User-extensible wrappers for BLAS and LAPACK

The tradition in linear algebra is to leave the coding to LAPACK writers, and call LAPACK for speed and accuracy. This has worked fairly well, but Julia exposes considerable opportunities for improvement.

Firstly, all of LAPACK is available to Julia users, not just the most common functions. All LAPACK wrappers are implemented fully in Julia code, using "ccall"[12], which does not require a C compiler, and can be called directly from the interactive Julia prompt. This makes it easy for users to contribute LAPACK functionality, and that is how Julia's LAPACK functionality has grown bit by bit. Wrappers for missing LAPACK functionality can also be added by users in their own code.

Consider the following example that implements the Cholesky factorization by calling LAPACK's `xPOTRF`. It uses Julia's metaprogramming facilities to generate four functions, each corresponding to the `xPOTRF` functions for `Float32`, `Float64`, `Complex64`, and `Complex128` types. The actual call to the Fortran functions is wrapped in `ccall`. Finally, the `chol` function provides a user-accessible way

[12]`http://docs.julialang.org/en/latest/manual/calling-c-and-fortran-code/`

to compute the factorization. It is easy to modify the template below for any LAPACK call.

In[70]:
```
# Generate calls to LAPACK's Cholesky for double, single, etc.
# xPOTRF refers to POsitive definite TRiangular Factor
# LAPACK signature:  SUBROUTINE DPOTRF( UPLO, N, A, LDA, INFO )
# LAPACK documentation:
*  UPLO    (input) CHARACTER*1
*          = 'U':  Upper triangle of A is stored;
*          = 'L':  Lower triangle of A is stored.
*  N       (input) INTEGER
*          The order of the matrix A.  N >= 0.
*  A       (input/output) DOUBLE PRECISION array, dimension (LDA,N)
*          On entry, the symmetric matrix A.  If UPLO = 'U', the leading
*          N-by-N upper triangular part of A contains the upper
*          triangular part of the matrix A, and the strictly lower
*          triangular part of A is not referenced.  If UPLO = 'L', the
*          leading N-by-N lower triangular part of A contains the lower
*          triangular part of the matrix A, and the strictly upper
*          triangular part of A is not referenced.
*          On exit, if INFO = 0, the factor U or L from the Cholesky
*          factorization A = U**T*U or A = L*L**T.
*  LDA     (input) INTEGER
*          The leading dimension of the array A.  LDA >= max(1,N).
*  INFO    (output) INTEGER
*          = 0:  successful exit
*          < 0:  if INFO = -i, the i-th argument had an illegal value
*          > 0:  if INFO = i, the leading minor of order i is not
*                positive definite, and the factorization could not be
*                completed.

# Generate Julia method potrf!
   for  (potrf,  elty) in  # Run through 4 element types
     ((:dpotrf_,:Float64),
      (:spotrf_,:Float32),
      (:zpotrf_,:Complex128),
      (:cpotrf_,:Complex64))
# Begin function potrf!
     @eval begin
         function potrf!(uplo::Char, A::StridedMatrix{$elty})
             lda = max(1,stride(A,2))
             lda==0 && return A, 0
             info = Array(Int, 1)
# Call to LAPACK:ccall(LAPACKroutine,Void,PointerTypes,JuliaVariables)
             ccall(($(string(potrf)),:liblapack), Void,
                    (Ptr{Char}, Ptr{Int}, Ptr{$elty}, Ptr{Int}, Ptr{Int}),
                       &uplo,    &size(A,1),    A,         &lda,    info)
             return A, info[1]
          end
     end
end

chol(A::Matrix) = potrf!('U', copy(A))
```

6.3 Easy and flexible parallelism

Once single processor performance has reached reasonable limits it becomes appropriate to consider parallelism. Parallel computing has not reached the levels of interactivity required for innovation. Julia provides many facilities for parallelism, which are described in detail in the manual[13].

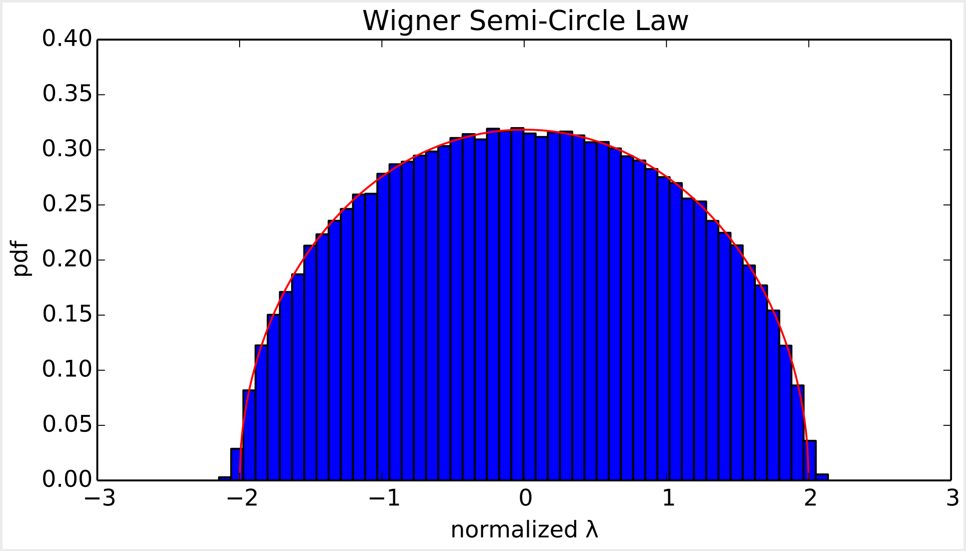
We begin on a single processor with a user performing a simple Monte-Carlo experiment: the famous semicircle law from Random Matrix Theory for a random eigenvalue of a symmetric matrix:

In[71]:
```
n=100 # Matrix Size
t=1000 # Number of Trials
sym(A)=A+A' # Symmetrize Matrix

# Eigenvalues of t nxn matrices saved as vector
z= [[eigvals(sym(randn(n,n))) for i=1:t]...]
z/=sqrt(2n)

# Histogram Plot
# using PyPlot
plt.figure(figsize=(8,3.5))
x=-2:.01:2;
plot(x,sqrt(4.-x.^2)/(2π),"r")
plt.hist(z,bins=50,normed=true)

# Customize and save to file
axis([-3,3,0,.4])
xlabel("normalized λ")
ylabel("pdf")
title("Wigner Semi-Circle Law")
plt.savefig("myfig.pdf")
```



[13] http://docs.julialang.org/en/latest/manual/parallel-computing/

Now suppose we wish to perform a more complicated histogram in parallel. Here is another example from Random Matrix Theory, the computation of the scaled largest eigenvalue in magnitude of the so called stochastic operator Airy operator

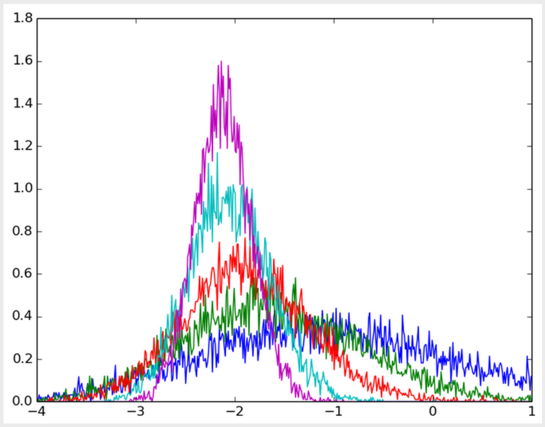
$$\frac{d^2}{dx^2} - x + \frac{1}{2\sqrt{\beta}} dW.$$

This is just the usual finite difference discretization of $\frac{d^2}{dx^2} - x$ with a "noisy" diagonal.

We illustrate an example of the famous Tracy-Widom law being simulated with Monte Carlo experiments for different values of the inverse temperature parameter β. The code on 1 processor is fuzzy and unfocused, as compared to the same simulation on 75 processors, which is sharp and focused.

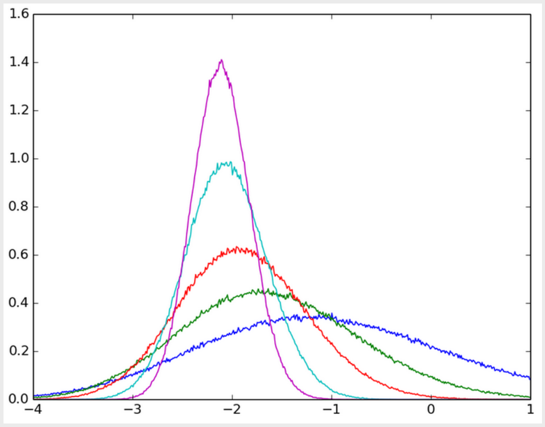
In[72]:
```
 @everywhere begin                        # define on every processor
 function stochastic(β=2,n=200)
   h=n^-(1/3)
   x=0:h:10
   N=length(x)
   d=(-2/h^2 .-x) +  2 sqrt(h*β)*randn(N) # diagonal
   e=ones(N-1)/h^2                        # subdiagonal
   eigvals(SymTridiagonal(d,e))[N]     # smallest negative eigenvalue
end
end
```

In[73]:
```
t = 10000
for β=[1,2,4,10,20]
  hist([stochastic(β) for i=1:t], -4:.01:1)[2]
  plot(midpoints(-4:.01:1),z/sum(z)/.01)
end
```



In[74]:
```
# Readily adding 75 processors sharpens the Monte Carlo simulation
addprocs(75)
```

In[75]:
```
t = 10000
for β=[1,2,4,10,20]
z = @parallel (+) for i=1:nprocs()
     hist([stochastic(β) for i=1:t], -4:.01:1)[2]
end
plot(midpoints(-4:.01:1),z/sum(z)/.01)
end
```



6.4 Performance Recap

In the early days of high level numerical computing languages, the thinking was that the performance of the high level language did not matter so much so long as most of the time was spent inside the numerical libraries. These libraries consisted of blockbuster algorithms that would be highly tuned, making efficient use of computer memory, cache, and low level instructions.

What the world learned was that only a few codes were spending a majority of their time in the blockbusters. Real codes were being caught by interpreter overheads, stemming from processing more aspects of a program at run time than are strictly necessary.

As we explored in Section 3, one of the hindrances of completing this analysis is type information. Programming language design thus becomes an exercise in balancing incentives to the programmer to provide type information and the ability of the computer to infer type information. Vectorization is one such incentive system. Existing technical computing languages would have us believe that this is the only system, or even if there were others, that somehow this was the best system.

Vectorization at the software level can be elegant for some problems. There are many matrix computation problems that look beautiful vectorized. These programs should be vectorized. Other programs require heroics and skill to vectorize sometimes producing unreadable code all in the name of performance. These are the ones that we object to vectorizing. Still other programs can not be vectorized very well even with heroics. The Julia message is to vectorize when it is natural, producing nice code. Do not vectorize in the name of speed.

Some users believe that vectorizing software is required to make use of special hardware capabilities including the ability to use SIMD instructions, multithreading, GPU units, and other forms of parallelism. The Julia message remains: vectorize when natural, when you feel it is right.

7 Conclusion and Acknowledgments

Julia was created to meet the needs of scientific computing. It has become so popular in such a short time that googling "Julia" puts the language on top of the list (even above Julia Roberts.) At the time of writing, not a day goes by where we don't learn that someone else has picked up Julia at Stanford, or at Berkeley, or at companies like Facebook or in finance and biotech. More than just a language, Julia has become a place for programmers, physical scientists, social scientists, computational scientists, mathematicians, and others to pool their collective knowledge in the form of online discussions and in the form of code. Technical computing is maturing and it is exciting to watch!

The authors gratefully acknowledge support from the MIT Deshpande center for technical innovation, the Intel Technology Science Center for Big Data, the DARPA Xdata program, the Singapore MIT Alliance, NSF Awards CCF-0832997 and DMS-1016125, VMWare Research, a DOE grant with Dr. Andrew Gelman of Columbia University for petascale hierarchical modeling, and a Citibank grant for High Performance Banking Data Analysis.

References

[1] Jeff Bezanson, Jiahao Chen, Stefan Karpinski, Viral B. Shah, and Alan Edelman. Array operators using multiple dispatch. *ARRAY'14*, 2014.

[2] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. arXiv:1209.5145v1, 2012.

[3] clang. `http://clang.llvm.org/`.

[4] James W. Demmel, Jack J. Dongarra, Beresford N. Parlett, William Kahan, Ming Gu, David S. Bindel, Yozo Hida, Xiaoye S. Li, Osni A. Marques, E. Jason Riedy, Christof V?mel, Julien Langou, Piotr Luszczek, Jakub Kurzak, Alfredo Buttari, Julie Langou, and Stanimire Tomov. Prospectus for the next LAPACK and ScaLAPACK libraries. Technical Report 181, LAPACK Working Note, February 2007.

[5] Claude Gomez, editor. *Engineering and Scientific Computing With Scilab*. Birkhäuser, 1999.

[6] Graydon Hoare. technicalities: interactive scientific computing #1 of 2, pythonic parts. `http://graydon2.dreamwidth.org/3186.html`, 2014.

[7] R. Ihaka and R. Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5:299–314, 1996.

[8] Marc A. Kaplan and Jeffrey D. Ullman. A scheme for the automatic inference of variable types. *Journal of the ACM*, 27(1):128–145, January 1980.

[9] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, pages 75–86, Palo Alto, California, Mar 2004.

[10] MIT License. `http://opensource.org/licenses/MIT`.

[11] M. Lubin and I. Dunning. Computing in operations research using julia. `arXiv:1312.1431`.

[12] mathematica. `http://www.mathematica.com`.

[13] matlab. `http://www.mathworks.com`.

[14] Markus Mohnen. A graphfree approach to dataflow analysis. In R. Horspool, editor, *Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 185–213. Springer Berlin / Heidelberg, 2002.

[15] Malcolm Murphy. Octave: A free, high-level language for mathematics. *Linux J.*, 1997, July 1997.

[16] Radu Muschevici, Alex Potanin, Ewan Tempero, and James Noble. Multiple dispatch in practice. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 563–582, New York, NY, USA, 2008. ACM.

[17] rust. `http://www.rust-lang.org/`.

[18] swift. `https://developer.apple.com/swift/`.

[19] Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. The numpy array: a structure for efficient numerical computation. *CoRR*, abs/1102.1523, 2011.