arXiv:1411.1607v4 [cs.MS] 19 Jul 2015

Julia: A fresh approach to numerical computing

Jeff Bezanson Alan Edelman Stefan Karpinski Viral B. Shah

MIT and Julia Computing*

July 7, 2015

Abstract

Bridging cultures that have often been distant, Julia combines expertise from the diverse fields of computer science and computational science to create a new approach to numerical computing. Julia is designed to be easy and fast. Julia questions notions generally held as "laws of nature" by practitioners of numerical computing:

1. High-level dynamic programs have to be slow,
2. One must prototype in one language and then rewrite in another language for speed or deployment, and
3. There are parts of a system for the programmer, and other parts best left untouched as they are built by the experts.

We introduce the Julia programming language and its design — a dance between specialization and abstraction. Specialization allows for custom treatment. *Multiple dispatch*, a technique from computer science, picks the right algorithm for the right circumstance. Abstraction, what good computation is really about, recognizes what remains the same after differences are stripped away. Abstractions in mathematics are captured as code through another technique from computer science, *generic programming*.

Julia shows that one can have machine performance without sacrificing human convenience.

*http://www.juliacomputing.com

Contents

1 Scientific computing languages: The Julia innovation **3**
1.1 Computing transcends communities . . . 3
1.2 Julia architecture and language design philosophy . . . 3

2 A taste of Julia **5**
2.1 A brief tour . . . 5
2.2 An invaluable tool for numerical integrity . . . 9
2.3 The Julia community . . . 10

3 Writing programs with and without types **11**
3.1 The balance between human and the computer . . . 11
3.2 Julia's recognizable types . . . 11
3.3 User's own types are first class too . . . 12
3.4 Vectorization: Key Strengths and Serious Weaknesses . . . 13
3.5 Type inference rescues "for loops" and so much more . . . 14

4 Code selection: Run the right code at the right time **15**
4.1 Multiple Dispatch . . . 15
4.2 Code selection from bits to matrices . . . 17
4.2.1 Summing Numbers: Floats and Ints . . . 17
4.2.2 Summing Matrices: Dense and Sparse . . . 18
4.3 The many levels of code selection . . . 20
4.4 Is "code selection" just traditional object oriented programming? . . . 21
4.5 Quantifying the use of multiple dispatch . . . 22
4.6 Case Study for Numerical Computing . . . 23
4.6.1 Determinant: Simple Single Dispatch . . . 24
4.6.2 A Symmetric Arrow Matrix Type . . . 24

5 Leveraging language design for high performance libraries **26**
5.1 Integer arithmetic . . . 26
5.2 A powerful approach to linear algebra . . . 28
5.2.1 Matrix factorizations . . . 28
5.2.2 User-extensible wrappers for BLAS and LAPACK . . . 29
5.3 High Performance Polynomials and Special Functions with Macros . . . 31
5.4 Easy and flexible parallelism . . . 31
5.5 Performance Recap . . . 34

6 Conclusion and Acknowledgments **35**

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 Python [43], R [19], Mathematica [27], Octave [30], Matlab [28], and SciLab [16], to name some, 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.

Many researchers today do their day-to-day work in dynamic languages. Still, C and Fortran remain the gold standard for computationally-intensive problems for performance. In as much as the dynamic language programmer has missed out on performance, the C and Fortran programmer has missed out on productivity. 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. The consequences have been more serious than many realize.

Julia's innovation is the very combination of productivity and performance. New users 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. Julia is fast because of careful language design and the right combination of the carefully chosen technologies that work very well with each other. This paper demonstrates some of these technologies using a number of examples. We invite the reader to follow along at `http://juliabox.org` using Jupyter notebooks [32, 36] or by downloading Julia `http://julialang.org/downloads`.

1.1 Computing transcends communities

Numerical computing research has always lived on the boundary of computer science, engineering, mathematics, and computational sciences. Readers might enjoy trying to label the "Top 10 algorithms"[11] by field, and may quickly see that advances typically transcend any one field with broader impacts to science and technology as a whole.

Computing is more than using an overgrown calculator. It is a cross cutting communication medium. Research into programming languages therefore breaks us out of our research boundaries.

The first decade of the 21st century saw a boost in such research with the High Productivity Computing Systems DARPA funded projects into the languages such as Chapel [6, 7], Fortress [37, 1] and X10 [34, 8]. Also contemporaneous has been a growing acceptance of Python. Up to around 2009 some of us were working on Star-P, an interactive high performance computing system for parallelizing various dynamic programming languages. Some excellent resources on Star-P that discuss these ideas are the Star-P user Guide [39], the Star-P Getting Started guide [40], and various papers [9, 14, 18, 10]. Julia continues our research into parallel computing, with the most important lesson from our Star-P experience being that one cannot design a high performance parallel programming system without a programming language that works well sequentially.

1.2 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. For example, you can run Python code much faster than the standard CPython implementation using the PyPy just-in-time compiler; but PyPy is currently incompatible with NumPy and the rest of SciPy.

Another important point is that just because a program is available in C or Fortran, it may not run efficiently from the high level language or be easy to "glue" it in. For example when Steven Johnson tried to include his C `erf` function in Python, he reported that Pauli Virtane had to write glue code[1] to vectorize the erf function over the native structures in Python in order to get good performance. Johnson also had to write similar glue code for Matlab, Octave, and Scilab. The Julia effort was, by contrast, effortless.[2] As another example, `randn`, Julia's normal random number generator was originally based on calling `randmtzig`, a C implementation. It turned out later, that a pure Julia implementation of the same code actually ran faster, and is now the default implementation. In some cases, "glue" can often lead to poor performance, even when the underlying libraries being called are high performance.

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 [26] and the Convex.jl [42] packages are great examples of the success of this approach—the entire library is written in Julia and uses many Julia language features described in this paper.

The Two Language Problem: As long as the developers' language is harder than the users' language, numerical computing will always be hindered. 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. Julia solves the two language problem. 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 Python+NumPy, R, and Matlab hide their for loops and integer operations, but they are still there, inside the C and Fortran, lurking behind the 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 type system, allowing optional type annotations (Section 3);
2. Multiple dispatch using these types to select implementations (Section 4);
3. Metaprogramming for code generation (Section 5.3);
4. A dataflow type inference algorithm allowing types of most expressions to be inferred [3, 5];
5. Aggressive code specialization against run-time types [3, 5];
6. Just-In-Time (JIT) compilation [3, 5] using the LLVM compiler framework [23], which is also used by a number of other compilers such as Clang [12] and Apple's Swift [41]; and
7. Julia's carefully written libraries that leverage the language design, i.e., points 1 through 6 above (Section 5).

[1] `https://github.com/scipy/scipy/commit/ed14bf0`

[2] Steven Johnson, personal communication. See `http://ab-initio.mit.edu/wiki/index.php/Faddeeva_Package`

Points 1, 2, and 3 above are especially for the human, and the focus of this paper. On details about the parts that are about language implementation and internals such as in points 4, 5, and 6, we direct the reader to our earlier work([3, 5]). Point 7 brings everything together to build high performance computational libraries in Julia.

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 through the program due to dataflow type inference.

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.

2 A taste of Julia

2.1 A brief tour

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, without explicitly forming the identity matrix as is commonly done using commands such as "eye." ("eye" , a homonym of "I", is used in such languages as Matlab, Octave, Go's matrix library, Python's Numpy, and Scilab.)

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.

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
```

A commonly used notation to express the solution x to the equation $Ax = b$ is `A\b`. If Julia

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.)

knows that A is a tridiagonal matrix, it uses an efficient $O(n)$ algorithm:

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
```

Note the `Array{ElementType,dims}` syntax. In the above example, the elements are 64 bit floats or `Float64`'s. The `1` indicates it is a one dimensional vector.

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 then angle. By default, complex numbers are incomparable in Julia.

If a numerical computing language "hard-wires" its sort to be one or the other, it misses an 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. Note the compact function definition syntax

available in Julia used in the example below and is of the form `f(x,y,...) = <expression>`.

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)
```

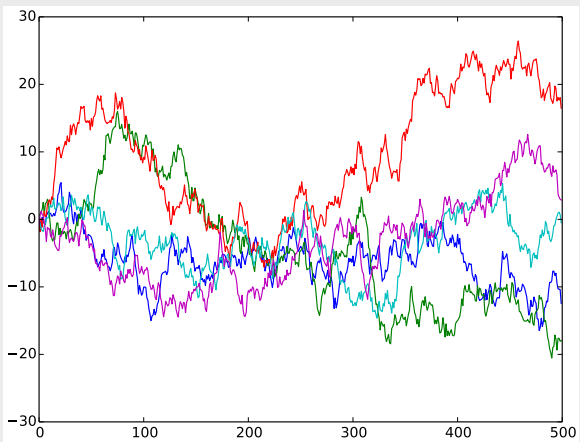
Out[6]:
```
5-element Array{Complex{Int64},1}:
 1+0im
 0+1im
-1+0im
 2+0im
-2+0im
```

To be sure, experienced computer scientists tend to suspect there is nothing new under the sun. The C function `qsort()` takes a `compar` function. Nothing really new there. Python also has custom sorting with a key. Matlab's sort is more basic. The real contribution of Julia, as will be fleshed out further in this paper, is that the design of Julia allows custom sorting to be high performance and flexible and comparable with implementations in other dynamic languages that are often written in

C.

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
```

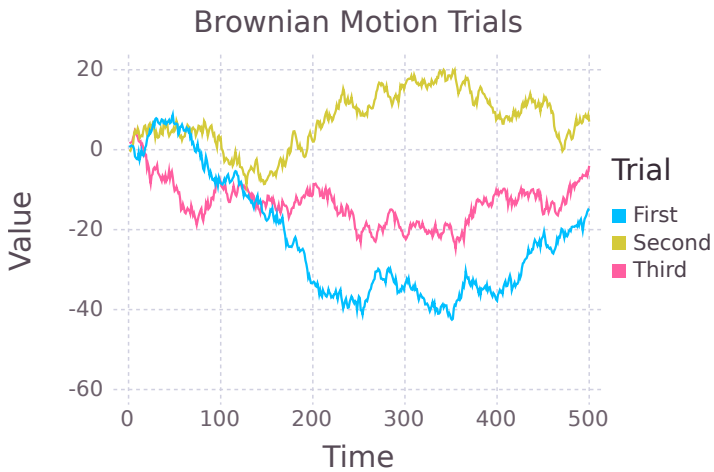


The next example that we have chosen for the introductory taste of Julia is a quick plot of Brownian motion, in two ways. The first such example uses the Python Matplotlib package for graphics, which is popular for users coming from Python or Matlab. The second example uses Gadfly.jl, another very popular package for plotting. Gadfly was built by Daniel Jones completely in Julia and was influenced by the well admired Grammar of Graphics (see [45] and [44])[3] Many Julia users find Gadfly more flexible and prefer its aesthetics. Julia plots can also be manipulated interactively with sliders and buttons using Julia's Interact.jl package[4]. The Interact.jl package page contains many examples of interactive visualizations[5].

In[8]:
```
Pkg.add("Gadfly") # Download the Gadfly package
using Gadfly # load the functionality into Julia


n = 500
p = [layer(x=1:n, y=cumsum(randn(n)), color=[i], Geom.line)
    for i in ["First","Second","Third"]]
labels=(Guide.xlabel("Time"),Guide.ylabel("Value"),
        Guide.Title("Brownian Motion Trials"),Guide.colorkey("Trial"))
plot(p...,labels...)
```

[3]See tutorial on `http://gadflyjl.org`
[4]https://github.com/JuliaLang/Interact.jl
[5]`https://github.com/JuliaLang/Interact.jl/issues/36`



The ellipses on the last line above is known as a `splat` operator. The elements of the vector `p` and the tuple `labels` are inserted individually as arguments to the `plot` function.

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[6] 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 numerical computing systems. If a difference is detected, one can also run the computation in higher precision. Kahan writes:

> Can the effects of roundoff upon a floating-point computation be assessed without submitting it to a mathematically rigorous and (if feasible at all) time-consuming error-analysis? In general, No.
>
> ...
>
> Though far from foolproof, rounding every inexact arithmetic operation (but not constants) in the same direction for each of two or three directions besides the default To Nearest is very likely to confirm accidentally exposed hypersensitivity to roundoff. When feasible, this scheme offers the best *Benefit/Cost* ratio. [21]

As an example, 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[9]:
```
h(n)=[1/(i+j+1) for i=1:n,j=1:n]
```

Out[9]:
```
h (generic function with 1 method)
```

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

In[10]:
```
H=h(15);
with_rounding(Float64,RoundNearest) do
    inv(H)[1,1]
end
```

Out[10]:
```
154410.55589294434
```

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

Out[11]:
```
-49499.606132507324
```

In[12]:
```
with_rounding(Float64,RoundDown) do
    inv(H)[1,1]
end
```

Out[12]:
```
-841819.4371948242
```

With 300 bits of precision, we obtain

In[13]:
```
with_bigfloat_precision(300) do
    inv(big(H))[1,1]
end
```

Out[13]:
```
-2.093971792507462701282801742144895161627088577037149597632326890471535076588249105499837625 2e+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 `Float64` results are senstive to the BLAS[24] and LAPACK[2], and may differ on different machines with different versions of Julia. For extended precision, Julia uses the MPFR library[33].

2.3 The Julia community

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 350 contributors and is available under the MIT License [25] for open source software. Over 1.3 million unique visitors have visited the Julia website since then, and Julia has now been adopted as a teaching tool in dozens of universities around the world[7]. While it was nurtured at the Massachusetts Institute of Technology, it is really the contributions from experts around the world that make it a joy to use for numerical computing. It is also recognized as a general purpose computing language unlike traditional numerical computing systems, allowing it to be used not only to prototype numerical algorithms, but also to deploy those algorithms, and even serve results to the rest of the world. A great example of this is Shashi Gowda's Escher.jl package[8], which makes it possible for Julia programmers to build beautiful interactive websites in Julia, and serve up the results of a Julia computation from the web server, without any knowledge of HTML or javascript. Another such example is the Sudoku as a service[9], by Iain Dunning, where a Sudoku puzzle is solved using the optimization capabilities of the JUMP.jl Julia package [26] and made available as a web service. This is exactly why Julia is being increasingly deployed in production environments in businesses, as seen in various talks at JuliaCon [10]. These use cases utilize not

[7] `http://julialang.org/community`
[8] https://github.com/shashi/Escher.jl
[9] http://iaindunning.com/2013/sudoku-as-a-service.html
[10] `http://www.juliacon.org`

just Julia's capabilities for mathematical computation, but also to build web APIs, database access, and much more. Perhaps most significantly, a rapidly growing ecosystem of over 600 open source, composable packages [11], which include a mix of libraries for mathematical as well as general purpose computing, is leading to the adoption of Julia in research, businesses, and in government.

3 Writing programs with and without types

3.1 The balance between human and the computer

Graydon Hoare, author of the Rust programming language [35], in an essay on "Interactive Scientific Computing" [17] 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.

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

Many users of Julia may never need to know about types for performance. Julia's type inference system often does the work, giving performance without type declarations.

Julia's design allows for the 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[14]:
```
rand(1,2,1)
```

Out[14]:
```
1x2x1 Array{Float64,3}:
[ :, :, 1] =
  0.789166 0.652002
```

In[15]:
```
[1 2; 3 4]
```

Out[15]:
```
2x2 Array{Int64,2}:
  1 2
  3 4
```

[11] urlhttp://pkg.julialang.org

In[16]:
```
[true; false]
```

Out[16]:
```
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[17]:
```
a = [eye(2) for i=1:2]
```

Out[17]:
```
2-element Array{Array{Float64,2},1}:
```

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[18]:
```
# Type Parameter 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[18]:
```
SymArrow{Int64}([1,2,3,4,5],[6,7,8,9])
```

The parameter in the array refers to the type of each element of the array. Code can and should be written independently of the type of each element.

Later in Section 4.6.2, we develop the symmetric arrow 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 4.6.2 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 have parameters 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[12].

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 sum computation. Note that due to the size of the problem, the computation is memory bound, and one does not observe the case with complex arithmetic to be twice as slower than the real case, even though it is performing twice as many floating point operations.

In[19]:
```
# 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
    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[20]:
```
x = ones(1_000_000)
@time prefix(x)

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

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

```
Out[20]:  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 provided as a `built-in` function in 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, which are essential when writing libraries that utilize multiple dispatch, but not for end-user programs that are exploring algorithms or a dataset.

Generally, in Julia, type annotations are not for performance. They are purely for code selection. (See Section 4). 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 dataflow type inference [29],[22],[5]. 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.

The 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.

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 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".

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: Run the right code at the right time

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.[13]

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 Multiple Dispatch

Multiple dispatch is the selection of a function implementation based on the types of each argument of the function. It is not only a nice notation to remove a long list of "case" statements, but it is part of the reason for Julia's speed. It is expressed in Julia by annotating the type of a function argument in a function definition with the following syntax: `argument::Type`.

[13] A few versions of this quote are relevant to Julia's power of abstractions and numerical 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`

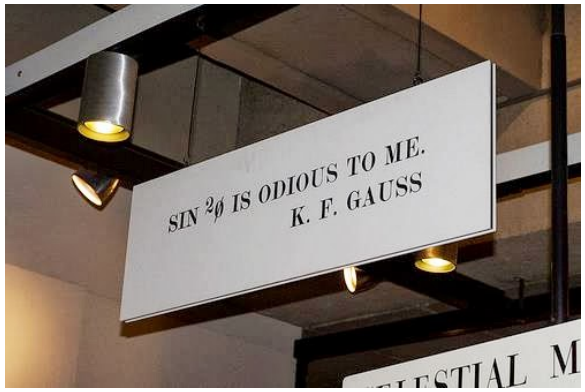


Figure 2: Gauss quote hanging from the ceiling of the longstanding old Boston Museum of Science Mathematica Exhibit.

Mathematical notations that are often used in print 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[21]:
```
*(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 2).) 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 many other uses abound. The fact that the compiler can pick the sharpest matching definition of a function based on its input types helps achieve higher performance, by keeping the code execution paths tight and minimal.

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 4.4 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.

f	Function	Operand Types
Low Level "+"	Add Numbers	{Float , Int}
High Level "+"	Add Matrices	{Dense Matrix , Sparse Matrix}
" * "	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[14] instruction VADDSD[15] 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 3 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[16], 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[22]:
```
# 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[23]:
```
# 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[24]:
```
methods(⊕)
```

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

Out[24]:
```
4 methods for generic function ⊕:
⊕ (x::Int64,y::Int64) at In[23]:3
⊕ (x::Float64,y::Float64) at In[23]:4
⊕ (x::Int64,y::Float64) at In[23]:5
⊕ (x::Float64,y::Int64) at In[23]: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. We show how to use a dense algorithm in the implementation of $\oplus$ when either A or B (or both) are dense. A sparse algorithm is used when both A and B are sparse.

In[29]:
```
# 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 # Use Dense + Sparse
# 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
    return C
end
```

We now have eight methods for the function $\oplus$, four for the low level sum, and four more for the high level sum.

In[30]:
```
methods(⊕)
```

Out[30]:
```
8 methods for generic function ⊕:
⊕ (x::Int64,y::Int64) at In[23]:3
⊕ (x::Float64,y::Float64) at In[23]:4
⊕ (x::Int64,y::Float64) at In[23]:5
⊕ (x::Float64,y::Int64) at In[23]:6
⊕ (A::Array{T,2},B::Array{T,2}) at In[29]:1
⊕ (A::Array{T,2},B::AbstractSparseArray{Tv,Ti,2}) at In[29]:1
⊕ (A::AbstractSparseArray{Tv,Ti,2},B::Array{T,2}) at In[29]:1
⊕ (A::AbstractSparseArray{Tv,Ti,2},B::AbstractSparseArray{Tv,Ti,2}) at
In[29]:2
```

Figure 3: 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` or `http://en.wikipedia.org/wiki/X86_instruction_listings` one can really learn to see the details of code selection in action at the lowest levels. More importantly one can begin to understand that Julia is fast because the assembly code produced is so tight.

In[25]:
```
f(a,b) = a + b
```

Out[25]:
```
f (generic function with 1 method)
```

In[26]:
```
# Ints add with the x86 add instruction
@code_native f(2,3)
```

Out[26]:
```
push RBP
mov RBP, RSP
add RDI, RSI
mov RAX, RDI
pop RBP
ret
```

In[27]:
```
# Floats add, for example, with the x86 vaddsd instruction
@code_native f(1.0,3.0)
```

Out[27]:
```
push RBP
mov RBP, RSP
vaddsd XMM0, XMM0, XMM1
pop RBP
ret
```

In[28]:
```
# Int + Float requires a convert to scalar double precision, hence
# the x86 vcvtsi2sd instruction
@code_native f(1.0,3)
```

Out[28]:
```
push RBP
mov RBP, RSP
vcvtsi2sd XMM1, XMM0, RDI
vaddsd XMM0, XMM1, XMM0
pop RBP
ret
```

4.3 The many levels of code selection

In Julia as in mathematics, functions are as important as the data they operate on, their arguments. Perhaps even more so. We can create a new function `foo` and gave it six definitions depending on the combination of types. In the following example we sensitize unfamiliar readers with terms from computer science language research. It is not critical that these terms be understood all at once.

In[31]:
```
# 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[31]:
```
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**. The word "polymorphism" refers to the use of the same name (foo, in this example) for functions with different types. 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. That we use "+" for ints,floats, dense matrices, and sparse matrices is the same name for different things. Methods are grouped into generic functions.

While mathematics is the art of giving the same name to seemingly different things, a computer has to eventually execute the right program in the right circumstance. Julia's code selection operates at multiple levels in order to translate a user's abstract ideas into efficient execution. A generic function can operate on several arguments, and the method with the most specific signature matching the arguments is invoked. It is worth crystallizing some key aspects of this process:

1. The same name can be used for different functions in different circumstances. For example, `select` may refer to the selection algorithm for finding the kth smallest element in a list, or to select records in a database query, or simply as a user-defined function in a user's own program. Julia's namespaces allow the usage of the same vocabulary in different circumstances in a simple way that makes programs easy to read.
2. A collection of functions that represent the same idea but operate on different structures are naturally referred to by the same name. Which method is called is based entirely on the types of all the arguments - this is multiple dispatch. The function `det` may be defined for all matrices at an abstract level. However, for reasons of efficiency, Julia defines different methods for different types of matrices, depending on whether they are dense or sparse, or if they have a special structure such as diagonal or tridiagonal.
3. Within functions that operate on the same structure, there may be further differences based on the different types of data contained within. For example, whether the input is a vector of Float64 values or Int32 values, the norm is computed in the same exact way, with a common

```
\* 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. ("Polymorphism" is the use of the same name for a function that may have different type 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 burden for numerical computing.

body of code, but the compiler is able to generate different executable code from the abstract specification.

4. Julia uses the same mechanism of code selection at the lowest and highest levels - whether it is performing operations on matrices or operations on bits. As a result, Julia is able to optimize the whole program, picking the right method at the right time, either at compile-time or run-time.

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

The method to be executed in Julia is not chosen by only one argument, which is what happens in the case of single dispatch, but through multiple dispatch that considers the types of all the arguments. 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 data 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. However a moment's thought shows that the following dynamic

situation in Julia is impossible to express in Java:

In[32]:
```
# It is possible for a static compiler to know that x,y are Float
x = rand(Bool) ?  1.0 :  2.0
y = rand(Bool) ?  1.0 :  2.0
x+y

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

Readers are familiar with single dispatch mechanism, as in Matlab. 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. 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) Julia has proven that this is remarkably well suited for numerical 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 `sub` 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 numerical 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?

4.5 Quantifying the use of multiple dispatch

In [4] we performed an analysis to substantiate the claim that multiple dispatch, an esoteric idea for numerical computing 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 4.5 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 numerical computing appears to have significantly favorable characteristics compared to previous applications.

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 [31]. The results indicate that Julia is using multiple dispatch far more heavily than previous systems.

To quantify how heavily a language feature is used, we use the following metrics for evaluating the extent of multiple dispatch [31]:

1. Dispatch ratio (DR): The average number of methods in a generic function.
2. Choice ratio (CR): For each method, the total number of methods over all generic functions it belongs to, averaged over all methods. This is essentially the sum of the squares of the number of methods in each generic function, divided by the total number of methods. The intent of this statistic is to give more weight to functions with a large number of methods.
3. Degree of specialization (DoS): The average number of type-specialized arguments per method.

Table 4.5 shows the mean of each metric over the entire Julia `Base` library, showing a high degree of multiple dispatch compared with corpora in other languages [31]. Compared to most multiple dispatch systems, Julia functions tend to have a large number of definitions. To see why this might be, it helps to compare results from a biased sample of common operators. These functions are the most obvious candidates for multiple dispatch, and as a result their statistics climb dramatically. Julia is focused on numerical computing, and so is likely to have a large proportion of functions with this character.

4.6 Case Study for Numerical Computing

The complexity of linear algebra software has been nicely captured in the context of LAPACK and ScaLAPACK by Demmel and Dongarra, et.al., [13] 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)
```

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.

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 transcend any one data type. One must be able to write a general implemenation and as long as the necessary operations are available, the code should just work [20]. That is the power of code reuse.

4.6.1 Determinant: Simple Single Dispatch

In traditional numerical 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 a QR decomposition of the matrix and find the determinant as the product of the diagonal elements of R.[17] For symmetric tridiagonals the usual 3-term recurrence formula[38] is used. (The first four are defined as one line functions; the symmetric tridiagonal uses the long form.)

In[33]:
```
# 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)^size(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 inside the method.

4.6.2 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`,

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

`SymTridiagonal`, `Bidiagonal`, `Tridiagonal`, `Diagonal`, and `Triangular` are all examples of Julia's matrix structures.

The operations on these matrices exist as Julia functions. 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 functions and types. 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. We build on the symmetric arrow type introduced in Section 3.3.

In[34]:
```
# Define a Symmetric Arrow Matrix Type
immutable SymArrow{T} <:  AbstractMatrix{T}
    dv::Vector{T} # diagonal
    ev::Vector{T} # 1st row[2:n]
  end
```

In[35]:
```
# Define its size
importall Base
size(A::SymArrow, dim::Integer) = size(A.dv,1)
size(A::SymArrow)= size(A,1), size(A,1)
```

Out[35]:
```
size (generic function with 52 methods)
```

In[36]:
```
# 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[36]:
```
getindex (generic function with 168 methods)
```

In[37]:
```
# Dense version of SymArrow
full(A::SymArrow) =[A[i,j] for i=1:size(A,1), j=1:size(A,2)]
```

Out[37]:
```
full (generic function with 17 methods)
```

In[38]:
```
# An example
S=SymArrow([1,2,3,4,5],[6,7,8,9])
```

Out[38]:
```
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[39]:
```
# 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[39]:
```
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 some numerical 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`. (See Section 4.6.1.) 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, i.e., the selection of code based on the type of a single argument. We can now turn to a 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[40]:
```
# 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 Leveraging language design for high performance libraries

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.

5.1 Integer arithmetic

A simple but crucial example of a performance-critical language design choice is integer arithmetic. Julia uses machine arithmetic for integer computations.

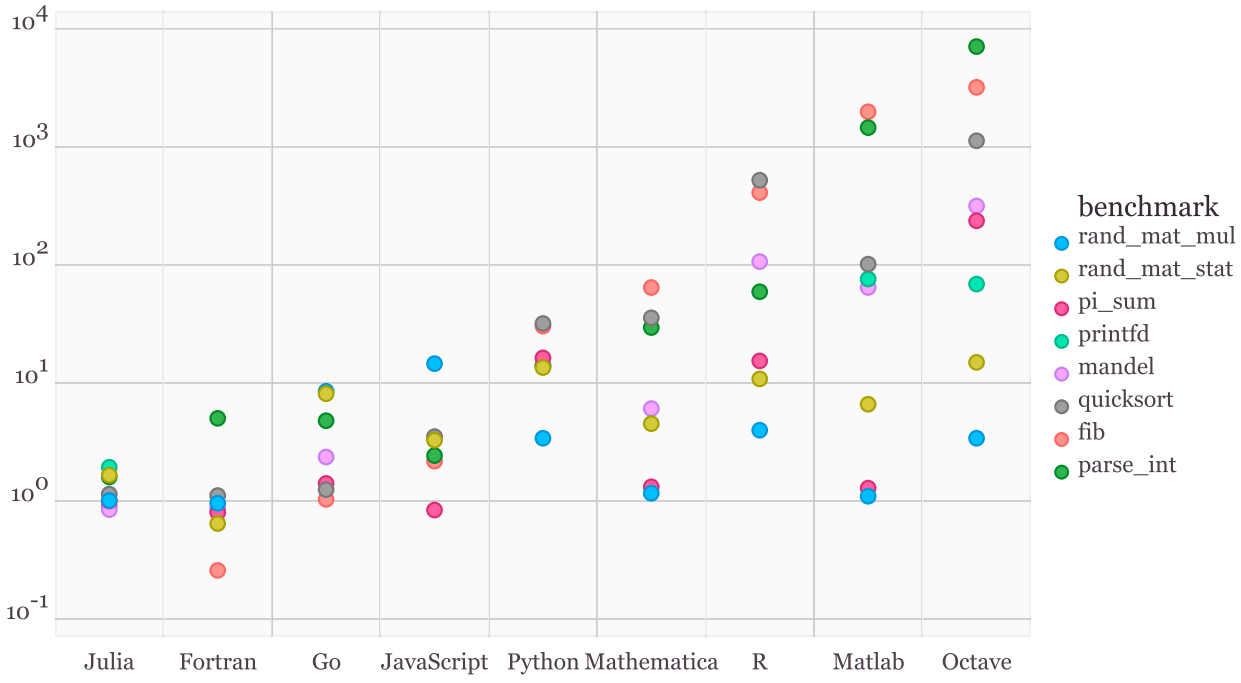


Figure 5: Performance comparison of various language performing simple micro-benchmarks. Benchmark execution time relative to C. (Smaller is better, C performance = 1.0).

Consider what happens if we make the number of loop iterations fixed:

In[41]:
```
# 10 Iterations of f(k)=5k-1 on integers
function g(k)
    for i = 1:10
      k = f(k)
    end
    k
end
```

Out[41]:
```
g (generic function with 2 methods)
```

In[42]:
```
code_native(g,(Int,))
```

Out[42]:
```
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 $f^{(10)}(k) = -2441406 + 9765625k$.

5.2 A powerful approach to linear algebra

We describe how the Julia language features have been used to provide a powerful approach to linear algebra[20].

5.2.1 Matrix factorizations

For decades, orthogonal matrices have been represented internally as products of Householder 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 [20] 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 4×3 matrix. What is the size of Q? The right answer, of course, is that it depends: it could be 4×4 or 4×3. The underlying representation is the same.

In Julia one can compute `Aqr = qrfact(rand(4,3))`. Then one extract Q from the factorization with `Q=Aqr[:Q]`. This Q retains its clever underlying structure and therefore is efficient and applicable when multiplying vectors of length 4 or length 3, contrary to the rules of freshman linear algebra,

but welcome in numerical libraries for saving space and faster computations.

In[43]:
```
A=[1 2 3
1 2 1
1 0 1
1 0 -1]

Aqr = qrfact(A));
Q = Aqr[:Q]
```

Out[43]:
```
4x4 QRCompactWYQ{Float64}:
 -0.5  -0.5  -0.5
 -0.5  -0.5   0.5
 -0.5   0.5  -0.5
 -0.5   0.5   0.5
```

In[44]:
```
Q*[1,0,0,0]
```

Out[44]:
```
4-element Array{Float64,1}:
 -0.5
 -0.5
 -0.5
 -0.5
```

In[45]:
```
Q*[1, 0, 0]
```

Out[45]:
```
4-element Array{Float64,1}:
 -0.5
 -0.5
 -0.5
 -0.5
```

5.2.2 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`[18], 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

[18] `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[46]:
```
# 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))
```

5.3 High Performance Polynomials and Special Functions with Macros

Julia has a macro system that provides easy custom code generation, bringing a level of performance that is otherwise difficult to achieve. A macro is a function that runs at parse-time, and takes parsed symbolic expressions in and returns transformed symbolic expressions out, which are inserted into the code for later compilation.

For example, a library developer implemented an `@evalpoly` macro that uses Horner's rule to evaluate polynomials efficiently. Consider

In[47]:
```
@evalpoly(10,3,4,5,6)
```

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

In[48]:
```
macroexpand(:@evalpoly(10,3,4,5,6))
```

We reproduce the key lines below

Out[48]:
```
#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 as fast as possible. The overhead of implementing an explicit loop, accessing coefficients in an array, and possibly a subroutine call (if it is not inlined), is substantial compared to just inlining the whole polynomial evaluation.

Steven Johnson reports in his EuroSciPy notebook[19]

> This is precisely how erfinv is implemented in Julia (in single and double precision), and is 3 to 4 times faster than the compiled (Fortran?) code in Matlab, and 2 to 3 times faster than the compiled (Fortran Cephes) code used in SciPy.
>
> The difference (at least in Cephes) seems to be mainly that they have explicit arrays of polynomial coefficients and call a subroutine for Horner's rule, versus inlining it via a macro.

Johnson also used the same trick in his implementation of the digamma special function for complex arguments[20] following an idea of Knuth:

> 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 Easy and flexible parallelism

Parallel computing remains an important research topic in numerical computing. Parallel computing has yet to reach the level of richness and interactivity required for innovation that has been achieved with sequential tools. The issues discussed in Section 3.1 on the balance between the human and

[19] https://github.com/stevengj/Julia-EuroSciPy14/blob/master/Metaprogramming.ipynb
[20] https://github.com/JuliaLang/julia/issues/7033

the computer become more pronounced in the parallel setting. Part of the problem is that parallel computing means different things to different people:

1. At the most basic level, one wants instruction level parallelism within a CPU, and expects the compiler to discover such parallelism in the code. In Julia, this can be achieved explicitly with the use of the `@simd` primitive. Beyond that,
2. In order to utilize multicore and manycore CPUs on the same node, one wants some kind of multi-threading. Currently, we have experimental multi-threading support in Julia, and this will be the topic of a further paper. Julia currently does provide a `SharedArray` data structure where the same array in memory can be operated on by multiple different Julia processes on the same node.
3. Then, there is distributed memory, often considered the most difficult kind of parallelism. This can mean running Julia on anything between half a dozen to thousands of nodes, each with multicore CPUs.

In the fullness of time, there may be a unified programming model that addresses this hierarchical nature of parallelism at different levels, across different memory hierarchies.

Our experience with Star-P [9] taught us a valuable lesson. Star-P parallelism [40, 39] included global dense, sparse, and cell arrays that were distributed on parallel shared or distributed memory computers. Before the evolution of the cloud as we know it today, the user used a familiar front end (usually Matlab) as the client on a laptop or desktop, and connected seamlessly to a server (usually a large distributed computer). Blockbuster functions from sparse and dense linear algebra, parallel FFTs, parallel sorting, and many others were easily available and composable for the user. In these cases Star-P called Fortran/MPI or C/MPI. Star-P also allowed a kind of parallel for loop that worked on rows, planes or hyperplanes of an array. In these cases Star-P used copies of the client language on the backend, usually Matlab, octave, python, or R.

Our experience taught us that while we were able to get a useful parallel computing system this way, bolting parallelism onto an existing language that was not designed for performance or parallelism is difficult at best, and impossible at worst. One of our (not so secret) motivations to build Julia was to have the right language for parallel computing.

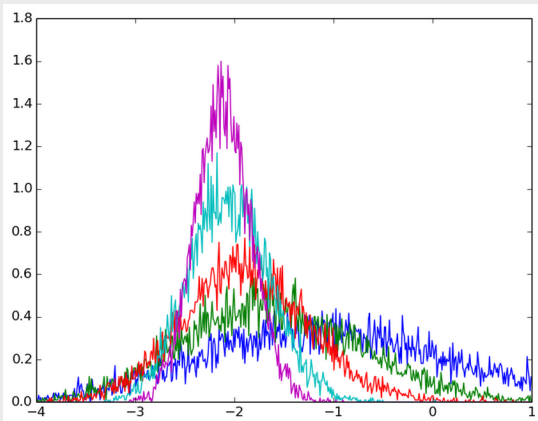
Julia provides many facilities for parallelism, which are described in detail in the Julia manual[21]. Distributed memory programming in Julia is built on two primtives - *remote calls* that execute a function on a remote processor and *remote references* that are returned by the remote processor to the caller. These primitives are implemented completely within Julia. On top of these, Julia provides a distributed array data structure, a `pmap` implementation, and a way to parallelize independent iterations of a loop with the `@parallel` macro - all of which can parallelize code in distributed memory. These ideas are exploratory in nature, and will certainly evolve. We only discuss them here to emphasize that well-designed programming language abstractions and primitives allow one to express and implement parallelism completely within the language, and explore a number of different parallel programming models with ease. We hope to have a detailed discussion on Juila's approach to parallelism in a future paper.

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

We proceed with one example that demonstrates `@parallel` at work, and how one can impulsively grab a large number of processors and explore their problem space quickly.

In[49]:
```
@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) +  2sqrt(h*β)*randn(N) # diagonal
  e=ones(N-1)/h^2                          # subdiagonal
  eigvals(SymTridiagonal(d,e))[N]     # smallest negative eigenvalue
end
end
```

In[50]:
```
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
```



Suppose we wish to perform a complicated histogram in parallel. We use an example from Random Matrix Theory, (but it could easily have been from finance), the computation of the scaled largest eigenvalue in magnitude of the so called stochastic Airy operator [15]

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

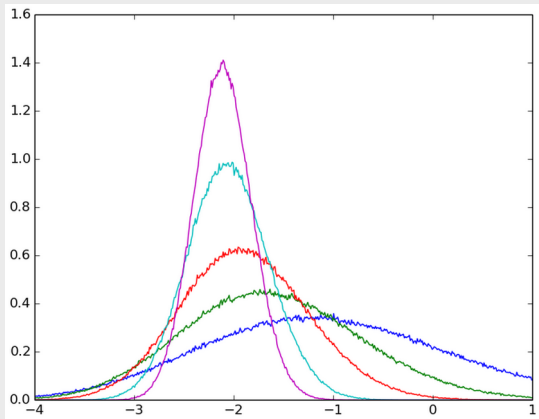
This is just the usual finite difference discretization of $\frac{d^2}{dx^2} - x$ with a "noisy" diagonal.

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 1024 processors, which is sharp and focused, and runs in exactly the same wall clock time as the sequential run. It is this ability of being able to perform scientific computation at the speed of thought conveniently without the traditional fuss associated with parallel computing, that we believe will make a new era of scientific discovery

possible.

In[51]:
```
# Readily adding 1024 processors sharpens the Monte Carlo simulation in
# the same time
addprocs(1024)
```

In[52]:
```
t = 10000
for β=[1,2,4,10,20]
  z = @parallel (+) for p=1:nprocs()
     hist([stochastic(β) for i=1:t], -4:.01:1)[2]
  end
  plot(midpoints(-4:.01:1),z/sum(z)/.01)
  end
end
```



5.5 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 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 numerical 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 vectorization is required to make use of special hardware capabilities such as SIMD instructions, multithreading, GPU units, and other forms of parallelism. This is not strictly true, as compilers are increasingly able to apply these performance features to explicit loops. The Julia message remains: vectorize when natural, when you feel it is right.

6 Conclusion and Acknowledgments

We built Julia to meet our needs for numerical computing, and it turns out that many others wanted exactly the same thing. At the time of writing, not a day goes by where we don't learn that someone else has picked up Julia at universities and companies around the world, in fields as diverse as engineering, mathematics, physical and social sciences, finance, biotech, and many others. 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. Numerical computing is maturing and it is exciting to watch!

Julia would not have been possible without the enthusiasm and contributions of the Julia community[22]. We thank Michael La Croix for his beautiful Julia display macros. We are indebted at MIT to Jeremy Kepner, Chris Hill, Saman Amarasinghe, Charles Leiserson, Steven Johnson and Gil Strang for their collegial support which not only allowed for the possibility of an academic research project to update technical computing, but made it more fun too. The authors gratefully acknowledge financial support from the MIT Deshpande center for numerical 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, grants from Aramco oil thanks to Ali Dogru and Shell oil thanks to Alon Arad, and a Citibank grant for High Performance Banking Data Analysis, and the Gordon and Betty Moore foundation.

References

[1] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The fortress language specification version 1.0. `http://research.sun.com/projects/plrg/fortress.pdf`, 2008.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[3] Jeff Bezanson. *Abstraction in Technical Computing*. PhD thesis, Massachusetts Institute of Technology, 2015.

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

[5] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. Julia: a Fast Dynamic Language for Technical Computing. arXiv:1209.5145v1, 2012.

[6] B.L. Chamberlain. A Brief Overview of Chapel. `http://chapel.cray.com/papers/ChapelCUG13.pdf`, 2013.

[7] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, August 2007.

[8] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, October 2005.

[9] Ron Choy and Alan Edelman. Parallel MATLAB: Doing it right. In *Proceedings of the IEEE*, volume 93, pages 331–341, 2005.

[10] Ron Choy, Alan Edelman, John R. Gilbert, Viral Shah, and David Cheng. Star-P: High productivity parallel computing. In *In 8th Annual Workshop on High-Performance Embedded Computing (HPEC 04)*, 2004.

[22] https://github.com/JuliaLang/julia/graphs/contributors

[11] Barry A. Cipra. The best of the 20th century: Editors name top 10 algorithms. SIAM News. `https://www.siam.org/pdf/news/637.pdf`.

[12] The Clang project. `http://clang.llvm.org/`.

[13] 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 Vomel, 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.

[14] Alan Edelman, Parry Husbands, and Steve Leibman. Interactive supercomputing's star-p platform: Parallel matlab and mpi homework classroom study on high level language productivity. In *Proceedings of the 10th High Performance Embedded Computing Workshop (HPEC 2006)*, 2006.

[15] Alan Edelman and Brian Sutton. From Random Matrices to Stochastic Operators. *Journal of Statistical Physics*, 127:1121–1165, 2007.

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

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

[18] Parry Husbands, Charles L. Isbell, Jr., and Alan Edelman. Interactive supercomputing with mitmatlab, 1998.

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

[20] Andreas Noack Jensen. Fast and generic linear algebra in Julia. Technical report, MIT, 2015.

[21] William Kahan. How futile are mindless assessments of roundoff in floating-point computation? `http://www.cs.berkeley.edu/~wkahan/Mindless.pdf`, 2006.

[22] 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.

[23] 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.

[24] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, September 1979.

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

[26] Miles Lubin and Iain Dunning. Computing in Operations Research using Julia. *INFORMS Journal on Computing*, 27(2):238–248, 2015.

[27] Mathematica. `http://www.mathematica.com`.

[28] Matlab. `http://www.mathworks.com`.

[29] Markus Mohnen. A Graph-Free Approach to Data-flow Analysis. In R. Horspool, editor, *Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 185–213. Springer Berlin / Heidelberg, 2002.

[30] Malcolm Murphy. Octave: A Free, High-Level Language for Mathematics. *Linux J.*, 1997, July 1997.

[31] 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.

[32] The Jupyter Project. `http://jupyter.org/`.

[33] The MPFR Project. `http://www.mpfr.org/`.

[34] The X10 project. http://x10-lang.org/.

[35] Rust. http://www.rust-lang.org/.

[36] Helen Shen. Interactive notebooks: Sharing the code, Nature Toolbox, Volume 515, Issue 7525, Nov 2014. http://www.nature.com/news/interactive-notebooks-sharing-the-code-1.16261.

[37] Guy Steele Jr. Parallel programming and code selection in fortress. In *"PPoPP '06 Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming"*, page 1, 2006.

[38] Gilbert Strang. *Introduction to Linear Algebra*. Wellesley-Cambridge Press, 2003.

[39] Interactive Supercomputing. Star-p user guide. http://www-math.mit.edu/~edelman/publications/star-p-user.pdf.

[40] Interactive Supercomputing. Getting started with star-p; taking your first test-drive. http://www-math.mit.edu/~edelman/publications.php, 2006.

[41] Swift. https://developer.apple.com/swift/.

[42] Madeleine Udell, Karanveer Mohan, David Zeng, Jenny Hong, Steven Diamond, and Stephen Boyd. Convex optimization in Julia. *SC14 Workshop on High Performance Technical Computing in Dynamic Languages*, 2014.

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

[44] Hadley Wickham. ggplot2. http://ggplot2.org/.

[45] Leland Wilkinson. *The Grammar of Graphics (Statistics and Computing)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.