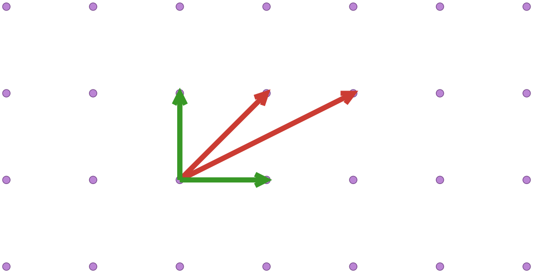
Lattice Reduction with LLLplus.jl

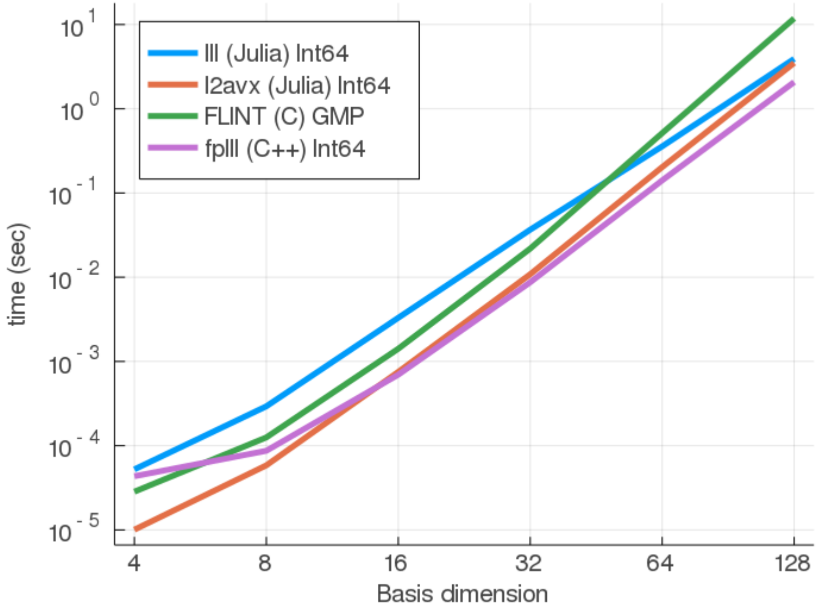
Chris Peel

Bay Area Julia Users



April 11, 2020

Preview: How do we compare w powerful C++ libraries?



When could I use lattice reduction?

Lattice reduction and other lattice tools are often used in places where one would normally use linear algebra, but an integer-valued solution is desired

There are practical uses:

- Integer programming
- Post-quantum cryptography (LWE)
- Cryptanalysis (cracking SSH, HTTPS)
- Digital communication
- Encrypted ML (FHE in Julia)
- Coding theory
- Finding anagrams :-)

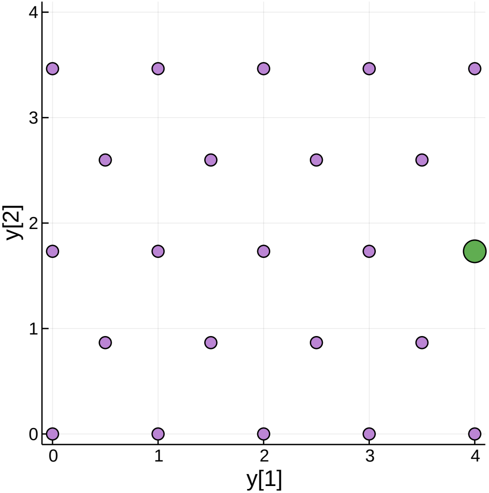
And there are theoretical uses

- Disproving Merten's Conjecture
- Sphere packing (with Julia!)
- Diophantine equations
  - Solving $x^3 + y^3 + z^3 = d$
- Geometry of flat tori
- Finding Spigot formulas
- Factoring Polynomials
- Computing the Riemann theta function

Outline

- Background
  - Basics, Definitions
  - Lattice Reduction
  - Lattice Problems
- `LLLplus.jl`
  - What's in it?
  - Performance
- Demo Applications
  - Cracking Cryptosystems
  - Finding spigot formulas
- Summary

What's a Lattice?



- A full-rank discrete additive subgroup of (say) $\mathbb{R}^n$ or $\mathbb{C}^n$
- $\mathbb{Z}^n$ is a lattice in $\mathbb{R}^n$
- Gaussian integers are a lattice in $\mathbb{C}^n$

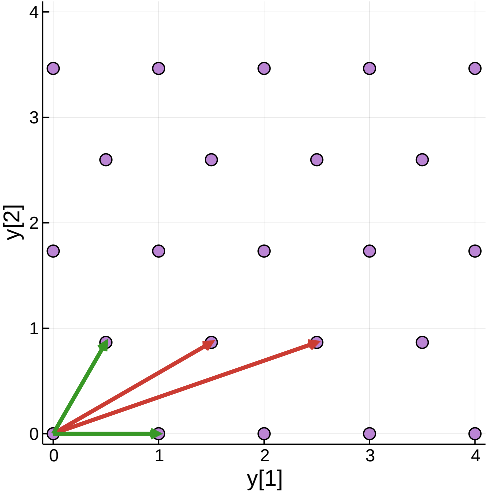
A practical definition

For a basis matrix B, and a vector of integers $\mathbf{z}$, the set of points $\mathbf{y}$ reachable by $\mathbf{y} = B\mathbf{z}$ is a lattice

How did you generate the green lattice point?

```
julia> B=[2.5      1.5
         0.866025 0.866025]  # hexagonal lattice
2×2 Array{Float64,2}:
 2.5       1.5
 0.866025  0.866025
julia> zgreen=[1
               1]
2-element Array{Int64,1}:
 1
 1
julia> ygreen=B*zgreen
2-element Array{Float64,1}:
 4.0
 1.73205
julia> Pkg.add("Plots"); using Plots;
julia> plot([ygreen[1]],[ygreen[2]], markershape = :circle,
            markersize = 10,
            markercolor = RGB(0.376, 0.678, 0.318))
```

For a lattice, how many bases are possible?



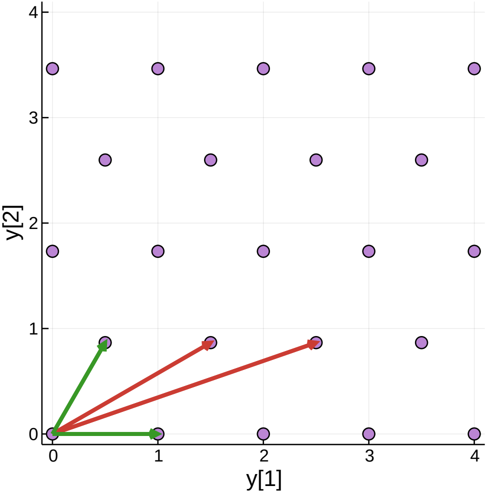
There are an infinite number of bases for a lattice; which one should we use?

$$Br = \begin{bmatrix} 2.5 & 1.5 \\ .86602 & .86602 \end{bmatrix}$$

$$Bg = \begin{bmatrix} 1.0 & .5 \\ 0.0 & .86602 \end{bmatrix}$$

In many problems, we want a short, close-to-orthogonal basis, like the green basis

How are different lattice bases related?



A **unimodular matrix** is square, integer-valued, and has determinant ±1. Its inverse is also unimodular

Relation between Bases

Every pair of lattice bases B_1 and B_2 are related by a unimodular matrix T: $B_1 = B_2 T$

For the bases in the figure, $B_{red} = B_{green} T$, where

$$T = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}$$

Can you illustrate bases and unimodular matrices?

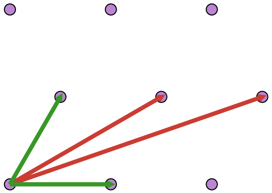
```
julia> Br=[2.5    1.5; 0.866025 0.866025];

julia> Bg=[1.0     .5; 0.0       .866025 ];

julia> T = inv(Bg)*Br       # try `Int.(T)`
2×2 Array{Float64,2}:
 2.0  1.0
 1.0  1.0

julia> det(T)
1.0

julia> Ti = inv(T)
2×2 Array{Float64,2}:
  1.0  -1.0
 -1.0   2.0

julia> det(Ti)
1.0
```

How can we find a good basis?

Use lattice reduction

Given lattice with basis B_1, the goal of lattice reduction is to find another basis B_2 for the same lattice which has short, closer-to-orthogonal basis vectors

Often, “short” and “orthogonal” are defined according to the Euclidian norm. So $B_2^T B_2$ is closer to diagonal than $B_1^T B_1$, and the diagonal elements of $B_2^T B_2$ are smaller than those of $B_1^T B_1$



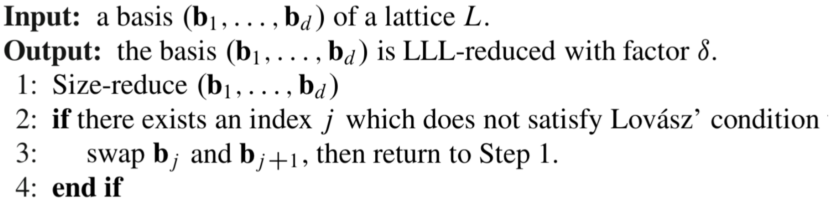
‘Lattice reduction is like QR for integer problems.’
Jack Poulson

Instead of an orthonormal Q, we have a close-to-orthogonal reduced basis, and instead of a triangular R we have a unimodular matrix: $B_1 = B_2 T$

How does one do lattice reduction?

The most important lattice reduction technique is from Lenstra, Lenstra, and Lovász[1], known as the LLL algorithm

LLL in pseudocode



```
Input: a basis $(\mathbf{b}_1, \ldots, \mathbf{b}_d)$ of a lattice $L$.
Output: the basis $(\mathbf{b}_1, \ldots, \mathbf{b}_d)$ is LLL-reduced with factor $\delta$.
1: Size-reduce $(\mathbf{b}_1, \ldots, \mathbf{b}_d)$
2: if there exists an index $j$ which does not satisfy Lovász' condition
3:     swap $\mathbf{b}_j$ and $\mathbf{b}_{j+1}$, then return to Step 1.
4: end if
```

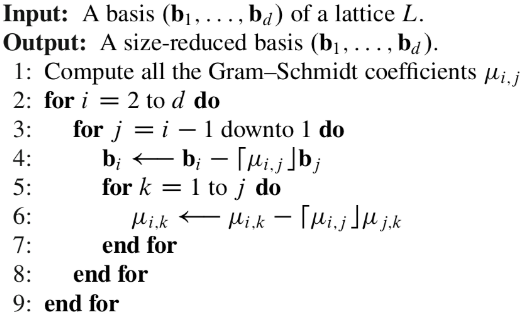
Lovász' condition is $||\mathbf{b}_{j+1}||^2 \geq (\delta - \mu_{j+1,j}^2)||\mathbf{b}_j||^2$ where the coeficients μ are Gram-Schmidt coefficients from size reduction

[1] A. K. Lenstra; H. W. Lenstra Jr.; L. Lovász; "Factoring polynomials with rational coefficients". Mathematische Annalen 261, 1982.

Size Reduction? Gram-Schmidt? Do I need to know this?

No, most LLL users can skip previous, current, next slides :-)

Size Reduction pseudocode[2]



```
Input: A basis $(\mathbf{b}_1, \ldots, \mathbf{b}_d)$ of a lattice $L$.
Output: A size-reduced basis $(\mathbf{b}_1, \ldots, \mathbf{b}_d)$.
Compute all the Gram–Schmidt coefficients $\mu_{i,j}$
for $i = 2$ to $d$ do
    for $j = i - 1$ downto 1 do
        $\mathbf{b}_i \longleftarrow \mathbf{b}_i - \lceil \mu_{i,j} \rfloor \mathbf{b}_j$
        for $k = 1$ to $j$ do
            $\mu_{i,k} \longleftarrow \mu_{i,k} - \lceil \mu_{i,j} \rfloor \mu_{j,k}$
        end for
    end for
end for
```

Size reduction is GS with rounding

There are LLL variants which use

- Gram-Schmidt (shown)
- Givens rotations
- Householder rotations
- A Cholesky decomposition (fastest)

[2] The LLL and size reduction pseudocode are from P. Q. Nguyen "Hermite's constant and lattice algorithms," a chapter of The LLL Algorithm, Springer, Berlin, Heidelberg, 2009, pp 19-69

Givens-based LLL in Julia

```
function lll(H::Matrix{Td},δ::Float64=3/4) where {Td<:Number}
    B = copy(H);   N,L = size(B);  _,R = qr(B)
    lx  = 2
    while lx <= L
        for k=lx-1:-1:1
            rk = R[k,lx]/R[k,k]
            mu = round(rk)
            if abs(mu)>0
                B[:,lx]   -= mu * B[:,k]
                R[1:k,lx] -= mu * R[1:k,k]
            end
        end
        nrm = norm(R[lx-1:lx,lx])
        if δ*abs(R[lx-1,lx-1])^2 > nrm^2
            B[:,[lx-1,lx]]    = B[:,[lx,lx-1]]
            R[1:lx,[lx-1,lx]] = R[1:lx,[lx,lx-1]]
            cc = R[lx-1,lx-1] / nrm
            ss = R[lx,lx-1]   / nrm
            Θ = [cc' ss; -ss cc]  # Givens rotation
            R[lx-1:lx,lx-1:end] .= Θ * R[lx-1:lx,lx-1:end]
            lx = max(lx-1,2)
        else; lx = lx+1; end
    end
    return B
end
```

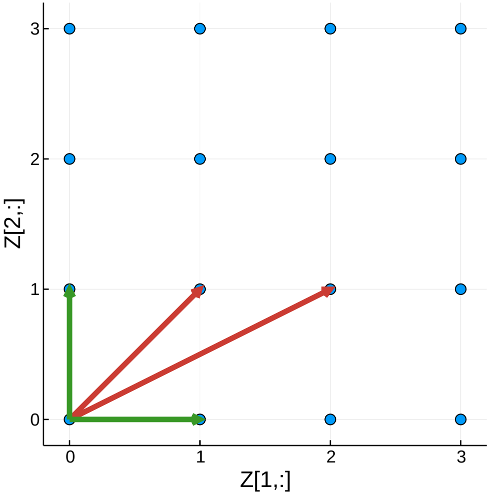
What should I remember about the LLL?

Remember two things:

- LLL runs in polynomial time; as fast as $O(d^5)$ for bases of size d
- LLL reduces the basis: $||\mathbf{b}_1|| \leq (\frac{2}{\sqrt{4\delta - 1}})^{d-1}\lambda_1(\mathcal{L})$

The LLL is the baseline lattice tool. Its polynomial speed and acceptable reduction quality is what brought interest to lattice tools

What other types of lattice reduction are there?



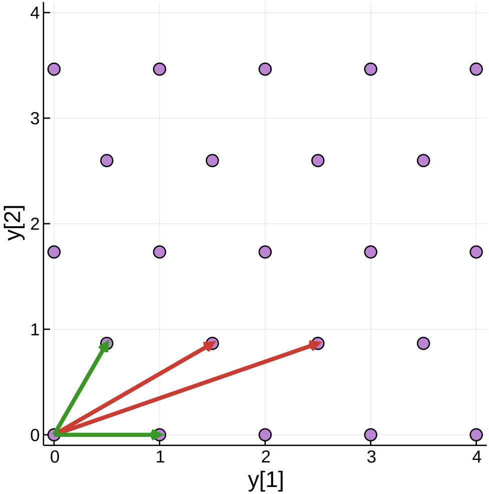
Seysen

Seysen lattice reduction simultaneously reduces a basis B and it's dual B^{-T}. It is not used as often as LLL reduction

Brun

Brun's technique for integer relations can be used for lower-quality, faster lattice reduction than LLL

Shortest Vector Problem: $\arg\min_{\mathbf{b}\in\mathcal{L},\mathbf{x}\neq\mathbf{0}} ||\mathbf{b}||$



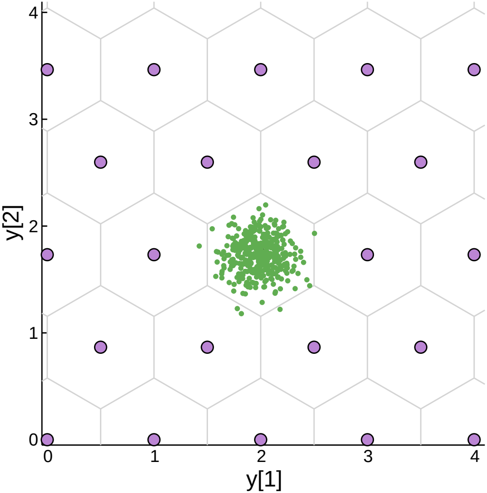
SVP

Find one of the shortest non-zero vectors in the lattice. Think of this as something like a vector-matrix equivalent of the greatest common divisor

Solving SVP exactly requires exponential time

LLL (and other lattice reduction techniques) approximate SVP in polynomial time

Closest Vector Problem: $\arg\min_{\mathbf{x}\in\mathbb{Z}^n} ||B\mathbf{x} - \mathbf{y}||$



CVP

Find the closest point in the lattice to a given vector, which is usually not a lattice point

Sovling CVP exactly requires exponential time

LLL can be used to approximate CVP in polynomial time

Outline

Background

- Basics, Definitions
- Lattice Reduction
- Lattice Problems

`LLLplus.jl`

- What's in it?
- Performance

Demo Applications

- Cracking Cryptosystems
- Finding spigot formulas

Summary

Lattice Tools in `LLLplus.jl`

Lattice Tool	Function	Use case
LLL lattice reduction	`lll`	most lattice problems
Seysen lattice reduction	`seysen`	math, WiFi
Brun lattice reduction	`brun`	math, WiFi
Lagrange/Gauss reduction	`gauss`	historical context
V-BLAST	`vblast`	WiFi, approx CVP
CVP solver	`cvp`	WiFi, GGH
SVP solver	`svp`	NTRU, RLWE

The `lll` tool is the most polished, everything else is less refined

There are also several auxiliary functions: `issizereduced`, `islllreduced`, `orthogonalitydefect`, `hermitefactor`, `seysencond`, `gen_qary_b`

Demos Functions, Applications

LLLplus.jl function	Application demoed
`subsetsum`	cryptanalysis, integer relations
`integerfeasibility`	integer programming feasibility
`rationalapprox`	find rational approx for vector
`spigotBBP`	spigot formulas for irrationals

The following packages also illustrate useful applications:

Other Packages	Application
MUMIMO.jl	broadband wireless
PolynomialFactors.jl	factor polynomials over integers
Theta.jl	compute Riemann theta function

How about an LLL demo?

```
julia> Br=[2.5    1.5; 0.866025 0.866025];
julia> Pkg.add("LLLplus"); using LLLplus

julia> B,T,_ = lll(Br); B
2×2 Array{Float64,2}:
 -1.0  -0.5
  0.0   0.866025

julia> T
 2×2 Array{Int64,2}:
 -1  -2
  1   3

julia> [det(T) det(inv(T))]
1×2 Array{Float64,2}:
 -1.0  -1.0

julia> islllreduced(B)
true
```

What types does LLLplus.lll work on?

LLLplus.lll works on bases over all Signed integers, AbstractFloats, Complex, and user-defined subtypes like BitIntegers. I've tried around 34 types.

```
julia> using BitIntegers, LLLplus
julia> B = rand(0:Int512(2)^33,2,2)+
         im*rand(0:Int512(2)^33,2,2)
2×2 Array{Complex{Int512},2}:
 5941420354+5486248041im  5574890144+3732896516im
 2538719538+1638107804im  3830374646+2133953247im
julia> Blll,Tlll = lll(B); Tlll
2×2 Array{Complex{Int512},2}:
 -1+0im  -1+2im
  1+0im   2-2im
```

To have LLLplus.lll work with a new type, check that LinearAlgebra.qr works, then add a method to LLLplus.getIntType for float types

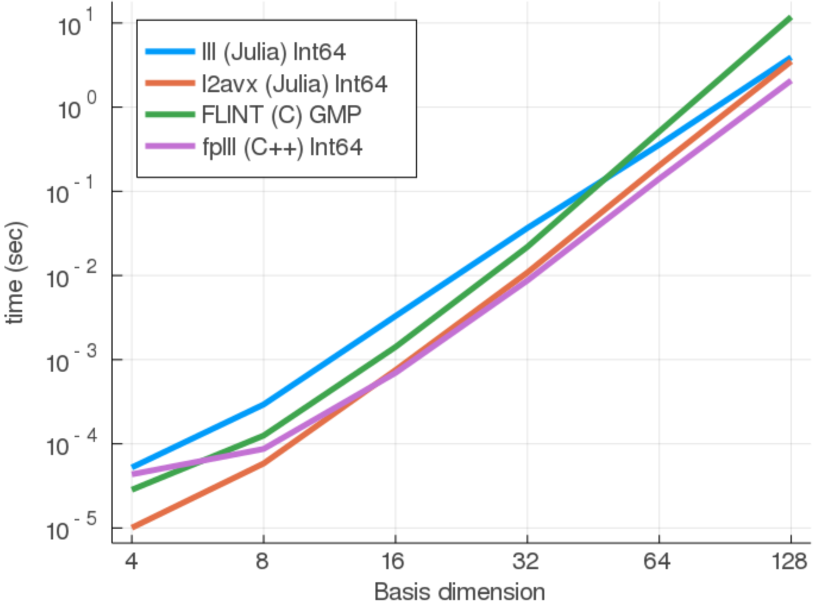
How can we characterize the performance of LLLplus.jl?

We'll focus on the LLL functions; it's the most polished tool. Earlier we said that (1) LLL runs in polynomial time and (2) reduces the basis. We'll start by comparing average execution time, then compare execution time with the quality of reduction

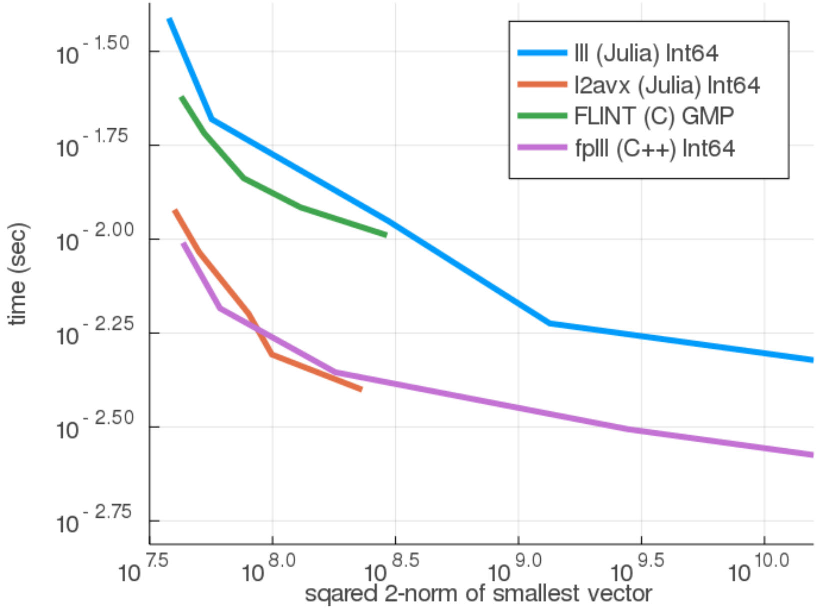
One of the prominent uses for fast lattice reduction is for cryptanalysis of cryptographically-tough bases. We will use `fplll`'s `gen_qary` function with parameter $b = 25$ bits to generate 20 random integer bases. The parameter b indicates the bit depth of the largest elements of the basis. Real cryptographic applications would require something more like $b = 45$ bits

We will compare against fplll, which is written in C++ by academic cryptographers and cryptanalysts. We'll also compare against Nemo.jl's lll function, which uses the FLINT C library written by number theorists

Execution time of LLLplus.jl is in the same ballpark as pros



Time v reduction quality also good for $N = 16$-dim bases



What optimizations did you use? Are there more to try?

I followed the Performance Tips in the Julia manual:

- Measure performance with `@time`
- Avoid global variables
- Profiling
- Break functions into multiple definitions
- Write "type-stable" functions

Other useful tools:

- LoopVectorization.jl
- Revise.jl

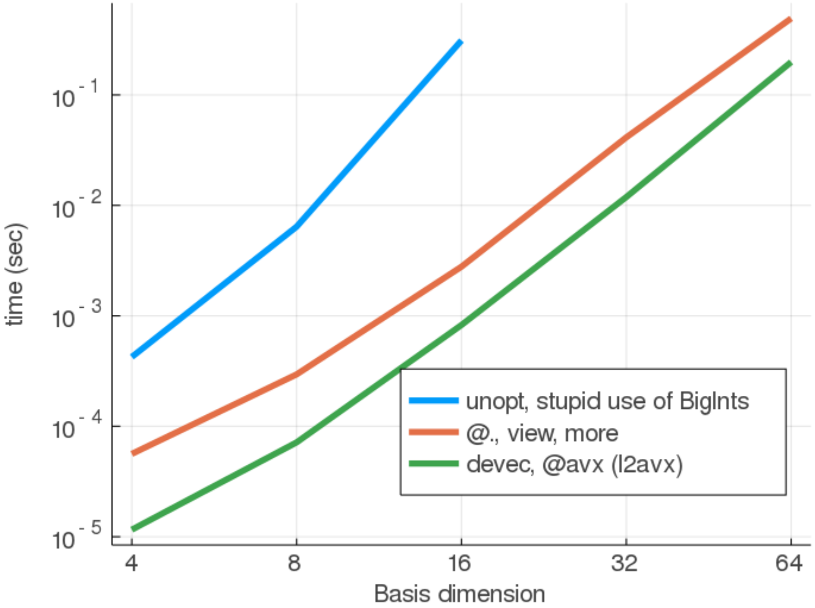
There's more that could be done

- Try widemul again
- Revisit `@fastmath`, `views`, `LoopVectorization.jl`, devectorization
- Try everything to the left again

Even so...

The current code is good enough, at least for now

How did those optimizations help?



An optimization wish

Can devectorization, `@inbounds`, `@simd`, and `@avx` somehow be done automatically? For example, could the following change from `LLLplus.lll` somehow be a compiler optimization?

```julia
for k=ll-1:-1:1
    rk = R[k,ll]/R[k,k]
    mu = round(rk)
    if abs(mu)>zeroTi
        # vectorized, easy-to-read
        # B[:,ll]   -= mu * B[:,k]
        # R[1:k,ll] -= mu * R[1:k,k]
        # T[:,ll]   -= mu * T[:,k]

        # devectorized, @simd
        @simd for n=1:N; B[n,ll]-= mu * B[n,k]; end
        @simd for n=1:k; R[n,ll]-= mu * R[n,k]; end
        @simd for n=1:L; T[n,ll]-= mu * T[n,k]; end
    end
end
```

Outline

Background

- Basics, Definitions
- Lattice Reduction
- Lattice Problems

`LLLplus.jl`

- What's in it?
- Performance

Demo Applications

- Cracking Cryptosystems
- Finding spigot formulas

Summary

Subset-Sum and Integer Relations

Subset-Sum

Given a vector $\mathbf{a}$ of integers, and a sum s, if there is a binary vector $\mathbf{x}$ such that $\mathbf{x}^T\mathbf{a} = s$, find it.

The LLL-based technique from Lagarias and Oldyzko was designed to solve low-density subset-sum problems. It breaks the Merkle–Hellman knapsack cryptosystem and is widely useful, for example to solve related problems like:

Integer Relations

Given a vector $\mathbf{a}$ of real numbers, if there is an integer vector $\mathbf{x}$ such that $\mathbf{x}^T\mathbf{a} = 0$, find it.

Integer relations solvers can be used to make spigot algorithms, say giving the nth digit of π without computing any of the other digits

A subset-sum problem JuMP can't solve

```
julia> setprecision(BigFloat,300); N=50; Bitdepth=190;
julia> # Bitdepth can be 256+, just doesn't fit on screen
       a=rand(0:2^BigInt(Bitdepth)-1,N);
julia> a[1:3]
3-element Array{BigInt,1}:
 911200129391658686469201173324473216271570073348033300075
 666563007748951582781404496296235427608875017772431026416
 832622399672004543019820919656212490862510181960061392073
julia> xtrue=rand(Bool,N); s=a'*xtrue;
julia> @elapsed x,_=LLLplus.subsetsum(a,s)
2.535546165
julia> s-x'*a
0.0
```

To solve this JuMP needs BigInt support, and likely a lattice solver

Ask me offline about cases where LLL is much faster than JuMP

Finding a spigot formula for π

Let's say we guess (somehow) that

$$\pi = \sum_{k=0}^{\infty} \frac{1}{b^k} \left(\frac{a_1}{(nk+1)^s} + \ldots + \frac{a_n}{(nk+n)^s} \right)$$

for $b = 16$, $n = 8$, and $s = 1$, how would we find the coeficients a?

Since it's tricky to sum to ∞, we'll start with first 45 terms.

LLLplus has a demo function for this:

```
julia> spigotBBP(BigFloat(pi),1,16,8,45,true);
A solution was found w error -4.728672e-60. In LaTeX it is
\alpha= \sum_{k=0}^\infty \frac{1}{16^k} \left(\frac{4}{8k+1}-
            \frac{2}{8k+4}-\frac{1}{8k+5}-\frac{1}{8k+6}\right)
```

In other words

$$\pi = \sum_{k=0}^{\infty} \frac{1}{16^k} \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right)$$

Discussion

> “I think it is safe to say that nobody really understands how the LLL algorithm works. The theoretical analyses are a long way from describing what ‘really’ happens in practice.” Victor Shoup

Possible Julia projects using lattice tools

- Write a `UnimodularMatrices.jl` package with fast determinant, inverse for unimodular matrix type
- Write solvers for integer programming and integer least squares problems, and connect the solvers to **JuMP**
- Write a lattice **cryptanalysis** package that could, for example, try to crack the data in Keno's encrypted ML tool
- Check to see if differential programming can enable fast solvers for SVP or CVP. Use these to inform the design of upcoming **post-quantum cryptography** standards
- Check to see if LLL can provide a **feasibility pump** for IPs

I'm happy to talk more about any of these offline

Lattice Problems

Lattice Problem	Definition	Uses
Shortest Vector Problem (SVP)	$\arg\min_{\mathbf{b}\in\mathcal{L},\mathbf{x}\neq\mathbf{0}} \vert\vert\mathbf{b}\vert\vert$	NTRU, RLWE
Closest Vector Problem (CVP)	$\arg\min_{\mathbf{x}\in\mathbb{Z}^n} \vert\vert B\mathbf{x}-\mathbf{y}\vert\vert$	WiFi, GGH
SVP$_\gamma$, GapSVP$_\beta$,CVP$_\gamma$, GapCVP$_\beta$, SIVP, BDD, SBP, SBP$_\gamma$,...		cryptography
Box constrained CVP	$\arg\min_{\mathbf{x}\in\mathbb{Z}^n,\mathbf{l}\leq\mathbf{x}\leq\mathbf{u}} \vert\vert B\mathbf{x}-\mathbf{y}\vert\vert$	WiFi, JuMP?
Mixed integer problems	$\arg\min_{\mathbf{z}\in\mathbb{R}^k,\mathbf{x}\in\mathbb{Z}^n} \vert\vert B\mathbf{x}+A\mathbf{z}-\mathbf{y}\vert\vert$	JuMP?