Rust-CAS

A Computer Algebra System written in Rust Language

Sory, J.A

May 22, 2021

Rust-CAS Authors

Colophon
This document was typeset with the help of KOMA-Script and LaTeX using the kaobook class.
The source code of this book is available at:
`https://github.com/fmarotta/kaobook`
(You are welcome to contribute!)

Publisher
First printed in May 2021 by Rust-CAS Authors

I am, somehow, less interested in the weight and convolutions of Einstein's brain than in the near certainty that people of equal talent have lived and died in cotton fields and sweatshops.

– Stephen Jay Gould

Contents

List of Figures

List of Tables

Introduction 1

1.1 What is Rust-CAS

Rust-CAS is a learning project whose purpose is to improve the authors skill with computer science and higher level mathematics in addition to produce useful software for the author and potentially others. This documentation exists primarily to provide information to others seeking to learn more advanced computing and a basic idea of implementation. This is not intended to be a competitor to the most serious computer algebra systems, but rather comparable in some ways and potentially more effective in others. The emphasis in this project is to be on number theorectic functions, multiprecision arithmetic, and sequence generation. The main goals and features are the following:

Arbitrary-precision Perform addition, subtraction, multiplication, division, exponentiation k-factorial, nthroots, and logarithms on arbitrary sized digits with greater than naive efficiency
Rationals Complex, polynomial, and arbitrary precision rationals
Complex Complex and hypercomplex numbers produced by Cayley construction. Rational, radical and arbitrary precision coefficients
Radicals Arbitrary precision, rational and complex radicals `\Sets` Generating sets for all Complex,Rational, Arbitrary-precision, Polynomial and Radical
Matrices Complex, Rational, and arbitrary precision matrix operations including addition/subtraction, multiplication, lu decomposition
Machine Logic Implementation of first-order logic to solve number theorectic and set theorectic problems.
Polynomials Addition, subtraction, multiplication, division, gcd, roots, interpolation, and grobner basis
Functions Generate sets from functions at runtime, in addition to producing derivatives integration and limits
Symbolic Solve symbolic equations

Why create Rust-CAS

The purpose of Rust-CAS and the documentation in this book is to address the absence of a unified computer algebra library in Rust and to provide an introductory description of various algorithms, techniques and approaches used in mathematical computing.

About the Author

The chief author is a computational physics major currently in pursuit of an Associates in Physics with plans on higher education. Their primary interests are computational mathematics, physics and computer algebra.

This Book

This Book is the documentation for the Rust-Computer Algebra System. The algorithms here are partly derived from textbooks and partly from the author. Algorithms derived from others will be credited, while algorithms derived from the author will simply have as detailed an explanation as I believe necessary as the author has not published any citable papers, nor do they intend to. Any algorithms derived by the author are almost certainly trivial enough that a formal paper is unnecessary.

The primary textbooks read by the author and utilized in this publication are as follows

Modern Computer Algebra Gathen et al
Abstract Algebra Dummit
Principles of Mathematical Analysis Rudin
First Order Logic and Theorem Proving Fitting

Additional inspiration comes from the Rust-Lang team whose work I took inspiration from while avoiding direct copying.

Green color refers to hardware instruction sets, blue refers to theorectical functions (pseudocode), red refers to real Rust-CAS functions and olive refers to real standard Rust functions

How to read the pseudocode used here. The authors utilize a variant of pseudocode to explain many of the algorithms and would like to establish how to read it for clarity. Below we give two examples of the pseudocode and an explanation.

```
  64_bit_binary_func(prev_carry,x,y)
       x_as_128bit <- x
       y_as_128bit <- y

       result_128_bit <- bin_op(x_as_128bit + prev_carry,
    y_as_128bit)

       carry <- result_128_bit/ 2^64
       lower_result <- result_128_bit mod 2^64
return (carry, lower_result)

```

At the top we have the function name 64_bit_binary_func. Next we have the assignment, x <- n refers assigning n to x making x = n. Then we have an assignment from the output of a binary function to result_128_bit, only the mapping of the function is stored i.e $f(x) = 2*x$ then 4 would be stored in the variable. One thing to note is that programming syntax is avoided as much as possible in this pseudocode so all arithmetic operations are represented mathematically. So 2ˆ 64 refers to exponentiation rather than the $\veebar$, aka XOR, operation like it does in programming languages. Additionally modular arithmetic is represented *mod n* rather than with the % sign. This function returns two values (yes you are mathematically allowed to do that, it's called a vector-valued function) and below we will see how that is handled here.

```
 bin_op_over_array(array_1, array_2)
   carry <- 0

```

```
for elements in array_1

 result_array<- lower_result from 64_bit_binary_func(carry,
   array_1,array_2)

 carry <- carry from 64_bit_binary_func(carry,array_1,array_2)
end
  if carry > 0
   then
    result <- carry+2^64

  return result_array

```

Here we have a binary function that takes arrays as it's arguments. We assign 0 to carry. Then we say that for every i in the first array we assign the value from the 64_bit_binary_function to the corresponding index in result_array. Notice the **from** keyword that shows that we are taking the lower_result from the 2-vector that is returned by the function. End signifies that the loop has ended and the if statement pushes back carry $+ 2^{64}$

1.2 Structure

Rust-CAS is currently structured by defining Additive and Multiplicative identities for a set amount of types. Rust-CAS does not implement all machine-supported types and is not intended too. We only implement i64,u64 and f64 primitive types, efficiency improvements in computation by using smaller datatypes are largely inconsequential. The current supported datatypes are, u64,i64,f64,Mpz,Matrix, Set, Rational, Complex, Quaternion, Radical, and Polynomial. Future supported types will be hypercomplex<n> algebraic datatypes, and functions.

Datatypes

Complex 2

2.1 Complex

Standard Complex Numbers

The complex numbers are numbers of the form $a + bi$, in Rust-CAS it is implemented as

```
struct Complex<T>{
 re: T,
 im: T,
}

```

Arithmetic

Real operations

Real_add To perform addition of a element of $\mathbb{R}$ with and element of $\mathbb{R}^2$ we only need to add the initial components together. I.e the real components, we ignore the imaginary component.

Real_sub To subtract a real from a complex we do the same as the addition.

Scalar To perform a real and complex multiplication we multiply each component of the complex number with the real making it a scalar multiplication, hence the name.

Real_div To divide a complex by a real is simply the reverse. Perform scalar multiplication of the complex with the inverse of the real.

Pow Exponentiation of a complex by a real is probably the most complex operation here. In order to do that we must first define the complex multiplication, see below for a description. Then we perfrom the multiplication n times where n is the real number. It is currently defined only for integer powers.

Complex operations

Transformations

Number Theory

Gaussian Integers

Eisenstein Integers

Quaternions

Arithmetic

Hurwitz Quaternions

Cayley Construction

Mpz | 3

The Mpz {Multiprecision $\mathbb{Z}$} class is a struct representing the integers to an arbitrary length. It utilizes an array of unsigned 64-bit integers to represent the number and an enum to represent the sign. Instead of using radix-10^n elements, we use radix 2^{64}. There are several efficiency advantages to this. 1. You can use larger numbers, if you are multiplying 2 64-bit numbers in a 128-bit register the largest numbers you can multiply is $2^{64} - 1$ vs 10^{19} the former of which is 1.8 times larger meaning you need only a little more than half the arithmetic operations for the same size number. 2. You can speed up certain operations simply by shifting, and 2 is a much more common factor than 10. We will see this once we get to the arithmetic section. The advantage radix-10^{19} has going for it is human readibility, which for very large integers is largely pointless. Therefore no attempt has been made to implement radix-10^{19} arithmetic and there will likely be no implementation to convert the base representation to a human readable one. The source code is below.

```
struct Mpz{
      sign: Sign,
      elements: Vec<u64>,
}
```

3.1 Methods

Now that we have a defined structure to represent the integers, we have to define operations on it. We will cover the interesting ones in detail, and the rest will simply be listed.

Shift

Shift_left is a useful operation in machine-sized integers and remains so for array-sized integers, we will see why later. However implementing it is going to be tricky. First, we need to find how many bits to shift left, then figure out what they are and carry them over to the integer ahead. One way to find the bit value at the k-place is if $n \veebar 2^k > n$ this works by flipping the bit in the k-place to 1 and checking if it is larger than it was before. If it was then the bit in the k-place was zero. For example $101010 \veebar 100 \longrightarrow 101110$ which is a greater number than 101010. Now that we have found how to find the values of each place we need to be able to transfer them over to the next integer.

To do that we only need to $\veebar$ back the values of 2^k when it was a part of the sum of n, we just need to do it at the lower bound instead of the upper bound. As n can be represented as $2^{k_i n} + 2^{k_{i-1} n} \ldots$

Here is an algorithm for the carry-left-shift

```
carry_shift_left(x , shiftlength)
carry <-  0
 for i in [1;shiftlength]
  if  xor(x, 1* 2^(63-i))  > x
   then
    carry = xor(carry,1*2^(shiftlength -(i+1)))
 end
    return (carry,x*2^shiftlength mod 2^64)
```

In Rust-CAS this is implemented utilizing the overflowing_shl function. The actual code is shown below [1]

1: In languages that do not have an intrinsic like overflowing_shl you can prevent overflow by xor 0 into the leading places and then left shifting.

```
 fn carry_shl(x: u64, places: u64)->(u64,u64){

    let  xclone = x.clone();
    let mut carry_value = 0u64;

for i in 0..places{
   if xclone^(1u64<<(63u64-i)) < xclone{
      carry_value^=1<<(places-1-i)
   }
}

(carry_value,xclone.overflowing_shl(places as u32).0)
}
```

In order to efficiently implement this left shift on an array we first need to handle the values greater than 64 with a better method than skipping through n/64 elements for each element we are shifting. We note that each place of 0 is equal to $0 \cdot 2^{n64}$, therefore to shift values we can shift the values n mod 64 and then insert n/64 elements at the start of the array. Insertion of course is very expensive, (O log n) per element inserted, so the reverse is much faster. First, collect a vector of n/64 zeroes and then pushback the shifted original vector afterwards. Shown below.

```
mpz_shl(mpz,shift)
   for i in [1;shift/64]
   shifted_mpz <- 0
   end
  for j in  mpz
    shifted_mpz <- shift from carry_shift_left(mpz[j],shift mod
   64)

    carry <- carry from carry_shift_left(mpz[j],shift mod 64)
  end

  return shifted_mpz
```

Note the similarity of this form to the general binary function over an array shown in the Arithmetic section.

Bitwise

Logarithm

Other

3.2 Arithmetic

To implement arithmetic over a $2^{64} - 1$ array we have several obstacles. The main one is that you cannot perform operations on values larger than $2^{64} - 1$ and the arithmetic operations require numbers that exceed that bound at some point before they can be reduced. The general work around is to cast to a larger datatype, typically 128-bit unsigned integer and perform the arithmetic and then convert back to 64-bit. A general algorithm showing this is below, using bin_op to represent a binary operation such as addition, subtraction, or multiplication.

```
64_bit_binary_func(carry,x,y)
   x_as_128bit <- x
   y_as_128bit <- y

  result_128_bit <- bin_op(x_as_128bit + prev_carry,y_as_128bit)

  carry <- result_128_bit/ 2^64
  lower_result <- result_128_bit mod 2^64

return (carry, lower_result)
```

To implement this over an array we start from the back and iterate through array taking the carry to the next element. Note that for subtraction we subtract the carry rather than add it. Here is the pseudocode for it.

```
bin_op_over_array(array_1, array_2)
  carry <- 0

for elements in array_1

 result_array<- lower_result from 64_bit_binary_func(carry,
   array_1,array_2)

 carry <- carry from 64_bit_binary_func(carry,array_1,array_2)
end
  if carry > 0
   then
    result <- carry+2^64

  return result_array
```

Addition

To implement addition for the multiprecision array we can implement the bin_op_over_array for addition. However, we can do much better than that, modern cpus have instructions that permit carry_adds with no additional cost. In Rust-CAS we utilize the _addcarry_u64 instruction. This instruction takes two unsigned 64-bit integers adds the carry flag and the one of the integers to the other in-place, and then returns the carry flag from that addition. Pseudocode implementation is below.

```
  add_mpz(mpz_1,mpz_2)
  carry <- 0

    for i in mpz_1 and j in mpz_2
       carry <-  _addcarry_u64(carry,i,j,i)
    end
    return mpz_1

```

To handle integers of different size, we can split the array into two arrays, the first one of which is the same length as the other. Then only perform addition between the two subarrays and then add the carry through the higher . This is the strategy used in the Rust BigInt library. However in most cases splitting the array costs more than simplying adding leading zeroes until the numbers are of equal length and then performing the addition-carry through the array. This is the strategy used in Rust-CAS and shown below.

```
 add_unequal_mpz(mpz_1,mpz_2)

  if mpz_1 len > mpz_2 len
   for i in [0;mpz_1-mpz_2]
    mpz_2 <-0
   end

  if mpz_1 len > mpz_2 len
   for i in [0;mpz_2-mpz_1]
    mpz_1 <-0
   end

   return add_mpz(mpz_1, mpz_2)



```

Subtraction

Multiplication

Classical

Toom-k

Not implemented but a future description and or implementation will be here.

Schonhage-Strassen

Not implemented but a future description and or implementation will be here.

Furer

Not implemented but a future description and or implementation will be here.

N log n or the Harvey-Hoeven algorithm

Not implemented but a future description and or implementation will be here.

Division

Euclidean

Shifting Division

Standard

Not implemented but a future description and or implementation will be here.

Exponentiation

Standard

Modular

Not implemented but a future description and or implementation will be here.

`\marginnote[-12pt]{Text} or \marginnote[*-3]{Text}`

3.3 Advanced Functions

Primality

Factorization

Rational 4

Polynomial 5

The Polynomial datatype is a representation of multivariate polynomials. A key observation to make when handling multivariates is to recognize that they can be composed of the dot products of the the univariate components. For instance given a polynomial $P(x)$ where $P(x) = 15x^3y^2 + 3x^2y$ we can break it down into a representation of three vectors the real vector the x vector and there powers and the y vector and it's degree. Like so $P(x) = \mathbb{R}$ = {15,3} x degrees {3,2} y degrees={2,1} and the variables {x} and {y}. The standard representation utilizes splitting the polynomial into univariates and having each degree symbolized by the bit value of the place. For instance in the above example we would represent $15x^3 + 3x^2$ as {15,3,0} as we have $15 \cdot x^3$ we place 15 into the third place and then 3 into the second place and nothing in the third as there is no x^0. This representation makes addition and subtraction very easy as it is only a matter of adding values at the indexes, multiplication therefore becomes a shift and add to the next element. Like so, {15,3,0} · {3,2} = {45,9,0,30,6,0}. However, this is not the schema utilized in Rust-CAS. Rather, we use a slightly different representation. P(x) would be instead represented as {15,3,2,3,2,1}. Here we have the polynomial represented as {$\mathbb{R}$,$\mathbb{F}_{[x]}$,$\mathbb{F}_{[y]}$ }. While this does complicate some of the operations somewhat, this representation was chosen due to it's scalability. Instead of needing a minimum of n bytes to represent a polynomial of n-degree, our datatype size is instead bounded by the number of variables and nomials. Currently the polynomial uses 64-bit signed integer so it's degree is bounded by $2^{63} - 1$ (this would be several terabytes in standard representation). As we can see this representation allows more efficient manipulation of very large polynomials. If we implement Mpz for the polynomial then the bounds could be effectively infinite. The struct code is shown below

```
struct Polynomial{

 dimen: (usize,usize) // First elements represents the number of
    variables, second element the number of nomials
 elements: Vec<i64>
}
```

5.1 Composition and Decomposition

Array 6

Nearly all of Rust-CAS's datatypes are based on arrays, however the complexity is simple enough to collect into a single section.

6.1 Matrix

Matrix is a datatype to represent matrices. Utilizing indexing one can use a one-dimensional vector to represent a 2-dimensional matrix.

Methods

Arithmetic

Matrix Reductions

6.2 Sets

Sets are an Array based datatype with various set-theorectic methods implemented and the ability to generate and store sequences. The actual structure is very simple just being an generic array.

```
struct Set<T>{
 elements: Vec<T>
}
```

The interesting results are from the generating functions.

Methods

Set Theory

Complement The relative complement of a set refers to the elements that are in one set but not the other. It is not commutative, meaning that $complement(\mathbb{A}, \mathbb{B}) \nsubseteq complement(\mathbb{B}, \mathbb{A})$

Set_union

Generic

card Returns the cardinality of the set, in this case the length. In order to produce true cardinality you have to reduce it to a formal set first, by removing all non-unique elements.

clear Clears the set but retains it's capacity, exists for convenience.

empty Clears the set and shrinks to zero elements. Equivalent to setting to $\emptyset$

new Initializes a new set, takes the argument of a vector.

Functions

Number Theorectic 7

7.1 Primality

Fast Primes

Computing small primes is largely an easy task however there are applications where you want to be able to compute small primes as fast as possible. For instance a fast approach to computing the factorial of n is to find the primes under n and perfrom exponentiation and multiply the results together (a full algorithm is given in the Mpz arithmetic section). This requires computing a small range of primes very rapidly.

The fastest way to get primes is to forgo any computation and use a look-up table. However if we decide to produce a table of the primes in the interval $[0; 10^6]$ we are left with a table of 78498 elements that take up 313992 bytes assuming that we are using 32-bits to store each number since 32-bit is the smallest register that can store 10^6. This is rather large for a small number of elements, even if we restrict ourselves to the interval $[0; 256]$ we get only 54 elements stored in 54 bytes.

Alternately we can store whether or not a number is prime as a boolean value in the index of the number. i.e for the first 8 integers in the interval $[0; 8]$ we have {false,true,true,false,true,false,true,false}. Booleans are stored as 8-bit/1-byte lengths and so this arrangement is quite a bit less efficient than the previous one. However we can observe that there are only two states in our boolean array and our number system is also comprised of two states, {0,1}.

With that observation we see that we can store our 8-byte boolean array into a single byte 01010110. This is a considerable improvement over our boolean array and even over a standard look-up table. To store all the values of the numbers up to 10^6 we can see that we will need $10^6 \div 8 = 125000$ bytes. This is a 2.5 times improvement in storage. As the density of primes decreases at greater intervals the efficiency is not linear and will in fact decrease. For instance the efficiency improvement is only $813120884 \div 536870912 \approx 1.5$ times at the interval $[0; 2^{64}]$.This does however increase once you have to use 64-bit registers to store your numbers, before it starts to decrease again.

To further improve our storage we can change the bits from mapping in 1:1 correspondance with the integers to a subset with higher prime density. The next subset with higher prime density than $\mathbb{Z}$ is $2\mathbb{Z} + 1$, aka the odd numbers. In this case then the rightmost bit in 01101110 would refer to the value of 2(0)+1, the next 2(1)+1 and so on. If you check the primality of 1,3,5,7,9,11,13,15 you will see that it corresponds with the values of the bits in each place. Currently Rust-CAS uses the set $2\mathbb{Z} + 1$ for simplicity and the fact that the benefits diminish the smaller the subset you map to. Future implementations may utilize smaller subsets, if computation is not an efficiency penalty.

The actual implementation is in the form of a 20000 element 64-bit array. In order to perform the lookup we find the corresponding index of the set $\mathbb{Z}$ in the set $2\mathbb{Z}+1$. For example if our inputs are {1,2,3,4,5} then our index in the set is {0,DNE [1] ,1,DNE,2}. To perform these conversion we simply take the input and subtract 1 and divide by 2. Then we need to check the value of the bit, we can use the strategy shown in shift_left or alternately shift right and check the value of the last bit. The source code is shown below.

1: DNE here refers to Does Not Exist as $2\mathbb{Z}+1 \subset \mathbb{Z}$

```
    const FASTER_PRIME: [usize;20000]=[
    9326130857677998958, ..., 324267973559730176];

    fn fast_prime(x:usize)->bool} //takes 64-bit argument
    if x ==2{
       return true
    }
    if x%2==0{
       return false
    }
let k=(x-1)>>1; //equivalent to (x-1)/2

    FASTER_PRIME[k>>6]>>(k%64)&1 !=0
}


```

The two if statements cover the cases of x == 2 and x being even. Then the value of k is assigned to be the value of . Next we obtain the index of the number to check by dividing the value of k by 64. Then we shift it by k mod 64 places and check the final digit. If it is not 0 then we say that it is true that the number is prime. For example to check if 1379 is prime, we first check if it is even or 2. We see that it is false in both cases. Next we see that 1379 is the $(1379-1)/2 = 689$th odd number. Then we divide 689 by 64 $689 >> 6 = 10$ to get the index of the number to check against. In this case the number is 13844351991940268577 the binary representation of which is 110000000010000100000010011001001010000001000100100100001000100001. Then we find the corresponding bit by shifting by 689 mod 64 = 49 places. Our number is now 110000000010000 and we can & with 1 to see that it is in fact even and that the statement "1379 is prime" is false. A brief benchmark shows that this implementation is approximately 8 times faster than a naive erastothenes and likely faster than the vaunted *primesieve* for short intervals.

Algebraic Functions 8

Geometry 9

9.1 Normal Figures and Tables

References 10

10.1 Citations

Issues and Future Implementations

Issues 11

11.1 Errors

11.2 Improvements

Benchmark 12

12.1 Computational and Memory Analysis

12.2 Benchmarks

Appendix

A
Heading on Level 0 (chapter)

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

A.1 Heading on Level 1 (section)

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Heading on Level 2 (subsection)

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Heading on Level 3 (subsubsection)

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift –

not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Heading on Level 4 (paragraph) Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

A.2 Lists

Example for list (itemize)

- First item in a list
- Second item in a list
- Third item in a list
- Fourth item in a list
- Fifth item in a list

Example for list (4*itemize)

- First item in a list
  - First item in a list
    - First item in a list
      - First item in a list
      - Second item in a list
    - Second item in a list
  - Second item in a list
- Second item in a list

Example for list (enumerate)

1. First item in a list
2. Second item in a list
3. Third item in a list
4. Fourth item in a list
5. Fifth item in a list

Example for list (4*enumerate)

1. First item in a list
    a) First item in a list
        i. First item in a list
            A. First item in a list
            B. Second item in a list
        ii. Second item in a list
    b) Second item in a list
2. Second item in a list

Example for list (description)

First item in a list
Second item in a list
Third item in a list
Fourth item in a list
Fifth item in a list

Example for list (4*description)

First item in a list

- **First** item in a list
    - **First** item in a list
        - **First** item in a list
        - **Second** item in a list
    - **Second** item in a list
- **Second** item in a list

Second item in a list

Notation

The next list describes several symbols that will be later used within the body of the document.

c	Speed of light in a vacuum inertial frame
h	Planck constant

Greek Letters with Pronounciation

Character	Name	Character	Name
α	alpha *AL-fuh*	ν	nu *NEW*
β	beta *BAY-tuh*	ξ, Ξ	xi *KSIGH*
γ, Γ	gamma *GAM-muh*	o	omicron *OM-uh-CRON*
δ, Δ	delta *DEL-tuh*	π, Π	pi *PIE*
ϵ	epsilon *EP-suh-lon*	ρ	rho *ROW*
ζ	zeta *ZAY-tuh*	σ, Σ	sigma *SIG-muh*
η	eta *AY-tuh*	τ	tau *TOW (as in cow)*
θ, Θ	theta *THAY-tuh*	υ, Υ	upsilon *OOP-suh-LON*
ι	iota *eye-OH-tuh*	ϕ, Φ	phi *FEE, or FI (as in hi)*
κ	kappa *KAP-uh*	χ	chi *KI (as in hi)*
λ, Λ	lambda *LAM-duh*	ψ, Ψ	psi *SIGH, or PSIGH*
μ	mu *MEW*	ω, Ω	omega *oh-MAY-guh*

Capitals shown are the ones that differ from Roman capitals.