Chapter 1

The Field

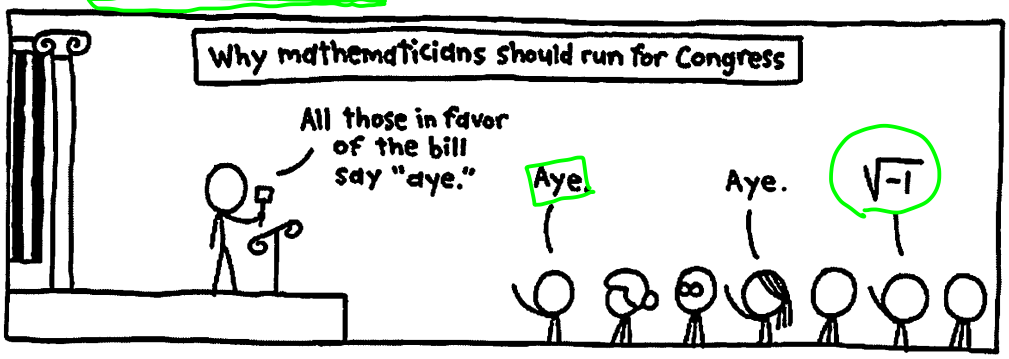
...the different branches of
Arithmetic—Ambition, Distraction,
Uglification, and Derision.

Lewis Carroll, *Alice in Wonderland*

We introduce the notion of a field, a collection of values with a *plus* operation and a *times* operation. The reader is familiar with the field of *real numbers* but perhaps not with the field of *complex numbers* or the field consisting just of zero and one. We discuss these fields and give examples of applications.

1.1 Introduction to complex numbers

If you stick to real numbers, there are no solutions to the equation $x^2 = -1$. To fill this void, mathematicians invented $\mathbf{i}$. That's a bold letter i, and it's pronounced "i", but it is usually defined as *the square root of minus 1*.



Guest Week: Bill Amend (excerpt, `http://xkcd.com/824`)

By definition,

$$\mathbf{i}^2 = -1$$

Multiplying both sides by 9, we get

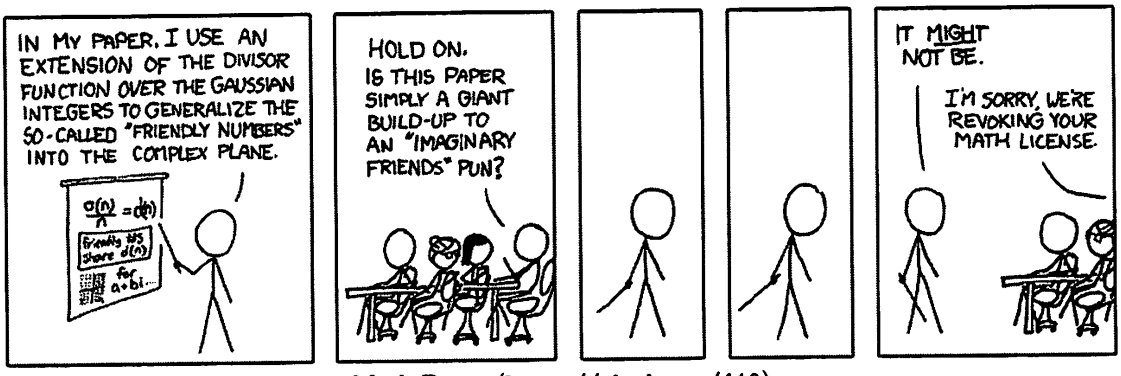
$$9\mathbf{i}^2 = -9$$

which can be transformed to

$$(3\mathbf{i})^2 = -9$$

Thus $3\mathbf{i}$ is the solution to the equation $x^2 = -9$. Similarly, for any positive number b, the solution to $x^2 = -b$ is $\sqrt{b}$ times $\mathbf{i}$. The product of a real number and $\mathbf{i}$ is called an *imaginary* number.

What about the equation $(x-1)^2 = -9$? We can solve this by setting $x - 1 = 3\mathbf{i}$, which yields $x = 1 + 3\mathbf{i}$. The sum of a real number and an imaginary number is called a *complex number*. A complex number has a *real part* and an *imaginary part*.



Math Paper (`http://xkcd.com/410`)

1.2 Complex numbers in Python

Python supports complex numbers. The square root of -9, the imaginary number 3i, is written `3j`.

```
>>> 3j
3j
```

Thus `j` plays the role of `i`. (In electrical engineering, i means "current")

The square root of -1, the imaginary number i, is written `1j` so as to avoid confusion with the variable `j`.

```
>>> j
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'j' is not defined
>>> 1j
1j
```

Since Python allows the use of + to add a real number and an imaginary number, you can write the complex solution to $(x-1)^2 = -9$ as `1+3j`:

```
>>> 1+3j
(1+3j)
```

In fact, the operators +, -, *, /, and ** all work with complex numbers. When you add two complex numbers, the real parts are added and the imaginary parts are added.

```
>>> (1+3j) + (10+20j)
(11+23j)
>>> x=1+3j
>>> (x-1)**2
(-9+0j)
```

Python considers the value `(-9+0j)` to be a complex number even though its imaginary part is zero.

As in ordinary arithmetic, multiplication has precedence over addition; exponentiation has precedence over multiplication. These precedence rules are illustrated in the following evaluations.

```
>>> 1+2j*3
(1+6j)
>>> 4*3j**2
(-36+0j)
```

You can obtain the real and imaginary parts of a complex number using a dot notation.

```
>>> x.real
1.0
>>> x.imag
3.0
```

It is not an accident that the notation is that used in object-oriented programming languages to access instance variables (a.k.a. member variables). The complex numbers form a class in Python.

```
>>> type(1+2j)
<class 'complex'>
```

This class defines the procedures (a.k.a. methods, a.k.a. member functions) used in arithmetic operations on complex numbers.

1.3 Abstracting over *fields*

In programming languages, use of the same name (e.g. +) for different procedures operating on values of different datatypes is called *overloading*. Here's an example of why it's useful in the present context. Let us write a procedure `solve1(a,b, c)` to solve an equation of the form $ax + b = c$ where a is nonzero:

```
>>> def solve1(a,b,c): return (c-b)/a
```

It's a pretty simple procedure. It's the procedure you would write even if you had never heard of complex numbers. Let us use it to solve the equation $10x + 5 = 30$:

```
>>> solve1(10, 5, 30)
2.5
```

The remarkable thing, however, is that the same procedure can be used to solve equations involving complex numbers. Let's use it to solve the equation $(10 + 5\mathrm{i})x + 5 = 20$:

```
>>> solve1(10+5j, 5, 20)
(1.2-0.6j)
```

The procedure works even with complex arguments because the correctness of the procedure does not depend on what kind of numbers are supplied to it; it depends only on the fact that the *divide* operator is the inverse of the *multiply* operator and the *subtract* operator is the inverse of the *add* operator.

The power of this idea goes well beyond this simple procedure. Much of linear algebra—concepts, theorems, and, yes, procedures—works not just for the real numbers but also for the complex numbers and for other kinds of numbers as well. The strategy for achieving this is simple:

- The concepts, theorems, and procedures are stated in terms of the arithmetic operators +, -, *, and /.
- They assume only that these operators satisfy certain basic laws, such as commutativity ($a + b = b + a$) and distributivity ($a(b + c) = ab + ac$).

Because the concepts, theorems, and procedures rely only on these basic laws, we can "plug in" any system of numbers, called a *field*.[1] Different fields arise in different applications.

In this book, we illustrate the generality of linear algebra with three different fields.

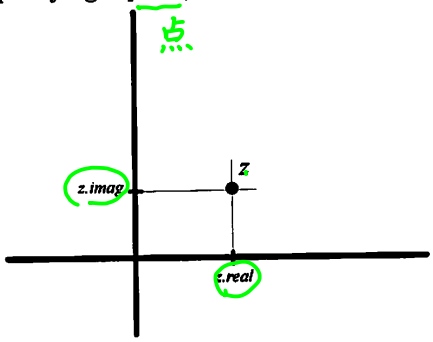
- $\mathbb{R}$, the field of real numbers,
- $\mathbb{C}$, the field of complex numbers, and
- $GF(2)$, a field that consists of 0 and 1.

In object-oriented programming, one can use the name of a class to refer to the set of instances of that class, e.g. we refer to instances of the class `Rectangle` as Rectangles. In mathematics, one uses the name of the field, e.g. $\mathbb{R}$ or $GF(2)$, to refer also to the set of values.

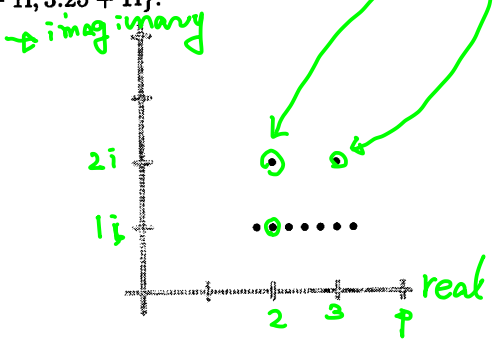
[1] For the reader who knows about object-oriented programming, a field is analogous to a class satisfying an interface that requires it to possess methods for the arithmetic operators.

1.4 Playing with $\mathbb{C}$

Because each complex number `z` consists of two ordinary numbers, `z.real` and `z.imag`, it is traditional to think of `z` as specifying a *point*, a location, in the plane (the *complex plane*).



To build intuition, let us use a set S of complex numbers to represent a black-and-white image. For each location in the complex plane where we want a dot, we include the corresponding complex number in S. The following figure shows $S = \{2+2\mathbf{i}, 3+2\mathbf{i}, 1.75+1\mathbf{i}, 2+1\mathbf{i}, 2.25+1\mathbf{i}, 2.5+1\mathbf{i}, 2.75+1\mathbf{i}, 3+1\mathbf{i}, 3.25+1\mathbf{i}\}$:



Task 1.4.1: First, assign to the variable `S` a list or set consisting of the complex numbers listed above.

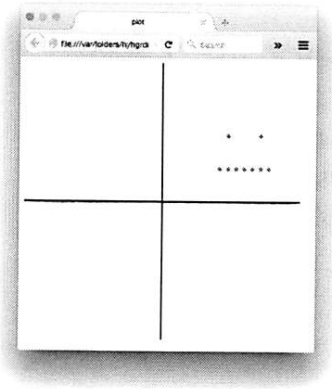
We have provided a module `plotting` for showing points in the complex plane. The module defines a procedure `plot`. Import this class from the module as follows:

```
>>> from plotting import plot
```

Next, plot the points in S as follows:

```
>>>> plot(S, 4)
```

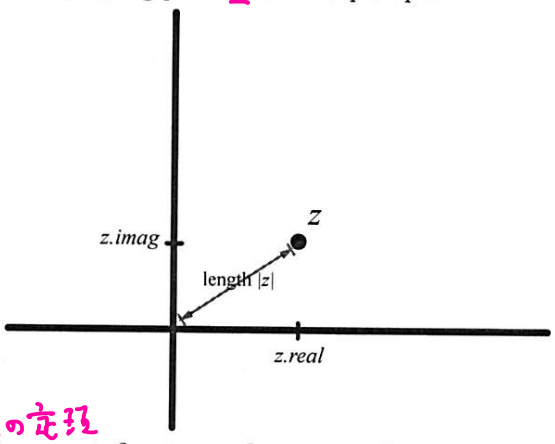
Python should open a browser window displaying the points of `S` in the complex plane. The first argument to `plot` is a collection of complex numbers (or 2-tuples). The second argument sets the scale of the plot; in this case, the window can show complex numbers whose real and imaginary parts have absolute value less than 4. The scale argument is optional and defaults to 1, and there is another optional argument that sets the size of the dots.



絶対値

1.4.1 The absolute value of a complex number

The *absolute value* of a complex number z, written $|z|$ (and, in Python, `abs(z)`) is the distance from the origin to the corresponding point in the complex plane.



ピタゴラスの定理

By the Pythagorean Theorem, $|z|^2 = (z.\text{real})^2 + (z.\text{imag})^2$.

```
>>> abs(3+4j)
5.0
>>> abs(1+1j)
1.4142135623730951
```

$3^2+4^2=5^2$

複素

共役

Definition 1.4.2: The *conjugate* of a complex number z, written $\bar{z}$, is defined as $z.\text{real} - z.\text{imag}$.

In Python, we write `z.conjugate()`.

```
>>> (3+4j).conjugate()
(3-4j)
```

z bar

Using the fact that $\mathbf{i}^2 = -1$, we can get a formula for $|z|^2$ in terms of z and $\bar{z}$:

$$|z|^2 = z \cdot \bar{z} \tag{1.1}$$

証明

Proof

$$
\begin{aligned}
z \cdot \bar{z} &= (z.\text{real} + z.\text{imag}\,\mathbf{i}) \cdot (z.\text{real} - z.\text{imag}\,\mathbf{i}) \\
&= z.\text{real} \cdot z.\text{real} - z.\text{real} \cdot z.\text{imag}\,\mathbf{i} + z.\text{imag}\,\mathbf{i} \cdot z.\text{real} - z.\text{imag}\,\mathbf{i} \cdot z.\text{imag}\,\mathbf{i} \\
&= z.\text{real}^2 - z.\text{imag}\,\mathbf{i} \cdot z.\text{imag}\,\mathbf{i} \\
&= z.\text{real}^2 - z.\text{imag} \cdot z.\text{imag}\,\mathbf{i}^2 \\
&= z.\text{real}^2 + z.\text{imag} \cdot z.\text{imag}
\end{aligned}
$$

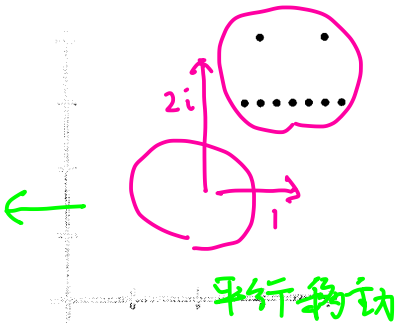
where the last equality uses the fact that $\mathbf{i}^2 = -1$. □

1.4.2 Adding complex numbers

Suppose we add a complex number, say $1 + 2\mathbf{i}$, to each complex number z in S. That is, we derive a new set by applying the following function to each element of S:

$$f(z) = 1 + 2\mathbf{i} + z$$

This function increases each real coordinate (the x coordinate) by 1 and increases each imaginary coordinate (the y coordinate) by 2. The effect is to shift the picture one unit to the right and two units up:



This transformation of the numbers in S is called a *translation*. A translation has the form

$$f(z) = z_0 + z \tag{1.2}$$

where z_0 is a complex number. Translations can take the picture anywhere in the complex plane. For example, adding a number z_0 whose real coordinate is negative would have the effect of translating the picture to the left.

Task 1.4.3: Create a new plot using a comprehension to provide a set of points derived from `S` by adding $1 + 2\mathbf{i}$ to each:

```
>>> plot({1+2j+z for z in S}, 4)
```

Quiz 1.4.4: The "left eye" of the set S of complex numbers is located at $2 + 2\mathbf{i}$. For what value of z_0 does the translation $f(z) = z_0 + z$ move the left eye to the origin?

Answer

$z_0 = -2 - 2\mathbf{i}$. That is, the translation is $f(z) = -2 - 2\mathbf{i} + z$.

Problem 1.4.5: Show that, for any two distinct points z_1 and z_2,

- there is a translation that maps z_1 to z_2,
- there is a translation that maps z_2 to z_1, and

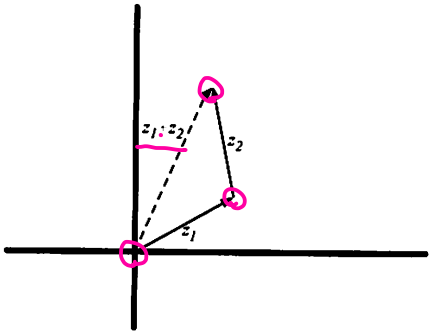
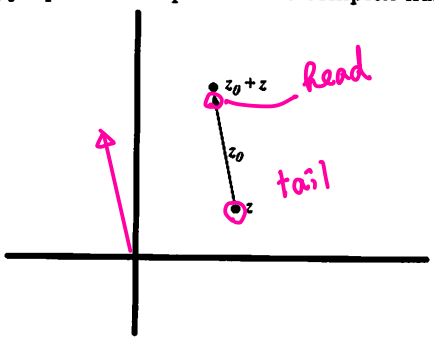


Figure 1.1: This figure illustrates the geometric interpretation of complex-number addition.

- there is *no* translation that both maps z_1 to z_2 and z_2 to z_1.

Complex numbers as arrows It is helpful to visualize a translation $f(z)$ by an arrow. The tail of the arrow is located at any point z in the complex plane; the head of the arrow is then located at the point $f(z)$, the translation of z. Of course, this representation is not unique.

Since a translation has the form $f(z) = z_0 + z$, we represent the translation by the complex number z_0. It is therefore appropriate to represent the complex number z_0 by an arrow.



Again, the representation is not unique. For example, the vector $z_0 = 5 - 2\mathbf{i}$ can be represented by an arrow whose tail is at $0 + 0\mathbf{i}$ and whose head is at $5 - 2\mathbf{i}$, or one whose tail is at $1 + 1\mathbf{i}$ and whose head is at $6 - 1\mathbf{i}$, or....

Problem 1.4.6: Draw a diagram representing the complex number $z_0 = -3 + 3\mathbf{i}$ using two arrows with their tails located at different points.

Composing translations, adding arrows Let $f_1(z) = z_1 + z$ and $f_2(z) = z_2 + z$ be two translations. Then their composition is also a translation:

$$\begin{aligned}(f_2 \circ f_1)(z) &= f_2(f_1(z)) \\ &= f_2(z_1 + z) \\ &= z_2 + z_1 + z\end{aligned}$$

and is defined by $z \mapsto (z_2 + z_1) + z$. The idea that two translations can be collapsed into one is illustrated by Figure 1.1, in which each translation is represented by an arrow.

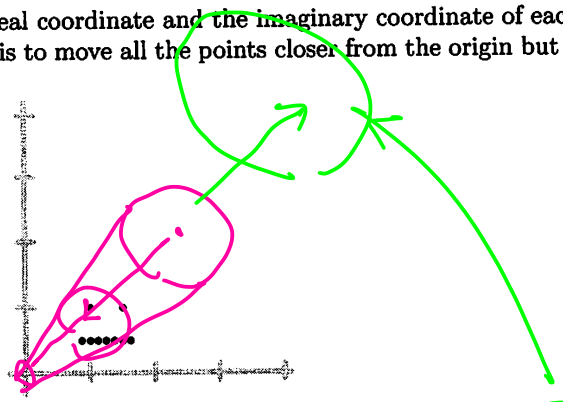
The translation arrow labeled by z_1 takes a point (in this case, the origin) to another point, which in turn is mapped by z_2 to a third point. The arrow mapping the origin to the third point is the composition of the two other translations, so, by the reasoning above, is $z_1 + z_2$.

1.4.3 Multiplying complex numbers by a positive real number

Now suppose we halve each complex number in S:

$$g(z) = \frac{1}{2}z$$

This operation simply halves the real coordinate and the imaginary coordinate of each complex number. The effect on the picture is to move all the points closer from the origin but also closer to each other:

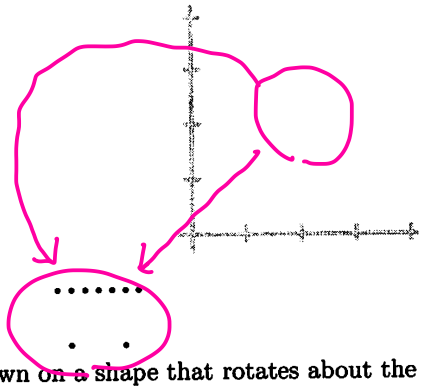


This operation is called *scaling*. The scale of the picture has changed. Similarly, doubling each complex number moves the points farther from the origin and from each other.

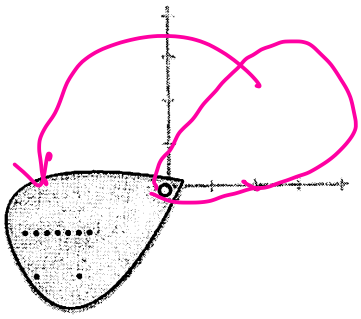
Task 1.4.7: Create a new plot titled "My scaled points" using a comprehension as in Task 1.4.3. The points in the new plot should be halves of the points in S.

1.4.4 Multiplying complex numbers by a negative number: rotation by 180 degrees

Here is the result of multiplying each complex number by -1:



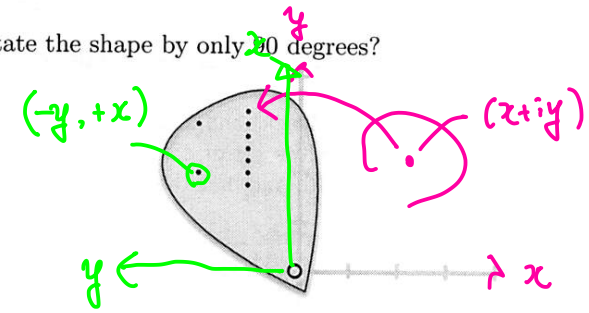
Think of the points as drawn on a shape that rotates about the origin; this picture is the result of rotating the shape by 180 degrees.



1.4.5 Multiplying by i: rotation by 90 degrees

"The number you have dialed is imaginary. Please rotate your phone by ninety degrees and try again."

How can we rotate the shape by only 90 degrees?



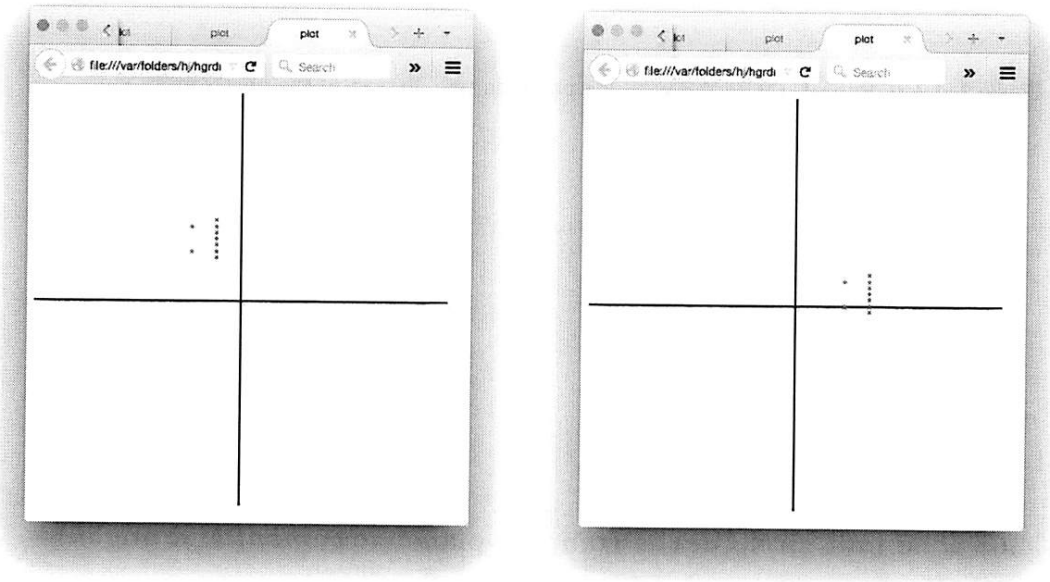
$i \cdot (x+iy)$
$= ix + i^2 y$
$= ix - y$
$= -y + ix$
$(-y, x)$

For this effect, a point located at (x, y) must be moved to $(-y, x)$. The complex number located at (x, y) is $x + \mathbf{i}y$. Now is our chance to use the fact that $\mathbf{i}^2 = -1$. We use the function

$$h(z) = \mathbf{i} \cdot z$$

Multiplying $x + \mathbf{i}y$ by $\mathbf{i}$ yields $\mathbf{i}x + \mathbf{i}^2 y$, which is $\mathbf{i}x - y$, which is the complex number represented by the point $(-y, x)$.

Task 1.4.8: Create a new plot in which the points of S are rotated by 90 degrees and scaled by 1/2. Use a comprehension in which the points of S are multiplied by a single complex number.



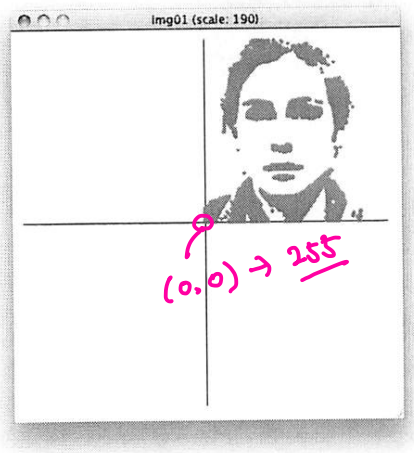
Task 1.4.9: Using a comprehension, create a new plot in which the points of S are rotated by 90 degrees, scaled by 1/2, and then shifted down by one unit and to the right two units. Use a comprehension in which the points of S are multiplied by one complex number and added to another.

22/12/8.

Task 1.4.10: We have provided a module `image` with a procedure `file2image(filename)` that reads in an image stored in a file in the `.png` format. Import this procedure and invoke it, providing as argument the name of a file containing an image in this format, assigning the returned value to variable `data`. An example grayscale image, `img01.png`, is available for download.

The value of `data` is a list of lists, and `data[y][x]` is the intensity of pixel `(x,y)`. Pixel `(0,0)` is at the bottom-left of the image, and pixel `(width-1, height-1)` is at the top-right. The intensity of a pixel is a number between 0 (black) and 255 (white).

Use a comprehension to assign to a list `pts` the set of complex numbers $x + y\mathbf{i}$ such that the image intensity of pixel (x, y) is less than 120, and plot the list `pts`.



Task 1.4.11: Write a Python procedure `f(z)` that takes as argument a complex number z so that when $f(z)$ is applied to each of the complex numbers in `S`, the set of resulting numbers is centered at the origin. Write a comprehension in terms of `S` and `f` whose value is the set of translated points, and plot the value.

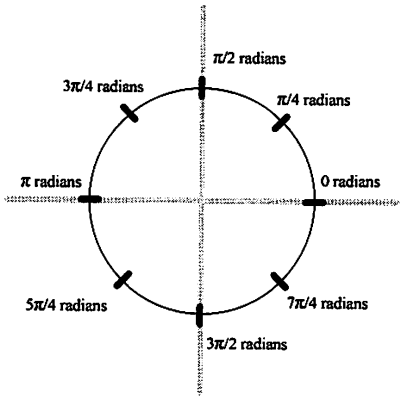
Task 1.4.12: Repeat Task 1.4.8 with the points in `pts` instead of the points in `S`.

1.4.6 The unit circle in the complex plane: *argument* and angle

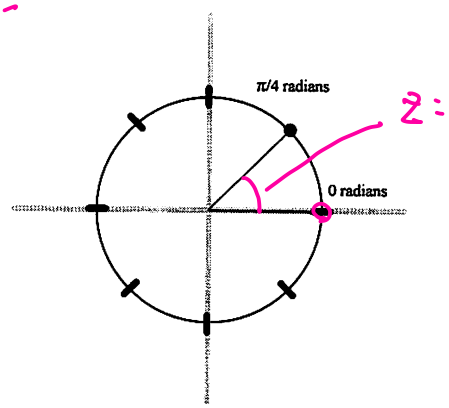
We shall see that it is not a coincidence that rotation by 180 or 90 degrees can be represented by complex multiplication: any rotation can be so represented. However, it is convenient to use *radians* instead of degrees to measure the angle of rotation.

The *argument* of a complex number on the unit circle

Consider the unit circle—the circle of radius one, centered at the origin of the complex plane.



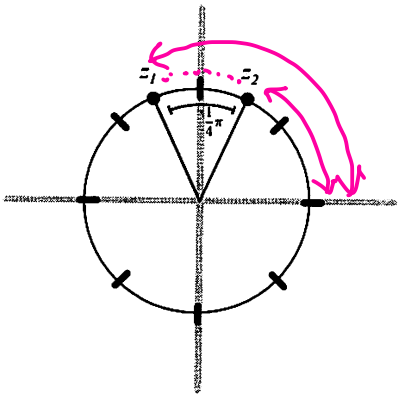
A point z on the circle is represented by the distance an ant would have to travel counterclockwise along the circle to get to z if the ant started at $1 + 0\mathbf{i}$, the rightmost point of the circle. We call this number the *argument* of z.



Example 1.4.13: Since the circumference of the circle is 2π, the point halfway around the circle has an argument of π, and the point one-eighth of the way around has an argument of $\pi/4$.

The angle formed by two complex numbers on the unit circle

We have seen how to label points on the unit circle by distances. We can similarly assign a number to the *angle* formed by the line segments from the origin to two points z_1 and z_2 on the circle. The angle, measured in radians, is the distance along the circle traversed by an ant walking counterclockwise from z_2 to z_1.



Example 1.4.14: Let z_1 be the point on the circle that has argument $\frac{5}{16}\pi$, and let z_2 be the point on the circle that has argument $\frac{3}{16}\pi$. An ant starting at z_2 and traveling to z_1 would travel a distance of $\frac{1}{8}\pi$ counterclockwise along the circle, so $\frac{1}{8}\pi$ is the angle between the origin-to-z_1 line segment and the origin-to-z_2 line segment.

注意

Remark 1.4.15: The argument of z is the angle formed by z with $1 + 0\mathbf{i}$.

1.4.7 Euler's formula

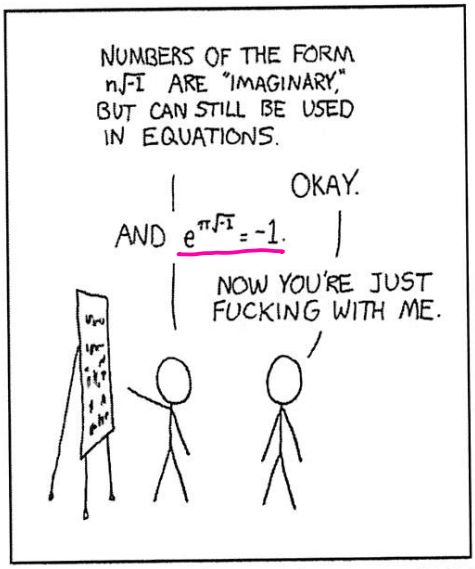
> He calculated just as men breathe, as eagles sustain themselves in the air.
> *Said of Leonhard Euler*

We turn to a formula due to Leonhard Euler, a remarkable mathematician who contributed to the foundation for many subfields of mathematics: number theory and algebra, complex analysis, calculus, differential geometry, fluid mechanics, topology, graph theory, and even music theory and cartography. Euler's formula states that, for any real number θ, $e^{\mathbf{i}\cdot\theta}$ is the point z on the unit circle with argument θ. Here e is the famous transcendental number 2.718281828....

超越数

Example 1.4.16: The point $-1 + 0\mathbf{i}$ has argument π. Plugging π into Euler's formula yields the surprising equation $e^{\mathbf{i}\pi} + 1 = 0$.

Field: 体
Group: 群.
(manim)



e to the π times $\mathbf{i}$ (`http://xkcd.com/179/`)

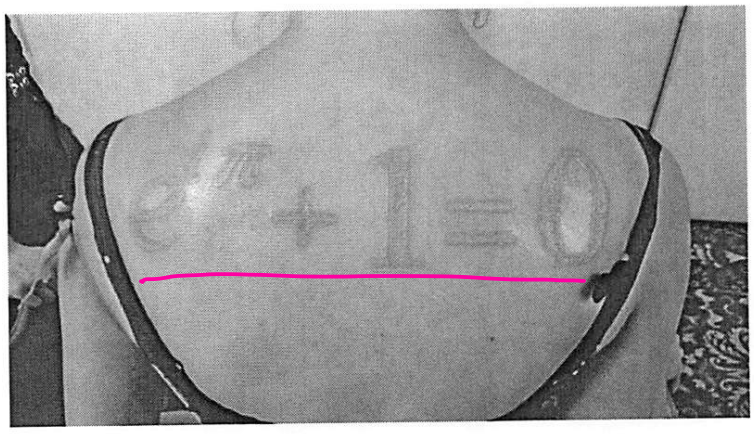
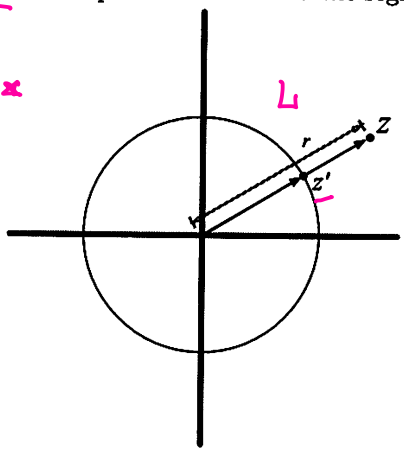


photo taken at 3PiCon in Springfield, MA, by Cory Doctorow

Task 1.4.17: From the module `math`, import the definitions `e` and `pi`. Let `n` be the integer 20. Let `w` be the complex number $e^{2\pi i/n}$. Write a comprehension yielding the list consisting of $w^0, w^1, w^2, \ldots, w^{n-1}$. Plot these complex numbers.

1.4.8 Polar representation for complex numbers

Euler's formula gives us a convenient representation for complex numbers that lie on the unit circle. Now consider any complex number z. Let L be the line segment in the complex plane from the origin to z, and let z' be the point at which this line segment intersects the unit circle.



Let r be the length of the line segment to z. Viewing z' as the result of scaling down z, we have

$$z' = \frac{1}{r}z$$

Let θ be the argument of z'. Euler's formula tells us that $z' = e^{\theta i}$. We therefore obtain

$$z = re^{\theta i}$$

The astute student might recognize that r and θ are the polar coordinates of z. In the context of complex numbers, we define the *argument* of z to be θ, and we define the *absolute value* of z (written $|z|$) to be r.

1.4.9 The First Law of Exponentiation

When powers multiply, their exponents add:

$$e^u e^v = e^{u+v}$$

We can use this rule to help us understand how to rotate a complex number z. We can write

$$z = re^{\theta i}$$

where $r = |z|$ and $\theta = \arg z$.

1.4.10 Rotation by τ radians

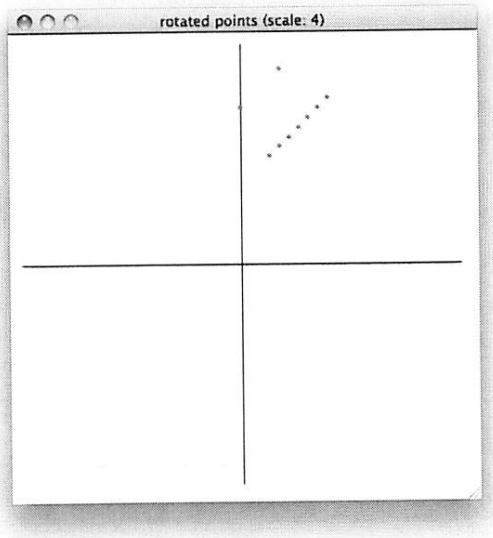
Let τ be a number of radians. The rotation of z by τ should have the same absolute value as z but its argument should be τ more than that of z, i.e. it should be $re^{(\theta+\tau)i}$. How do we obtain this number from z?

$$\begin{aligned} re^{(\theta+\tau)i} &= re^{\theta i}e^{\tau i} \\ &= ze^{\tau i} \end{aligned}$$

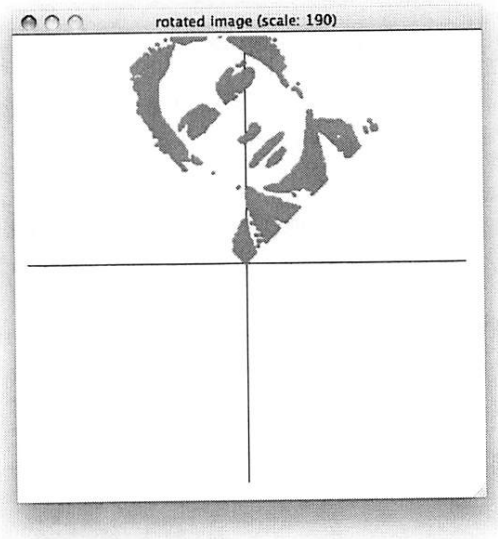
Thus the function that rotates by τ is simply

$$f(z) = ze^{\tau i}$$

Task 1.4.18: Recall from Task 1.4.1 the set S of complex numbers. Write a comprehension whose value is the set consisting of rotations by $\pi/4$ of the elements of S. Plot the value of this comprehension.

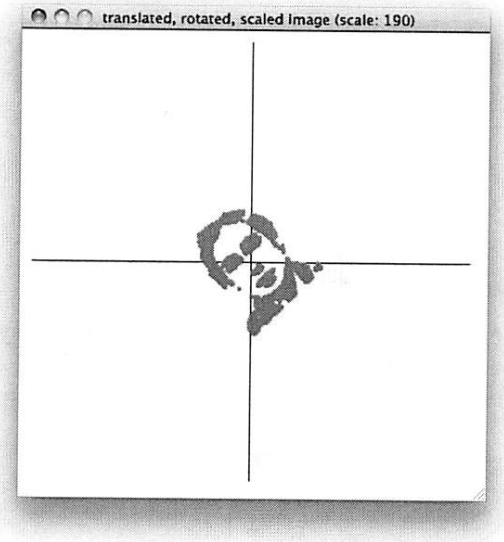


Task 1.4.19: Similarly, recall from Task 1.4.10 the list `pts` of points derived from an image. Plot the rotation by $\pi/4$ of the complex numbers comprising `pts`.



1.4.11 Combining operations

Task 1.4.20: Write a comprehension that transforms the set `pts` by translating it so the image is centered, then rotating it by $\pi/4$, then scaling it by half. Plot the result.



Because the complex numbers form a field, familiar algebraic rules can be used. For example, $a \cdot (b \cdot z) = (a \cdot b) \cdot z$. Using this rule, two scaling operations can be combined into one; scaling by 2 and then by 3 is equivalent to scaling by 6.

Similarly, since rotation is carried out by multiplication, two rotations can be combined into one; rotating by $\frac{\pi}{4}$ (multiplying by $e^{\frac{\pi}{4}\mathbf{i}}$) and and then rotating by $\frac{\pi}{3}$ (multiplying by $e^{\frac{\pi}{3}\mathbf{i}}$) is equivalent to multiplying by $e^{\frac{\pi}{4}\mathbf{i}} \cdot e^{\frac{\pi}{3}\mathbf{i}}$, which is equal to $e^{\frac{\pi}{4}\mathbf{i}+\frac{\pi}{3}\mathbf{i}}$, i.e. rotating by $\frac{\pi}{4} + \frac{\pi}{3}$.

Since scaling and rotation both consist in multiplication, a rotation and a scaling can be combined: rotating by $\frac{\pi}{4}$ (multiplying by $e^{\frac{\pi}{4}\mathbf{i}}$) and then scaling by $\frac{1}{2}$ is equivalent to multiplying by $\frac{1}{2}e^{\frac{\pi}{4}\mathbf{i}}$.

1.4.12 Beyond two dimensions

The complex numbers are so convenient for transforming images—and, more generally, sets of points in the plane—one might ask whether there is a similar approach to operating on points in three dimensions. We discuss this in the next chapter.

1.5 Playing with $GF(2)$

$GF(2)$ is short for *Galois Field 2*. Galois was a mathematician, born in 1811, who while in his teens essentially founded the field of abstract algebra. He died in a duel at age twenty.

The field $GF(2)$ is very easy to describe. It has two elements, 0 and 1. Arithmetic over $GF(2)$ can be summarized in two small tables:

$\cdot\times$	0	1
0	0	0
1	0	1

$\cdot+$	0	1
0	0	1
1	1	0

Addition is modulo 2. It is equivalent to exclusive-or. In particular, $1 + 1 = 0$.

Subtraction is identical to addition. The negative of 1 is again 1, and the negative of 0 is again 0.

Multiplication in $GF(2)$ is just like ordinary multiplication of 0 and 1: multiplication by 0 yields 0, and 1 times 1 is 1. You can divide by 1 (as usual, you get the number you started with) but dividing by zero is illegal (as usual).

We provide a module, `GF2`, with a very simple implementation of $GF(2)$. It defines a value, `one`, that acts as the element 1 of $GF(2)$. Ordinary zero plays the role of the element 0 of $GF(2)$. (For visual consistency, the module defines `zero` to be the value 0.)

```
>>> from GF2 import one
>>> one*one
one
```

```
>>> one*0
0
>>> one + 0
one
>>> one+one
0
>>> -one
one
```

1.5.1 Perfect secrecy revisited

In Chapter 0, we described a cryptosystem that achieves perfect secrecy (in transmitting a single bit). Alice and Bob randomly choose the key k uniformly from $\{\clubsuit, \heartsuit\}$. Subsequently, Alice uses the following encryption function to transform the plaintext bit p to a cyphertext bit c:

p	k	c
0	$\clubsuit$	0
0	$\heartsuit$	1
1	$\clubsuit$	1
1	$\heartsuit$	0

The encryption method is just $GF(2)$ addition in disguise! When we replace $\clubsuit$ with 0 and $\heartsuit$ with 1, the encryption table becomes the addition table for $GF(2)$:

p	k	c
0	0	0
0	1	1
1	0	1
1	1	0

For each plaintext $p \in GF(2)$, the function $k \mapsto k + p$ (mapping $GF(2)$ to $GF(2)$) is invertible (hence one-to-one and onto). Therefore, when the key k is chosen uniformly at random, the cyphertext is also distributed uniformly. This shows that the scheme achieves perfect secrecy.

Using integers instead of $GF(2)$

Why couldn't Alice and Bob use, say, ordinary integers instead of $GF(2)$? After all, for each $x \in \mathbb{Z}$, the function $y \mapsto x + y$ mapping $\mathbb{Z}$ to $\mathbb{Z}$ is also invertible. The reason this cannot work as a cryptosystem is that there is no uniform distribution over $\mathbb{Z}$, so the first step—choosing a key—is impossible.

Encrypting long messages

How, then, are we to encrypt a long message? Students of computer science know that a long message can be represented by a long string of bits. Suppose the message to be encrypted will consist of n bits. Alice and Bob should select an equally long sequence of key bits $k_1 \ldots k_n$. Now, once Alice has selected the plaintext $p_1 \ldots p_n$, she obtains the cyphertext $c_1 \ldots c_n$ one bit at a time:

$$\begin{aligned} c_1 &= k_1 + p_1 \\ c_2 &= k_2 + p_2 \\ &\vdots \\ c_n &= k_n + p_n \end{aligned}$$

We argue informally that this system has perfect secrecy. The earlier argument shows that each bit c_i of cyphertext tells Eve nothing about the corresponding bit p_i of plaintext; certainly the bit c_i tells Eve nothing about any of the other bits of plaintext. From this we infer that the system has perfect secrecy.

Our description of the multi-bit system is a bit cumbersome, and the argument for perfect secrecy is rather sketchy. In Chapter 2, we show that using vectors over $GF(2)$ simplify the presentation.

The one-time pad

The cryptosystem we have described is called the *one-time pad*. As suggested by the name, it is crucial that each bit of key be used only once, i.e. that each bit of plaintext be encrypted with its bit of key. This can be a burden for two parties that are separated for long periods of time because the two parties must agree before separating on many bits of key.

Starting in 1930, the Soviet Union used the one-time pad for communication. During World War II, however, they ran out of bits of key and began to re-use some of the bits. The US and Great Britain happen to discover this; they exploited it (in a top-secret project codenamed VENONA) to partially decrypt some 1% of the encrypted messages, revealing, for example, the involvement of Julius Rosenberg and Alger Hiss in espionage.

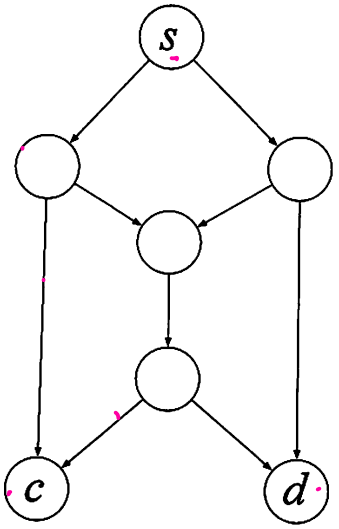
Problem 1.5.1: An 11-symbol message has been encrypted as follows. Each symbol is represented by a number between 0 and 26 ($A \mapsto 0, B \mapsto 1, \ldots, Z \mapsto 25$, *space* $\mapsto 26$). Each number is represented by a five-bit binary sequence ($0 \mapsto 00000, 1 \mapsto 00001, \ldots, 26 \mapsto 11010$). Finally, the resulting sequence of 55 bits is encrypted using a flawed version of the one-time pad: the key is not 55 random bits but 11 copies of the same sequence of 5 random bits. The cyphertext is

10101 00100 10101 01011 11001 00011 01011 10101 00100 11001 11010

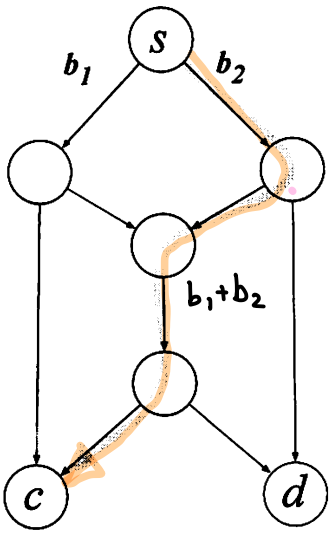
Try to find the plaintext.

1.5.2 Network coding

Consider the problem of streaming video through a network. Here is a simple example network:

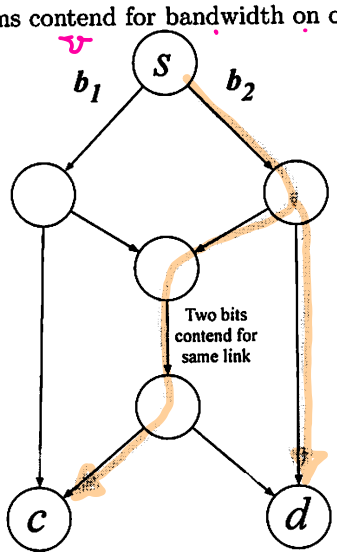


The node at the top labeled s needs to stream a video to each of the two customer nodes, labeled c and d, at the bottom. Each link in the network has a capacity of 1 megabit per second. The video stream, however, requires 2 megabits per second. If there were only one customer, this would be no problem; as shown below, the network can handle two simultaneous 1-megabit-per-second streams from s to c:

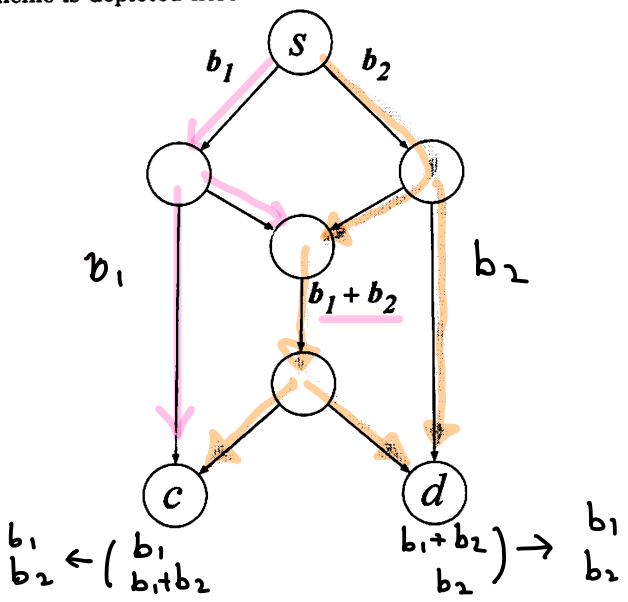


A million times a second, one bit b_1 is sent along one path and another bit b_2 is sent along another path. Thus the total rate of bits delivered to the customer is 2 megabits per second.

However, as shown below, we can't use the same scheme to deliver two bitstreams to each of two customers because the streams contend for bandwidth on one of the network links.



$GF(2)$ to the rescue! We can use the fact that network nodes can do a tiny bit (!) of computation. The scheme is depicted here:



At the centermost node, the bits b_1 and b_2 arrive and are combined by $GF(2)$ addition to obtain a single bit. That single bit is transmitted as shown to the two customers c and d. Customer c receives bit b_1 and the sum $b_1 + b_2$, so can also compute the bit b_2. Customer d receives bit b_2 and the sum $b_1 + b_2$, so can also compute the bit b_1.

We have shown that a network that appears to support streaming only one megabit per second to a pair of customers actually supports streaming two megabits per second. This approach to routing can of course be generalized to larger networks and more customers; the idea is called *network coding*.

1.6 Review questions

- Name three fields.
- What is the conjugate of a complex number? What does it have to do with the absolute value of a complex number?
- How does complex-number addition work?
- How does complex-number multiplication work?
- How can translation be defined in terms of complex numbers?
- How can scaling be defined in terms of complex numbers?
- How can rotation by 180 degrees be defined in terms of complex numbers?
- How can rotation by 90 degrees be defined in terms of complex numbers?
- How does addition of $GF(2)$ values work?
- How does multiplication of $GF(2)$ values work?

1.7 Problems

Python comprehension problems

Write each of the following three procedures using a comprehension:

Problem 1.7.1: `my_filter(L, num)`
input: list of numbers and a positive integer.
output: list of numbers not containing a multiple of `num`.
example: given `list = [1,2,4,5,7]` and `num = 2`, return `[1,5,7]`.

Problem 1.7.2: `my_lists(L)`
input: list `L` of non-negative integers.
output: a list of lists: for every element x in `L` create a list containing $1, 2, \ldots, x$.
example: given `[1,2,4]` return `[[1],[1,2],[1,2,3,4]]`. *example:* given `[0]` return `[[]]`.

Problem 1.7.3: `my_function_composition(f,g)`
input: two functions f and g, represented as dictionaries, such that $g \circ f$ exists.
output: dictionary that represents the function $g \circ f$.
example: given f = {0:'a', 1:'b'} and g = {'a':'apple', 'b':'banana'}, return {0:'apple', 1:'banana'}.

Python loop problems

For procedures in the following five problems, use the following format:

```
def <ProcedureName>(L):
    current = ...
    for x in L:
        current = ...
    return current
```

The value your procedure initially assigns to `current` turns out to be the return value in the case when the input list `L` is empty. This provides us insight into how the answer should be defined in that case. Note: You are not allowed to use Python built-in procedures `sum(·)` and `min(·)`.

Problem 1.7.4: `mySum(L)`
Input: list of numbers
Output: sum of numbers in the list

Problem 1.7.5: `myProduct(L)`
input: list of numbers
output: product of numbers in the list

Problem 1.7.6: `myMin(L)`
input: list of numbers
output: minimum number in the list

Problem 1.7.7: `myConcat(L)`
input: list of strings
output: concatenation of all the strings in `L`

Problem 1.7.8: `myUnion(L)`
input: list of sets
output: the union of all sets in `L`.

In each of the above problems, the value of `current` is combined with an element of `myList` using some operation $\diamond$. In order that the procedure return the correct result, `current` should be initialized with the *identity element* for the operation $\diamond$, i.e. the value i such that $i \diamond x = x$ for any value x.

It is a consequence of the structure of the procedure that, when the input list is empty, the output value is the initial value of `current` (since in this case the body of the loop is never executed). It is convenient to define this to be the correct output!

Problem 1.7.9: Keeping in mind the comments above, what should be the value of each of the following?

1. The sum of the numbers in an empty set.
2. The product of the numbers in an empty set.
3. The minimum of the numbers in an empty set.

4. The concatenation of an empty list of strings.
5. The union of an empty list of sets.

What goes wrong when we try to apply this reasoning to define the intersection of an empty list of sets?

Complex addition practice

Problem 1.7.10: Each of the following problems asks for the sum of two complex numbers. For each, write the solution and illustrate it with a diagram like that of Figure 1.1. The arrows you draw should (roughly) correspond to the vectors being added.

a. $(3 + 1\mathbf{i}) + (2 + 2\mathbf{i})$

b. $(-1 + 2\mathbf{i}) + (1 - 1\mathbf{i})$

c. $(2 + 0\mathbf{i}) + (-3 + .001\mathbf{i})$

d. $4(0 + 2\mathbf{i}) + (.001 + 1\mathbf{i})$

Multiplication of exponentials

Problem 1.7.11: Use the First Rule of Exponentiation (Section 1.4.9) to express the product of two exponentials as a single exponential. For example, $e^{(\pi/4)\mathbf{i}}e^{(\pi/4)\mathbf{i}} = e^{(\pi/2)\mathbf{i}}$.

a. $e^{1\mathbf{i}}e^{2\mathbf{i}}$

b. $e^{(\pi/4)\mathbf{i}}e^{(2\pi/3)\mathbf{i}}$

c. $e^{-(\pi/4)\mathbf{i}}e^{(2\pi/3)\mathbf{i}}$

Combining operations on complex numbers

Problem 1.7.12: Write a procedure `transform(a,b, L)` with the following spec:

- *input:* complex numbers a and b, and a list L of complex numbers
- *output:* the list of complex numbers obtained by applying $f(z) = az + b$ to each complex number in L

Next, for each of the following problems, explain which value to choose for a and b in order to achieve the specified transformation. If there is no way to achieve the transformation, explain.

a. Translate z one unit up and one unit to the right, then rotate ninety degrees clockwise, then scale by two.

b. Scale the real part by two and the imaginary part by three, then rotate by forty-five degrees counterclockwise, and then translate down two units and left three units.

$GF(2)$ arithmetic

Problem 1.7.13: For each of the following problems, calculate the answer over $GF(2)$.

a. $1 + 1 + 1 + 0$

b. $1 \cdot 1 + 0 \cdot 1 + 0 \cdot 0 + 1 \cdot 1$

c. $(1 + 1 + 1) \cdot (1 + 1 + 1 + 1)$

Network coding

Problem 1.7.14: Copy the example network used in Section 1.5.2. Suppose the bits that need to be transmitted in a given moment are $b_1 = 1$ and $b_2 = 1$. Label each link of the network with the bit transmitted across it according to the network-coding scheme. Show how the customer nodes c and d can recover b_1 and b_2.