Chapter 1

The Field

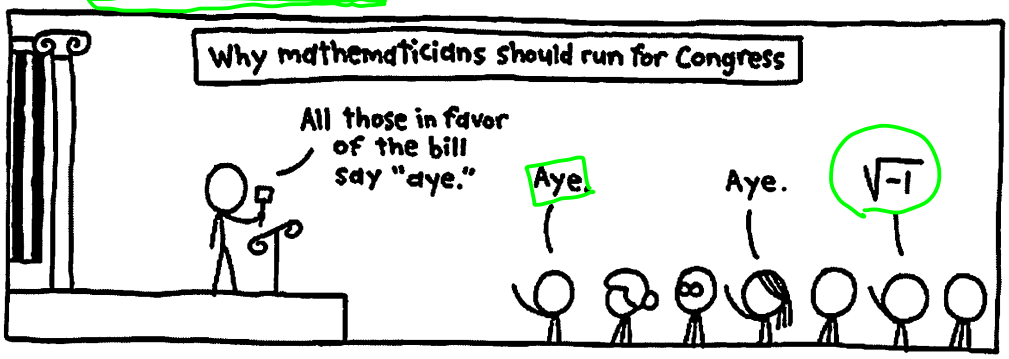
...the different branches of
Arithmetic—Ambition, Distraction,
Uglification, and Derision.

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

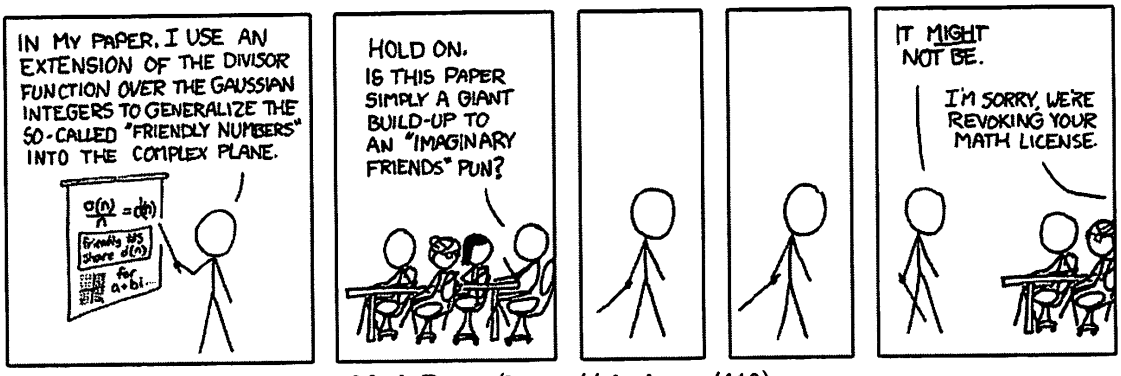
$$9\mathbf{i}^2 = -9$$

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.

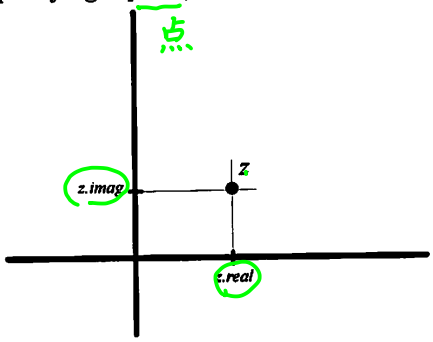
- $\mathbb{R}$, the field of real numbers,
- $\mathbb{C}$, the field of complex numbers, and
- $GF(2)$, a field that consists of 0 and 1.

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.

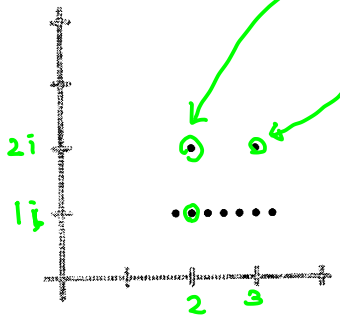
[1] For the reader who knows about object-oriented programming, a field is analogous to a class satisfying an interface that requires it to possess methods for the arithmetic operators.

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.