

# Fluxion Language Specification 0.1

## Part I: Syntax & Semantics

Ege Özkan

September 24, 2020



# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	A Defence of Fluxion . . . . .	6
1.1.1	Fluxion is not a Programming Language . . . . .	7
1.1.2	the Name Fluxion . . . . .	7
1.2	Contents of the Specification . . . . .	7
<b>2</b>	<b>Grammar</b>	<b>9</b>
2.1	Notation . . . . .	10
2.2	Common Lexical Units . . . . .	10
2.3	Literals . . . . .	10
2.3.1	Boolean Literals . . . . .	10
2.3.2	Numerical Literals . . . . .	10
2.3.3	Sets and Sequences . . . . .	11
2.3.4	Matrices and Vectors . . . . .	11
2.3.5	General Literal Grammar . . . . .	11
2.4	Operators . . . . .	11
2.4.1	Binary Operators . . . . .	11
2.4.2	Unary Operators . . . . .	12
2.5	Variables and Expressions . . . . .	12
2.6	Statements . . . . .	12
2.7	General Program Grammar . . . . .	13
<b>3</b>	<b>Fluxion's Type System</b>	<b>15</b>
3.1	An Overview of the Fluxion's Type System . . . . .	16
3.1.1	Unbound Variable . . . . .	16
3.1.2	Associated Grammar . . . . .	16
3.2	Booleans . . . . .	17
3.3	Numbers . . . . .	17
3.3.1	Integers . . . . .	17
3.3.2	Decimal . . . . .	17
3.3.3	Imaginary numbers . . . . .	17
3.3.4	Irrational Numbers . . . . .	18
3.4	Matrices . . . . .	18

3.4.1	Vectors . . . . .	18
3.5	Sets . . . . .	18
3.5.1	Sequences . . . . .	19
3.5.2	Predefined Sets and Sequences . . . . .	19
<b>4</b>	<b>Operations</b>	<b>21</b>
4.1	Operator Semantics . . . . .	22
4.2	Homogeneous Operations . . . . .	23
4.2.1	Numerical Operations . . . . .	23

# Chapter 1

## Introduction

This chapter includes the design philosophy of and concepts behind the Fluxion Language, and a defence of the Fluxion Language, that is, why it is needed.

## 1.1 A Defence of Fluxion

First thing one has to understand when it comes to language is its usefulness. As the readers may ask, *why* should Fluxion exist? Don't we already have many scientific programming languages like Python or Julia?

The answer to these questions can be given twofold: First, languages live and breath alongside with their ecosystem, even though their existence might be meaningful without a certain use case, and some even develop use cases they are originally not created for, in many cases, languages with a clear goal are easier to design.

Domain Specific Languages, are languages created to answer problems arising from a domain. Fluxion is such a language, its chief primary concern is to create a language, that can adequately represent mathematical constructs and expressions from many differing fields, to simplify, solve or to manipulate these constructs for the benefit of the users, who may wish to use Fluxion in their systems for a variety of purposes. These may range from pedagogical concerns, such as to teach pupils about certain mathematical concepts to proving actual mathematical problems.

When it comes to languages like Python, Julia or even Matlab, though they are useful, they are far too complex to be simple enough for users to actually engage with the symbolic mathematical underlayings of formulas, theorems and proofs. Fluxion aims to be simpler, much closer to mathematical notation, while still retaining *some* but not all of their power. Fluxion does not aim to be a general purpose programming language. It does not even aim to be *a* programming language. Fluxion is a Symbolic Computation Language, and it aims to excel at that one field.

Second, Fluxion does not aim to replace these languages, though some users will benefit from it, to such a degree that in some use cases, they may prefer Fluxion, Fluxion will be written in such a way that it can be embedded and interloop with many of these languages, which will allow users to concentrate on problems and expressions like never before.

When designing Fluxion, first aim was to create a language that is easy to understand, extremely close to mathematical notation even if it went against previously established conventions of Computer Science, or if it adhered to conventions lesser known, this allows people literate in Mathematics to easily pick up Fluxion.

The second aim was to make sure that mistakes were limited. For instance, unlike many other languages, star imports does not exist in Fluxion

to avoid namespace cluttering, in the reverse side of the import, module members are by default private, and must be exported explicitly using the `export` keyword.

Third aim of the Fluxion's *design* is for it to be extendable. Using the `introduce` keyword, users can introduce Fluxion new *contexts*, that can be about different domains of science using mathematics. A Physics domain may introduce Physics related constants and functions, a Mechanics domain may introduce functions and constants related to the mechanical engineering. This allows Fluxion to be tailored towards whatever purpose the user wishes it to be.

Fourth and final aim of the Fluxion is to be a free and open source alternative to the already established systems that are not as readily as accessible or perhaps extendable as the Fluxion language itself.

### 1.1.1 Fluxion is not a Programming Language

Fluxion is not a programming language. It is a symbolic computation language. However, this specification will refer to the language as *the Fluxion Language* for keeping things short.

### 1.1.2 the Name Fluxion

Fluxion was the original name given to the instantaneous rate of change of time sensitive function (or a Fluent Quantity, a *Fluent*) by Sir Isaac Newton[3]. In essence, Fluxion is an obscure term for derivatives[1]. Given that Fluxion arose from a Clojure program that was made to take derivatives, I felt it is a good name as any to give it.

## 1.2 Contents of the Specification

Subsequent chapters of this specification will talk about different components of Fluxion. Each chapter will talk about a component, its syntax, semantics and behaviour. Syntaxes of these components will be expressed via the Extended Backus-Naur Form[2] with very minor augmentations.

Next chapter contains the lexical structure of the language via Extended BNF, including the operators of the language, statements, expressions and literals. Third chapter contains these basic literal expressions, as well as the type system of Fluxion. Fourth chapter contains the operations on these literals. Chapter five discusses assignment and function definitions as well as built-in functions of the language. Chapter six discusses reductions, chapter seven discusses differentiation of various algebraic data types and finally, chapter eight discusses the various statements, such as `introduce` and `in`

the language has.

This is the first part of a two parter language specification. As such, the standard library facilities will be introduced in the second part of this specification. Implementation of the standard library is necessary for a complete implementation of the Fluxion language, however, a minimal implementation may leave it out.



## Chapter 2

# Grammar

This chapter includes the relevant grammar for statements, expressions and literals of the Fluxion language in Extended Backus-Naur Form.

## 2.1 Notation

The notation included in this specification is ISO14977 Extended Backus-Naur Form, with the addition of the kleen-star  $\langle a \rangle^*$ , this symbol is used in place of the  $\{\langle a \rangle\}$ , and represents that a certain lexeme may be repeated zero or more times.

## 2.2 Common Lexical Units

Below are some lexemes, either in description, or in extended BNF that will be used as the building block of more complex lexemes. first of all, let us define some production rules informally:

$\langle letter \rangle$  As it is used in the ISO14977, an English character within the ASCII, lowercase or uppercase.

$\langle unicode \rangle$  A character that is either a **letter** or any unicode string that is not an operator.

$\langle digit \rangle$  As it is defined in the ISO14977, a digit is a base ten numeral between 0 and 9.

$\langle space \rangle$  Space, or more formally the ASCII character number thirty two.

$\langle tab \rangle$  Tab, or more formally ASCII character number nine.

$\langle EOF \rangle$  End of file character. May also be substituted for end of string.

$\langle EOL \rangle$  End of line character.

## 2.3 Literals

Literals are the rough equivalents of rvalues in the C programming language. They are used to define "types".

### 2.3.1 Boolean Literals

$\langle boolean \rangle := \text{'True'} \mid \text{'False'}$

### 2.3.2 Numerical Literals

$\langle unsigned \rangle := \langle digit \rangle [ , \langle digit \rangle^* ]$

$\langle integer \rangle := [ \langle sign \rangle ], \langle unsigned \rangle$

$\langle decimal \rangle := \langle integer \rangle , \text{'.'} , \langle unsigned \rangle$

$$\langle imaginary \rangle := \langle number \rangle, \langle sign \rangle, \langle number \rangle, 'i'$$

$$\langle number \rangle := \langle decimal \rangle \mid \langle integer \rangle \mid \langle imaginary \rangle$$

### 2.3.3 Sets and Sequences

Sets and Sequences share the same grammar they differ on semantics. Therefore, both will share the  $\langle set \rangle$  production rule.

$$\langle finite \rangle := '{', \langle expression \rangle^*, '}'$$

$$\langle builder \rangle := '{', \langle identifier \rangle, '|', \langle logical\_condition \rangle, '}'$$

$$\langle set \rangle := \langle finite \rangle \mid \langle builder \rangle$$

### 2.3.4 Matrices and Vectors

Vectors are subset of Matrices, so much so that, there is no special production rule for vectors.

$$\langle row \rangle := \langle expression \rangle, (\langle whitespace \rangle, \langle expression \rangle)^*$$

$$\langle matrix \rangle := '[', \langle row \rangle, ('|', \langle row \rangle)^*, ']'$$

### 2.3.5 General Literal Grammar

$$\langle literal \rangle := \langle boolean \rangle \mid \langle number \rangle \mid \langle set \rangle \mid \langle matrix \rangle$$

## 2.4 Operators

### 2.4.1 Binary Operators

$$\langle algebraic\_operator \rangle := '+' \mid '-' \mid '/' \mid '*' \mid '^' \mid '->' \mid '_'$$

$$\langle comperasion\_operator \rangle := '<' \mid '>' \mid '<=' \mid '>=' \mid '\==' \mid '='$$

$$\langle logical\_operator \rangle := '|' \mid 'in' \mid \langle comperasion\_operator \rangle$$

$$\langle hybrid\_operator \rangle := '&'$$

$$\langle operator \rangle := \langle algebraic\_operator \rangle \mid \langle logical\_operator \rangle \mid \langle hybrid\_operator \rangle$$

### 2.4.2 Unary Operators

$$\langle sign \rangle := '+' \mid '-'$$

$$\langle prefix \rangle := \langle sign \rangle \mid '\backslash'$$

$$\langle postfix \rangle := '' \mid '!'$$

$$\langle unary\_operator \rangle := \langle prefix \rangle \mid \langle postfix \rangle$$

$\langle hybrid\_operator \rangle$ s are named as such as they can either be parsed as  $\langle logical\_operator \rangle$ s or as  $\langle comperasion\_operator \rangle$ s depending on the literals they are used with. In cases where ambiguity can arise, their algebraic versions have precedence.

## 2.5 Variables and Expressions

$$\langle identifier \rangle := [\langle identifier \rangle, '::'] \langle letter \rangle, \langle unicode \rangle^*$$

$$\langle function \rangle := \langle identifier \rangle, '(', (\langle variable \rangle \mid \langle literal \rangle \mid \langle expression \rangle), ')'$$

$$\langle variable \rangle := \langle identifier \rangle \mid \langle function \rangle$$

$$\langle cardinality \rangle := '!', \langle expression \rangle, '!'$$

$$\langle expression \rangle := \langle cardinality \rangle \mid [\langle prefix \rangle], (\langle literal \rangle \mid \langle variable \rangle) [, \langle operator \rangle, \langle expression \rangle], [\langle postfix \rangle]$$

$$\langle logical\_condition \rangle := ['\backslash'], (\langle boolean \rangle \mid \langle variable \rangle) [, \langle logical\_operator \rangle \langle logical\_condition \rangle]$$

$$\langle assignment \rangle := \langle variable \rangle, ':=', \langle expression \rangle$$

Here the  $\langle function \rangle$  production rule covers the function definition, as well as the function calls.

## 2.6 Statements

$$\langle introduce \rangle := \text{introduce}, \langle identifier \rangle$$

$$\langle solve \rangle := '?', \langle expression \rangle$$

$$\langle export \rangle := \text{export}, \langle identifier \rangle, (' , ' \langle identifier \rangle)^*$$

$$\langle statement \rangle := \langle solve \rangle \mid \langle introduce \rangle \mid \langle export \rangle$$

## 2.7 General Program Grammar

$$\langle line \rangle := \langle expression \rangle^*, \text{EOL}$$

$$\langle multiline\_comment \rangle := ';*', \langle unicode \rangle^*, '*;'$$

$$\langle single\_comment \rangle := ';;', \langle unicode \rangle^*, \text{EOL}$$

$$\langle multiline \rangle := \langle line \rangle, (\langle tab \rangle \mid (\langle space \rangle \langle space \rangle)) \langle line \rangle^*$$

$$\langle unit \rangle := \langle multiline \rangle \mid \langle statement \rangle \mid \langle assignment \rangle$$

$$\langle program \rangle := (\langle unit \rangle \mid \langle comment \rangle)^*, \text{EOF}$$

Multiline statements can be achieved by putting two spaces or a tab character in a line, in such case, programs treat this line as the continuation of the previous line.



## Chapter 3

# Fluxion's Type System

This chapter includes the explanation of the Fluxion's type system from a programming language perspective.

### 3.1 An Overview of the Fluxion's Type System

Fluxion is a dynamically typed language. Where the type of a variable is inferred by the language. Semantics of the operators change depending on which types they are acting upon in a given context.

All Fluxion types are immutable. As such, variables and literals when passed to functions, only pass by reference and never by value.

Fluxion has four main types. `boolean`, `number`, `set`, `matrix`. Of these four types, `number` is split into `integer`, `decimal` and `imaginary`; a subset of `set` types are `sequence` types and a subset of `matrix` types are `vectors`. `number` and `matrix` types can interact directly within an expression depending on the circumstances (which will shortly be described). Members of `sequences` can also be called upon to interact with `number` and `matrix` types, however, `boolean` types cannot interact with other types directly. Therefore, `number`, `set` and `matrix` types are called *algebraic* types.

#### 3.1.1 Unbound Variable

Since Fluxion is a Symbolic Computation Language, users may wish to leave variables without definition. In such cases, it is not guaranteed what a variable may represent. Such variables are called *Unbound variables*. If the variable `x` is undefined, the expression `x^2 + 2` is still a valid expression.

#### 3.1.2 Associated Grammar

Each Fluxion literal type can be produced by using an associated production rule. Types that adhere to the same production rules are special cases of their more general types.

Table 3.1: Grammar rules associated with each type.

	Rule	Relevant Section
<code>boolean</code>	$\langle \textit{boolean} \rangle$	§2.3.1
<code>number</code>	$\langle \textit{number} \rangle$	§2.3.2
<code>integer</code>	$\langle \textit{integer} \rangle$	§2.3.2
<code>decimal</code>	$\langle \textit{decimal} \rangle$	§2.3.2
<code>imaginary</code>	$\langle \textit{imaginary} \rangle$	§2.3.2
<code>set</code>	$\langle \textit{set} \rangle$	§2.3.3
<code>sequence</code>	$\langle \textit{set} \rangle$	§2.3.3
<code>matrix</code>	$\langle \textit{matrix} \rangle$	§2.3.4
<code>vector</code>	$\langle \textit{vector} \rangle$	§2.3.4



## 3.2 Booleans

Fluxion has two boolean values, **True** and **False**. These values may arise from logical operations. Unlike many languages, algebraic values themselves do not carry any significance in terms of boolean values.

## 3.3 Numbers

Numeric types are types that carry numeric values. They can be decimals, integers or imaginary numbers. A numerical literal written without a sign is presumed to be positive.

### 3.3.1 Integers

Integers are whole numbers. Fluxion allows an integer value to be between  $-(2^{64} - 1)$  and  $2^{64} - 1$ . Trying to assign a variable to an integer literal above this number, or attempting to operate on this number will result in an exception.

```
32
-23
0
```

Snippet 3.1: Example integers

### 3.3.2 Decimal

Decimals are numbers with a decimal point. They must be stored as fractions of integers within the language, thus allowing for high precision and easier reduction of expressions.

```
3.12
+4.234
-3.12
```

Snippet 3.2: Example decimals

### 3.3.3 Imaginary numbers

Imaginary numbers are created by summing a number with another number multiplied with  $i = \sqrt{-1}$ . This also makes **i** a reserved keyword. If **a := b + c\*i**, **b** is called the real part and **c** is called the imaginary part. Both **a** and **b** can take any value their type allows. (if **a** is an **integer**, it can be between negative and positive  $2^{64} - 1$ )

```

3 + 2i
a + b*i
-3 - 1i

```

Snippet 3.3: Example imaginals

### 3.3.4 Irrational Numbers

Irrational numbers are undefinable by normal user programs in Fluxion. However, special Irrational numbers that are often used in math, namely  $\pi$ ,  $\tau$  and  $e$  are predefined in the language as `pi`, `tau` and `euler`. Functions that act on these values are hardcoded to resolve them in mathematically correct ways.

For instance, `sin(pi)` resolves to 0 and `ln(euler)` resolves to 1. In expressions where such resolutions are impossible these variables are retained as variables. For more information, please refer to §6.1.2.

## 3.4 Matrices

A matrix consists of rows and columns. A matrix of  $n$  rows and  $m$  columns is told to be a matrix of size  $m \times n$ . In Fluxion, matrices may be as big as the memory allows. Moreover, matrices may hold values of arbitrary types, including other matrices.

```
[1 2 3 | a b c | (12 + a) | (13 + c) | 3]
```

Snippet 3.4: An example matrix

### 3.4.1 Vectors

Vectors are Matrices of dimension  $m \times n$  where at least one of  $m$  or  $n$  equals one. Vectors can be horizontal or vertical.

```
[1 2 3]
[1 | 2 | 3]
```

Snippet 3.5: Example vectors

## 3.5 Sets

Sets are container types that may be infinite or finite. They can be either defined using set builder notation, or by putting elements inside curly brackets. Sets are unordered, therefore, one cannot get elements outside a list. But one can check if an element is inside a list.

$\{1, 2, 3\}$  ; A finite set.  
 $\{\}$  ; An empty set.  
 $\{x \mid x \text{ in } dN\}$  ; A set with set builder notation.

Snippet 3.6: Example vectors

### 3.5.1 Sequences

Sequences are special lists, they are either lists that are finite or countably infinite. A set is said to be countably infinite if its members can be mapped one to one with the set of natural numbers. More formally, Sequences are sets that are enumeratable, for this reason, sequences are ordered.

### 3.5.2 Predefined Sets and Sequences

Fluxion programming language comes built-in with certain sets and sequences. First of these is the `units` sequence, which contain first three base vectors in order,  $\vec{i}$ ,  $\vec{j}$  and  $\vec{k}$ .

#### Number Sets

More usefully, Fluxion comes predefined with sets that represent number sets in math, these are called *domains* (domain of natural numbers, etc.). Some of these, ones that can be mapped to natural numbers via bijection are also sequences.

Table 3.2: Domains, their Fluxion equivalents and their enumeration method as sequences, if it exists.

	Variable	Enumeration
$\mathbb{N}$	<code>dN</code>	$n \rightarrow n$
$\mathbb{Z}$	<code>dZ</code>	$\{0, -1, 1, -2, \dots\}$
$\mathbb{Z}^+$	<code>dZp</code>	$\{1, 2, 3, 4, 5, \dots\}$
$\mathbb{Z}^-$	<code>dZn</code>	$\{-1, -2, -3, \dots\}$
$\mathbb{Q}$	<code>dQ</code>	§10.3.1
$\mathbb{Q}^-$	<code>dQn</code>	§10.3.2
$\mathbb{Q}^+$	<code>dQp</code>	§10.3.3
$\mathbb{R}$	<code>dR</code>	—
$\mathbb{C}$	<code>dC</code>	—



## Chapter 4

# Operations

This chapter includes how operators act upon different variable types and the results of these operations.

Table 4.1: A classification of operators in Fluxion

Domain	Arity	Position	Operators
Algebraic	Binary	Infix	$+$ , $-$ , $/$ , $*$ , $\wedge$ , $\&$ , $\text{in}$ , $->$ , $_$
	Unary	Prefix	$+$ , $-$ , $\backslash$
		Postfix	$!$ , $'$
		Midfix	$  $
Boolean	Binary	Infix	$<$ , $>$ , $<=$ , $>=$ , $=$ , $\backslash=$ , $\&$ , $ $
	Unary	Prefix	$\backslash$
Scope	Binary	Infix	$::$ , $:=$

## 4.1 Operator Semantics

As established in §2.4, Fluxion has operators of many types, moreover, semantics of operators may change depending on the type of the value they are acting upon, akin to operator overloading.

Immutability of the Fluxion types mean that, when an operator act upon two expressions of any type, the result evaluates to another expression. This may be a algebraic value, an unbound variable, a smaller operation or the same operation.

Operations are the most top level expression, whose grammar is defined within §2.5 as  $\langle expression \rangle$ . Operations generally consist of one or more expressions (called operands) bound by an operator. These operators can be categorised in different ways: By their *domain*, the type of variables they act on; by their *arity*, the number of variables they take and by their *position* with respect to the variables they take.

As can be seen in Table 4.1, all Fluxion operators either take one operand, and hence they are *unary* or they take two operands, and hence they are *binary*. They either act upon algebraic and boolean types, although, same symbol maybe used for different meanings, and hence may take occupy more than one category.

All binary operators in Fluxion are also infix operators, which means they are placed in between the expressions they act upon, for instance, the algebraic sum operator  $+$  is placed between two expressions, as in  $\mathbf{a} + \mathbf{b}$ . Prefix unary operators are placed before the expression they act upon, for instance, boolean not operator  $\backslash$ , is used as  $\backslash\mathbf{True}$ . On the other hand, postfix unary operators are placed following the expression they act upon,

such as  $\mathbf{a}!$ .

The midfix operator is the cardinality operator that is generally used for size and length related calculations. It is used such as  $|\mathbf{a}|$ . The scope operators are operators that somehow modify the variable scope and are outside the scope of this chapter. (§5, §10)

Algebraic Operations (but not operators) can also be classified as homogeneous and heterogeneous operations. Homogeneous operations are either binary operations that act on the same types of operands, or unary operations, whereas the heterogeneous operations are operations that act between two different types of values. For instance, in mathematics,  $12!$  and  $\vec{a} \times \vec{b}$  is homogeneous, whereas  $12\vec{a}$  is heterogeneous.

## 4.2 Homogeneous Operations

Domains	
Numbers	$+, -, /, *, \wedge, ',   $
Integers	$!$
Imaginals	$\backslash$
Sets	$+, -, \text{in}, \backslash,   , ', \&, *$
Matrices	$+, -, \&,   $
Vectors	$*$

Table 4.2: Domains of homogeneous algebraic operators, an homogeneous operator requires all of its operands to be of this type to work correctly.

### 4.2.1 Numerical Operations

#### Common Operations

Operations of addition (+) and subtraction(-) work commonly across all numeric types. All numbers can be written of form  $a + bi$ , and hence, summation of  $\mathbf{x} := \mathbf{a} + \mathbf{b}*\mathbf{i}$  and  $\mathbf{y} := \mathbf{c} + \mathbf{d}*\mathbf{i}$  evaluate to  $((\mathbf{a} + \mathbf{c}) + (\mathbf{b} + \mathbf{d})*\mathbf{i})$ . Substruction of these terms  $\mathbf{x} - \mathbf{y}$ , evaluates to  $(\mathbf{a} - \mathbf{c}) + (\mathbf{b} - \mathbf{d})*\mathbf{i}$

Operation of multiplication (\*) is trivial mathematical multiplication when both sides are doubles or integers. When one of the sides are an imaginary number, real number part is distributed on both the imaginary and real parts of the number, evaluating into another imaginary number in most

cases, such as  $x(a + bi) = (ax + bxi)$  mathematically. When both sides are of the imaginary type, the multiplication is expanded mathematically  $(a + bi)(c + di) = (ac + (a + b)i - bd) = (ac - bd) + (a + b)i$ .

Division  $/$  works on a similar manner when both sides are real numerals. When the right hand side of the expression is real. When the left hand side is an imaginary number and the right a real number, distributive property is applied similar to multiplication.

Exponentiation  $^$  is the trivial mathematical exponentiation for real numbers. However, for  $x, z \in \mathbb{C}$  and  $z = a + bi$ , expression  $x^z$  evaluates to  $x^a(\cos(b \ln |x|) + i \sin(b \ln(x)))$ . However, since the logarithms are the inverse of the exponential function, their values are only unique to a value of  $2k\pi$  and as such  $e^{\ln |a| - 2k\pi}(e^{2k\pi} \cos(b \ln |x|) + e^{2k\pi} i \sin(b \ln(x)))$  [4], where any value of  $K \in \mathbb{Z}$  is a valid answer to this question. As such, the Fluxion evaluates an exponentiation operation where the exponent is an imaginary number to a sequence of answers, ordered the same way as  $\mathbb{Z}$  ( $\mathbf{dZ}$ ).



# Bibliography

- [1] Collins English Dictionary. fluxion.
- [2] Information technology - Syntactic metalanguage - Extended BNF. Standard, International Organization for Standardization, Geneva, CH, December 1996.
- [3] Isaac Newton. *the Method of Fluxions and Infinite Series: With Its Application to the Geometry of Curve-lines*. Printed by Henry Woodfall; and sold by John Nourse, 1736.
- [4] robjohn (<https://math.stackexchange.com/users/13854/robjohn>). Complex numbers as exponents. Mathematics Stack Exchange. URL:<https://math.stackexchange.com/q/1859299> (version: 2016-07-14).