



$z^3$

*A 21<sup>st</sup> Century Language*

*ZCubes, Inc.*

<http://www.zcubes.com>

A  
21<sup>st</sup>  
CENTURY LANGUAGE  
FOR NATURAL, SIMPLE AND POWERFUL  
HUMAN COMPUTER INTERACTION.

More information @ <http://wiki.zcubes.com>

$z^3$

*A 21<sup>st</sup> Century Language*

BY

*ZCubes, Inc.*

SILKRAYS PUBLISHING, TEXAS  
UNITED STATES OF AMERICA.

z^3

*A 21<sup>st</sup> Century Language*

Copyright © 2016 by ZCubes, Inc.

First Edition

All rights reserved. No part of this book may be used or reproduced by any means, graphic, electronic, or mechanical, including photocopying, recording, taping or by any information storage retrieval system without the written permission of the publisher except in the case of brief quotations embodied in critical articles and reviews.

Silkrays books may be ordered through booksellers or by contacting:

Silkrays Publishing Corporation  
www.silkrays.com  
silkrays.publishing@gmail.com

The views expressed in this work are solely views that are of the author and do not necessarily reflect the views of the publisher, and the publisher hereby disclaims any responsibility for them.

ISBN: 978-0-9824326-4-8 (pbk)

Printed in the United States of America.

Silkrays Rev. Date: 12/30/2016

Dedicated to the:

*Innovators,  
who make all things happen.*

*And teachers,  
like Dr. Salih Yurttas,  
who make such innovators.*

“The difficulty lies not so much in developing new ideas as in escaping from old ones...”

*John Maynard Keynes*  
(*British Economist, 1883-1946*)

\*\*\*\*\*

“Sit down before fact as a little child, be prepared to give up every conceived notion, follow humbly wherever and whatever abysses nature leads, or you will learn nothing...”

*Thomas Huxley*  
(*English Biologist, 1825-1895*)

\*\*\*\*\*

“We use one stage of technology to create the next stage, which is why technology accelerates, why it grows in power...”

*Ray Kurzweil*  
(*Futurist, b. 1948*)

\*\*\*\*\*

“Creativity is just connecting things...”

*Steve Jobs*  
(*American Inventor, 1955-2011*)

\*\*\*\*\*

# Contents

1. Introduction.....	1	7.c.i. Simple Matrix Merging with Functions .....	28
2. What is $\mathbb{Z}^3$ ?.....	1	7.c.ii. Across Matrices Merging with Functions .....	28
2.a. ZCubes Platform .....	1	7.c.iii. Quick Multiplication Tables.....	28
2.b. $\mathbb{Z}^3$ Programming Interface.....	2	7.d. Puzzles and Other Interesting Computations ..	29
2.c. $\mathbb{Z}^3$ Console .....	2	7.d.i. Magic Square .....	29
2.d. ZCubes – Selected Features .....	3	7.d.ii. N-Queens Puzzle .....	30
2.e. Command Line Version of $\mathbb{z}^3$ .....	3	7.d.iii. Birthday Probability .....	31
3. Data Collections.....	4	7.d.iv. Towers Of Hanoi .....	32
3.a. Sets .....	4	7.d.v. Floyds Triangle.....	34
3.b. Set – Simply a Collection of Data .....	4	7.d.vi. Fractals-Mandelbrot .....	34
4. Set – Object Representations.....	5	7.d.vii. Lissajous .....	35
4.a. Set – Complex Set Layouts.....	6	7.d.viii. Graphing Data curve .....	37
4.b. Matrix – as a Set of Set(s) .....	6	7.e. Financial Functions .....	38
4.c. Matrix Operator(   ) .....	6	7.f. Statistical Functions: .....	40
4.d. Set input to Functions .....	8	8. Appendices .....	41
4.e.     Binary Operation .....	8	8.a. Appendix I Operators.....	41
4.e.i. Member Functions of Set.....	9	8.b. Appendix II: Simple Set and Objects .....	41
4.f. @ - Applied To Operator .....	10	8.b.i. Set .....	41
4.f.i. Combinatorial Arguments .....	10	8.b.ii. Associative Set/ Objects .....	41
4.f.ii. Applying Combinatorial Set to Set of Functions... ..	11	8.c. Appendix III: Javascript and $\mathbb{z}^3$ .....	42
4.f.iii. Simple Function Representations .....	11	8.c.i. Using Set Member Functions.....	42
4.f.iv. Easy Multi-Line Representation of $\mathbb{z}^3$ Code .....	11	8.d. Appendix IV Series Generation.....	43
4.f.v. Using     as "Such That" Boolean Expressions. ..	12	8.d.i. Arithmetic Series .....	43
4.g. Associative Set/Composite Set As Objects .....	12	8.d.ii. Geometric Series.....	43
4.g.i. Global Assignments using <<<.....	14	8.d.iii. Prepacked Series.....	43
5. Functions .....	14	8.d.iv. Date Series .....	43
5.a. Set of Functions .....	14	8.d.v. Alphabet Series .....	43
5.b. Simple Reusable Function Declarations .....	15	8.e. Appendix V Member Functions .....	43
5.b.i. Combinatorial Arguments .....	16	9. How to work with $\mathbb{z}^3$ .....	49
5.c. Set \$, \$\$, \$\$\$ and \$ _ Member Functions .....	17		
5.d. Set Functions and Set Programming.....	17		
5.e. Advanced computation of lists .....	18		
5.f. Series computation .....	19		
6. Built-in Functions in $\mathbb{z}^3$ .....	19		
6.a. Permutations and Combinations .....	19		
6.a.i. Common Number Series.....	20		
6.a.ii. Simple Number Stats .....	20		
6.a.iii. Set Operations .....	21		
7. $\mathbb{z}^3$ Simple Examples.....	22		
7.a. Sets and Related Structures.....	22		
7.a.i. Matrices .....	22		
7.a.ii. Toeplitz matrix .....	23		
7.a.iii. Matrix Sizes .....	25		
7.a.iv. Matrix Operations .....	25		
7.a.v. Matrix Arithmetic Operations .....	26		
7.b. Vector Operations.....	26		
7.b.i. Matrix Determinants .....	27		
7.c. Matrix Rotations .....	27		

## 1. INTRODUCTION

Why another programming language? Don't we have enough of them?

Well, let us try this real world experiment. Go to the best programmer you know. Pick the simplest formula you can think of:  $E=mc^2$ . Ask how the Energy (E) can be calculated, for a mass (m) of 1kg, 2kg, 3kg,... 10kg and for a constant Speed of Light ( $3 \times 10^8$  m/s). Let us just watch the programmer for what happens next. Yes, go ahead and start a stop watch!

It is likely that the programmer would pull up a spreadsheet, and type formulae notations into the document such as on the right, and within a minute or so, give you the answers.

C	=3*10^8
M	
1	=D5*\$E\$3^2
=D5+1	=D6*\$E\$3^2
=D6+1	=D7*\$E\$3^2
...	...
=D12+1	=D13*\$E\$3^2
=D13+1	=D14*\$E\$3^2

Or maybe, the programmer would make a program, in some computer language to do this, and will come back to you in about an hour!

Today, an ordinary computer can do billions of operations per second! And even with the best techniques, translating from our human language to computer language takes minutes or hours even for the simplest of equations! *This clearly shows the biggest problem with the current state of the art computer human interaction.*

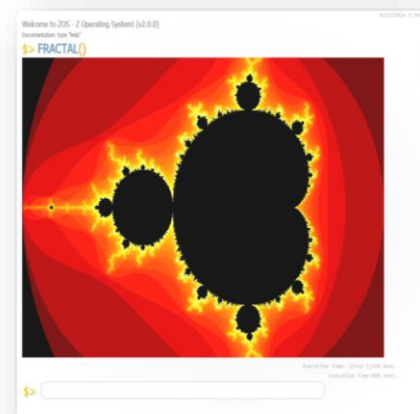
That is why we created a simple language for you and the machine called:

# $z^3$

*The vision behind the  $z^3$  language is specifically to make human interaction with computers convenient, simple and elegant, at any level of complexity, all immersed in a framework of immense power.*

## 2. WHAT IS $Z^3$ ?

$z^3$  is a general purpose language that is **easy to write** and **natural to read**, powered by high performance, scalable, computing constructs which *unlimits* thinking and expression.  $z^3$  console is launched from ZCubes platform on any HTML5 enabled browser.



**FIGURE 1 - FRACTAL PATTERN GENERATED BY USING  $Z^3$ .**

In the following sections of the document,  $z^3$  specifics will be described in easy to follow examples.

### 2.a. ZCubes Platform

ZCubes is a platform for users to create and manipulate information. The website address is

<http://www.zcubes.com>. To load the application simply click on the Z icon, or directly visit it at <http://www.zcubes.com/zspace/zcubes.aspx>.

Being an omni-functional platform, ZCubes allows creation of documents with unparalleled power, with almost any imaginable functionality provided at your finger tips. Upon load, the ZCubes Platform looks as below with a simple minimal interface:

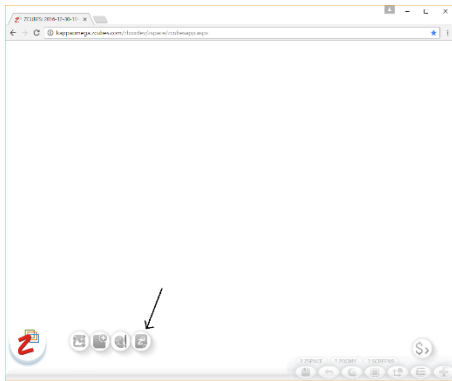
Menu items can be accessed by clicking the Z button.

ZAP is the desktop version of ZCubes that allows deeper access to programming interfaces than the web version. The methods to access  $z^3$  in ZAP is similar to the web-version.

## 2.b. $Z^3$ Programming Interface

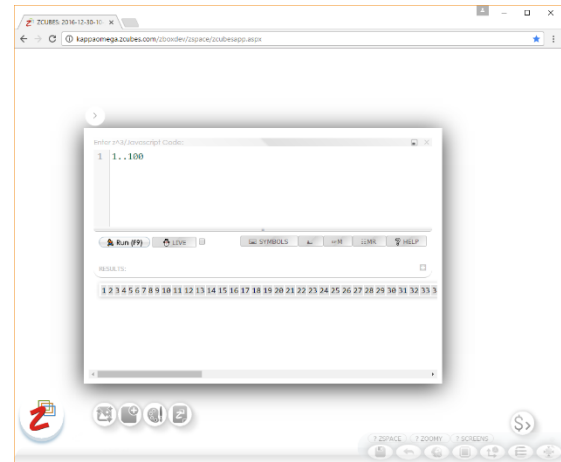
The  $z^3$  Code Cube Editor is a full-fledged programming interface to interact with ZCubes Logic. This code can be a part of the document if kept for more advanced uses. This is the preferred way to interact with  $z^3$  within the ZCubes interface.

To launch the code editor, click on  icon on the main menu of ZCubes Platform.




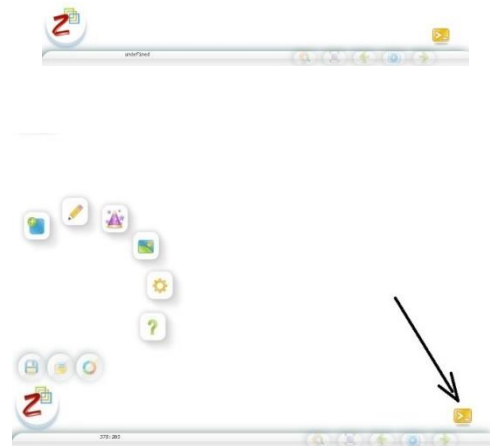
Upon entering code in  $z^3$  code cube, and pressing Run (F9), the results are displayed.

The code can also be interacted in Live Mode, and selected text can be altered dynamically using Live Scroll Bars. This feature is meant for advanced users.



## 2.c. $Z^3$ Console

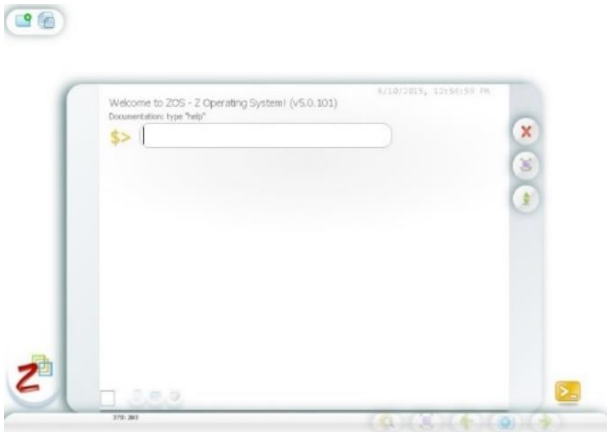
For simple interaction with the programming interface,  $z^3$  console may be used.  $z^3$  console is launched by clicking the icon  on the bottom right of the ZCubes platform. Please refer to Appendix 1 for operators, symbols and notations used in ZOS.



The ZOS Console to interact with  $z^3$  can be accessed using the command console button at the extreme right bottom.



Commands can now be typed into the *Enter ZOS Command* area as indicated below.



Entering command like `1..10@SIN` and pressing enter gives you the result in the window.

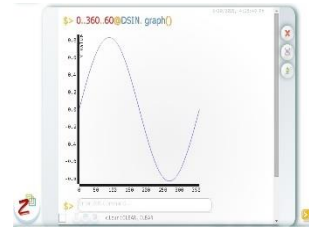
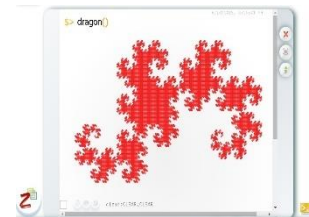


## 2.d. ZCubes – Selected Features

ZCubes is a 3D platform, which changes its nature based on the user's perspective.



For example, it can function as a blackboard in one moment, and a presentation tool in another moment, or a spreadsheet in yet another moment.

The platform changes like a chameleon based on the attributes users wish to have anytime. More details on how to work with ZCubes is explained in section 8.

## 2.e. Command Line Version of z^3

Server programs written as .z3 files can be run using the z^3 command line compiler/interpreter.

Windows versions are available at the moment. Mac and Linux versions will be following soon.

The latest version can be downloaded from <http://downloads.zcubes.com/zconsole/z3compiler.zip>

After downloading, the zip file can z3compiler.exe can be extracted a folder (such as c:\z3). This program can run any .z3 files, at the OS Command Line using command such as

➤ `z3compiler -i myprogram.z3`

Let us now dive into the z^3 language. Let us start with data structures in the next chapter.

### 3. DATA COLLECTIONS

Most are familiar with matrices, and arrays are used in languages to represent matrix like data structures. An array is a simple data structure to create, collect and manage data.  $z^3$  transforms conventional arrays into something much more powerful called *Sets*.

#### 3.a. Sets

*Sets* are new data types used in  $z^3$ . These are arrays (not necessarily rectangular) that are flexible in size, shape, types and contents, which make them extremely powerful. The term *set* is italicized throughout the document for easy identification.

The following are notable properties of *Sets*, compared to conventional arrays:

- *Sets* are unstructured arrays, of varied sizes and types.
- *Sets* may contain other *Sets of any complexity*.
- *Sets* are enhanced with several member functions<sup>1</sup> in  $z^3$ .

For example, *sets* can be printed out with the built-in member function **print()**, to get the internal representation in  $z^3$ .

*Set*-based  $z^3$  resolves complexity and scaling issues, while achieving high-performance, extreme flexibility or natural expressiveness.

#### 3.b. Set – Simply a Collection of Data

Let us start with a simple example.

At the ZOS Console, right after the command prompt indicated by , enter the command **1..3**. Let us use the member function .print() to display the *set* representation in  $z^3$ .

```
1. 1..3.print()
```

<sup>1</sup> A listing of these are given in Section 0

```
[1, 2, 3]
```

The “two dots” operator used in 1..3 creates a **simple set**, with 3 integer elements (1 at index 0, 2 at index 1, and 3 at index 2 positions). *Sets* have indexes starting at 0, which is a common practice in C-like languages.

Once a *set* is created, various operations can be performed on it. As you shall see later, the results of many of these operations are also *sets*, which mean we can continuously apply these operations until desired results are achieved.

An interesting point to note is that the *set* **1..3** can be *implicitly declared* without any extra word or punctuations unlike most languages<sup>2</sup>.

It is also important to see the use of .. operator as a technique to create a sequenced collection of number values (from lower end 1 to upper end 3 - in this case as a range). To create a *set* filled with a series, the [ ] operator is not required. Hence, whenever we use [ ] array operator along with .. operator, it indicates a **set of set(s)**. This is effectively a *set* with index 0 containing three elements (this inside element being similar to the array in *Example 1*).

```
2. [1..3]
```

```
1      2      3
```

This is clearer when we apply .print().

```
3. [1..3].print()
```

```
[ [1, 2, 3] ]
```

In the example above: the internal *set* representation is displayed as **[[1,2,3]]**.

A **set can contain other sets** (containing any type of data) **recursively** (or one within the other without limits), as in the following:

<sup>2</sup> Several Series Generation Techniques are detailed in Appendix IV Series Generation

4. **[2,4,1..3,2].print()**

```
[2,4,
      [1,2,3 ],
2]
```

In this example more complex items (including a *set*) is collected into one *set*: Location at *Index 0* contains integer 2, at *Index 1* contains integer 4, at *Index 2* contains a *set*, and at the location at the last *Index 3* contains integer 2.

*Sets* created can then be operated on using functions (or even *sets of functions*) using the @ operator as shown below:

5. **1..3@COS**

Number	COS
1	0.5403023058681398
2	-0.4161468365471424
3	-0.9899924966004454

However, aggregate functions such as SUM should be applied to the entire array, not to each element.

6. **1..3@SUM**

	SUM
1	1
2	2
3	3

In the example above, the result may appear confusing at first, since SUM function is applied *for each set element*, if we use the @ operator.

The following example shows the application of function SUM() to a *set (not separate elements)*. The three elements of *set* are added, resulting in 6, the expected answer.

7. **SUM(1..3)**

6

Let us see another operator ... (“three dots”) in action.

8. **1...8**

1      2      4      8

The “three dots” notation generates a *set* with 4 values, as a *geometric series* from 1 to 8 as 1, 2, 4, 8. Another example below shows geometric series from 1 to 30.

9. **1...30**

1      2      4      8      16

Now let us look at more complex scenarios.

In the following, the “two dots” (..) operator is compounded to make an even more powerful expression:

10. **1..10..2**

1      3      5      7      9

Here the first .. indicates a series from a *start* value to an *end* value, and the second .. is used as an *increment* operator. Hence, the result includes all numbers from 1 to 10, with an increment of 2.

## 4. SET – OBJECT REPRESENTATIONS

In z^3, {} is used for the creation of object data structures (also known as associative or composite *set* (4.g)).

The JavaScript associate array syntax and semantics are kept as they are.

11. **A={a:1..3,b:1..4}**

```
{
  "a": [
    1,
    2,
    3
  ],
  "b": [
    1,
    2,
    3,
    4
  ]
}
```

In *Example 11*, a *set* with two items (*a* as a *set* of 3 integer values, and *b* with another *set* of 4 integers) is dynamically created.

By simply using = (as assignment operator), *A* is defined as a variable, which now stores an *associative set*, which can then be referenced as below.

12.	<b>A.a</b>	
1	2	3

Now *A.a* refers to the attribute *a* of the newly created variable *A*. Likewise, *A.b* will display the contents of item *b* of variable *A*.

#### 4.a. Set – Complex Set Layouts

Now, consider a simple *set* of three elements (equivalent to 1..3):

13.	<b>[1,2,3]</b>		
1	2	3	

A more complex layout of a similar *set* is given below, where 1 and 2 are in one element (stored as a sub *set*) of the new *set*.

14.	<b>[1..2,3]</b>
1	2
3	

*Examples 15 to 18* below show slight variations on how non-regular layouts of *set* elements can be defined.

15.	<b>[1,2..3]</b>
1	
2	3

16.	<b>[1..3,1..3]</b>		
	1	2	3
	1	2	3

17.	<b>[1..2,1..5]</b>
1	2

1	2	3	4	5
18.	[1,1..2,4]			
1				
1	2			
4				

These examples clearly demonstrate that using clear and elegant operators in *z^3* (such as .. and ...), simple, consistent and powerful structures can be created.

#### 4.b. Matrix – as a Set of Set(s)

Arguably, a *matrix* (which can be represented as a *Set of Set(s)*) is one of the most frequently needed and useful data structures in programming.

A *table* can be visualized as an example of a matrix – such as a table containing rows and columns in a database, or a simple table with rows and columns in a simple document or the tabular grid(s) in spreadsheets.

In mathematical terminology, matrix definitions and usage are very common - such as identity matrix, diagonal matrix, sparse matrix, and so on. Hence, operations on matrices are provided by *z^3* using a rich *set* of operators and functions.

#### 4.c. Matrix Operator(| |)

Matrix construction can be done with the | | operator. For example, creation of a 4x4 identity matrix can be done with the following simple notation:

19.	4		
1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

|4| as it is displayed above gives a 4 by 4 identity matrix by definition. Note that the matrix operator | | is used here for both matrix declaration and initialization.

If only one dimension is given, it is assumed that the matrix requested is an identity matrix.

This highlights the simple and minimalist style provided by `z^3` throughout. Similarly, `|4,2|` constructs a 4x2 matrix.

---

20.     **| 4 , 2 |**

---

0	0
0	0
0	0
0	0

Similarly, `|4,2,2|` constructs a 4x2 matrix, with each cell having 2 values, all initialized to zero:

---

21.     **| 4 , 2 , 2 |**

---

0 0	0 0
0 0	0 0
0 0	0 0
0 0	0 0

The variations of matrix construction can be seen in the following examples.

The notation inside matrix definition can be with commas, the letter x, or a simple space.

---

22.     **| 2 x 2 |**

---

0	0
0	0

A square matrix of 2 by 2 with initial values 0, is given by this notation.

---

23.     **| 3 x 3 |**

---

0	0	0
0	0	0
0	0	0

Here a square matrix of 3 by 3 with initial values 0 is obtained. Similarly, you can define `|4x2|` or `|2x4|` as given below.

---

24.     **| 4 x 2 |**

---

0	0
0	0
0	0
0	0

---

25.     **| 2 x 4 |**

---

0	0	0	0
0	0	0	0

Now, let us look at some more advanced examples.

The example above gives a zero-filled 2 by 2 by 2 matrix (i.e., a matrix of 2 by 2, with each element containing a zero-filled 2-element matrix).

---

26.     **| 2 x 2 x 2 |**

---

0 0 0 0
0 0 0 0

The following gives a zero-filled matrix of 2 by 2 with 3 elements each.

---

27.     **| 2 x 2 x 3 |**

---

0	0	0	0	0	0
0	0	0	0	0	0

---

28.     **| 2 x 3 x 3 |**

---

0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

The matrix notation can scale-up for more complex uses, and for any size and level of dimensions.

#### 4.d. Set input to Functions

The simplest case is SUM of *set* 1,2,3 is calculated by the following:

---

29.      **SUM 1..3**

---

6

In the following *Example 30*, the *set* 1..3 and 2..4 are passed to the function SUM, and it gives the sum of all the numbers in both *set*. Compare this later, against the combinatorial arguments applied to functions (detailed in Section 4.f.i below).

The following *Commands (30, 31, 32)* calculate the SUM of all parameters (each of which may be a set containing a series) provided in each case.

---

30.      **SUM 1..3 2..4**

---

15

which is the same as:

---

31.      **SUM(1..3,2..4)**

---

15

---

32.      **SUM 1..3 2..4 1..4**

---

25

We can have any number of arguments to aggregate functions.

Commas and parenthesis can be omitted for simpler calls. For more complex calls, commas and parenthesis may be required to avoid ambiguity. However, it is always recommended use the brackets to avoid confusing situations.

---

<sup>3</sup> *x* and *y* represent variables from sets on each side of the binary operation. *x* and *y* can be replaced by any two names for the variables.

#### 4.e. | | Binary Operation

The binary operation with matrix operator | |, called the “two bars” operator, is simple and natural to use.

Two matrices of compatible sizes can be operated on with matrix operations, such as |+|, |-|, |\*|, and | / |, as given below.

---

33.      **[1,2,3] |+| [1,2,3]**

---

2                      4                      6

---

34.      **[1,2,3] |-| [1,2,3]**

---

0                      0                      0

---

35.      **[1,2,3] |\*| [[1],[2],[3]]**

---

14

---

36.      **[1,2,4] | / | [1,1,2]**

---

1                      2                      2

---

37.      **[[2],[2],[2]] |\*| [1,2,3]**

---

2                      4                      6  
2                      4                      6  
2                      4                      6

Also the two bars of this operator can brace *any arbitrary function or operator*.

*Examples 38 to 41* show combining of two matrices using a simple function represented with *x* and *y*, provided in between the | | operator<sup>3</sup>.

---

38.      **1..3 |x+y| 3..5**

---

4                      6                      8

39. `1..3|x*y|3..5`

3	8	15
---	---	----

40. `1..3|(x^2+SIN(y))|3..5`

1.1411200080598671	3.2431975046920716	8.041075725336862
--------------------	--------------------	-------------------

41. `1..3|SIN(x)+y|3..5`

3.8414709848078963	4.909297426825681	5.141120008059867
--------------------	-------------------	-------------------

#### 4.e.i. MEMBER FUNCTIONS OF SET

A variety of useful and powerful member functions (listed in 0

*Appendix V Member Functions*) are provided for *sets*. Examples include, *.print*, *.\$*, *.index*, *.tenth*, *.random*, etc.

By convention, member functions are generally in *all lowercase*, while primary functions are generally in *all uppercase*. These member functions are invoked using *dot-notation*, like most conventional object-oriented languages.

42. `|4x2|`

0	0
0	0
0	0
0	0

For example, *.transpose()* is a member function that will return the transpose of the matrix.

43. `|4x2|.transpose()`

0	0	0	0
0	0	0	0

44. `|3x3|.random(3)`

0.4868065193295479	1.9101847931742668	0.25111658102832735
0.5587615873664618	0.26558934850618243	2.996888898080215
0.9493648773059249	2.7146050329320133	2.2170031929854304

With the simple notation above, a 3x3 matrix is created, and filled with random numbers up to 3. The following shows the use of *.fillwith()* to fill the dynamically created matrix of size 3x3.

45. `|3x3|.fillwith(8)`

8	8	8
8	8	8
8	8	8

Note that in the case above all cells are filled with 8, and in the case below a series of numbers from 1..9 are used.

46. `|3x3|.fillwith(1..9)`

1	2	3
4	5	6
7	8	9

Again, matrix dimensions can be given implicitly (such as `|2|` for a `|2x2|` matrix).

Functions like *random()*, *deal()*, and many other available functions listed in the *Section 0*

*Appendix V Member Functions* can be used to manipulate *sets*.

47. `|2|.deal()`

0.44067314197309315	0.14973507658578455
0.15162911941297352	0.7144281121436507

48. `|2x3|.deal()`

0.24147144774906337	0.4198728590272367	0.07390831736847758
0.48515111417509615	0.0688244975153704	0.28236311418004334

49. `|3x3|.random(3)`

creates a 3 by 3 *set* and fills it with random numbers with values up to 3.

50. `A1=|2x2|.fillwith(3..6)`

3	4
5	6

The *.fillwith()* member function is used to fill a 2 by 2 matrix with numbers 3, 4, 5, and 6.

The member function *det* calculates the determinant of a Matrix.

---

51. **A1.det()**


---

-2.0000000000000001

Note that the simple function *DET()* and *det()* also can be invoked to calculate the determinant of a Matrix.

---

52. **det(A1)**


---

-2.0000000000000001

The statement above will calculate the determinant of the matrix A1 created in the previous step (*Example 50*).

---

53. **IM(3).across(IM(3))**


---

11	01	01	10	00	00	10	00	00
01	11	01	00	10	00	00	10	00
01	01	11	00	00	10	00	00	10
10	00	00	11	01	01	10	00	00
00	10	00	01	11	01	00	10	00
00	00	10	01	01	11	00	00	10
10	00	00	10	00	00	11	01	01
00	10	00	00	10	00	01	11	01
00	00	10	00	00	10	01	01	11

The *.across()* member function of *set* is powerful way to operate on two matrices, cell by cell. The result of this operation applies the operation among EVERY combination of cells among the matrices<sup>4</sup>.

Even when we graduate into more complex combinations of computational constructs, the language keeps its fundamental simplicity and consistency. Looping construct is rarely needed in  $z^3$ , as functions can be applied iteratively with

---

<sup>4</sup> This is similar to the TENSOR product of matrices. Here it simply provides the row column .index() as result.

implicit initialization (and increments) in a natural order and minimalistic style.

While *set* and functions can be used in very rich ways with  $z^3$ 's syntactical simplicity, traditional programming language syntax and style also can be used. This is beneficial for backward compatibility and for creating clever new possibilities of mixing traditional and newer styles.

#### 4.f. @ - Applied To Operator

To apply a function or *set* of functions to a *set* of values, we use the APPLIED TO operator, indicated by @.

---

54. **1..360@DSIN**


---

gives you a list of SIN(x) values for input ranging from 1 to 360.

The @ operator can be a powerful ally when mixed with the concept of *Combinatorial Arguments* detailed below.

#### 4.f.i. COMBINATORIAL ARGUMENTS

The @ operator differs from a simple call of a function on a *set* with a simple twist. The @ operator treats input data *set* as *Combinatorial Set*.

That is, in [1..4,2..3]@SUM, the SUM function is applied to **COMBINATIONS of values in the two sets**, namely [1,2,3,4] and [2,3] (denoted by 1..4 and 2..3).

Note that this is not the same as the *set* [[1,2,3,4],[2,3]] being treated as input to the SUM function. The following example makes this clear:

---

55. **[1..4,2..3]@SUM**


---

	-	SUM
1	2	3
1	3	4
2	2	4
2	3	5
3	2	5



3	3	6
4	2	6
4	3	7

Every combination from each set 1..4 and 2..3 are paired as parameters into the SUM function in the listing above. This powerful function enables users to avoid loop-in-loop constructs that are often seen in C-like programming (e.g., for i=1..4 and for j=2..3).

By extension, combinatorial arguments can be extended to inner loops of ANY depth (i, j, k, l, m, ... etc)

We can also reverse two sides of the @ operator, as given below where the function SUM is applied to the combinatorial data given on the right, giving the same result as *Example 55*.

56. **SUM@ [1..4, 2..3]**

Compare this with *Example 31*, which is similar but with simple application of SUM to the full set of input, without any combinatorial loop.

#### 4.f.ii. APPLYING COMBINATORIAL SET TO SET OF FUNCTIONS

In  $z^3$ , data sets can be collected into any number of set or combinations, and applied to any number of functions. It does not matter whether data or functions are given first.

The following examples shows this expressive power of  $z^3$ , with data and/or functions separated by comma, which behaves as a list operator.

57. **[1..4, 2..3]@ [SUM, AVG]**

	-	SUM	AVG
1	2	3	1.5
1	3	4	2
2	2	4	2
2	3	5	2.5
3	2	5	2.5
3	3	6	3
4	2	6	3
4	3	7	3.5

The inputs are generated and applied in a natural order iteratively. This makes  $z^3$  keep its readable form, without needing a lot of complex looping expressions.

#### 4.f.iii. SIMPLE FUNCTION REPRESENTATIONS

Functions can be made on the fly using expressions within quotes as given below (example: "x^2+5\*y")

58. **[1..4, 2..3]@ [SUM, AVG, "x^2+5\*y"]**

	-	SUM	AVG	$x^2+5 \cdot y$
1	2	3	1.5	11
1	3	4	2	16
2	2	4	2	14
2	3	5	2.5	19
3	2	5	2.5	19
3	3	6	3	24
4	2	6	3	26
4	3	7	3.5	31

59. **[1..4, 2..3]@ [SUM, AVG, "x^2+67\*y", "SIN(COS(z))"]**

	-	SUM	AVG	$x^2+67*y$	SIN(COS(z))
1	2	3	1.5	135	0.5143952585235492
1	3	4	2	202	0.5143952585235492
2	2	4	2	138	-0.4042391538522658
2	3	5	2.5	205	-0.4042391538522658
3	2	5	2.5	143	-0.8360218615377305
3	3	6	3	210	-0.8360218615377305
4	2	6	3	150	-0.6080830096407656
4	3	7	3.5	217	-0.6080830096407656

Here, the data set on the left are passed to the set of functions listed on the right.

In the functions that are expressed as strings such as in ("x^2+67\*y") in *Example Error! Reference source not found.*, the symbols x and y represent the first and second of the combinatorial arguments.

This feature allows users to express content of a function in the most natural manner, without complex function declaration decorations.

#### 4.f.in. EASY MULTI-LINE REPRESENTATION OF $z^3$ CODE

Note that code can be split into multiple lines (using the Shift+Enter key) in the ZOS platform editor, as given in the *Commands 60 and 61*.

```

60.  ARRAY (2,2,2)
      .merge (
        ARRAY (2,2,3) ,
        [SUM,AVG,"x+y^2-1"]
      )

```

5	2.5	10
5	2.5	10
5	2.5	10
5	2.5	10

The examples above indicate `.merge()` member function, that operates on two *sets* (one on the left, and one as a parameter), merged with the functions listed as second parameter of the member function.

In the *Example 61*, `.rand()` function fills the *set* created using the `ARRAY` function. Then the `across()` operation takes each such cell combination from each matrix, and then applies the list of functions `[SUM, AVG]` given as the second parameter.

```

61.  ARRAY (2,2,2)
      .rand ()
      .across (
        ARRAY (2,2,3)
        .rand () ,
        [SUM,AVG]
      )

```

The partial output is given below.

0.9995882413350046	0.4997941206675023	1.200744666159153	0.6003723330795765	0.764609636
1.6460307827219367	0.8230153913609684	1.6693326707463712	0.8346663353731856	1.411052177
0.14775128616020083	0.07387564308010042	0.34890771098434925	0.17445385549217463	0.245043733
0.794193827547133	0.3970969137735665	0.8174957155715674	0.4087478577857837	0.89148627

#### 4.f.v. USING || AS "SUCH THAT" BOOLEAN EXPRESSIONS

We can check a *set* against logical expressions, like in the following examples:

```

62.  4..8|x>5|

```

```

false  false  true  true  true

```

Here, the range of values is tested for a logical conditional test, to give *truthiness* of the check in a simple *set* series.

In *Example 62*, the *set* 4..8 is passed on to the check  $x > 5$ . The result is a Boolean *set* of *true* and *false*.

```

63.  1..100|x>5?x:55|

```

In *Example 63*, 1..100 is passed on to the check  $x > 5$ . The result is the value itself or 55, based on the result of the check on each specific element.

It is possible to compound conditions in a traditional sense easily with logical operands like `&&` (*and logical check*) as follows:

```

64.  1..4|x>2&&x<4|

```

```

false  false  true  false

```

#### 4.g. Associative Set/Composite Set As Objects

Object instances can be expressed easily using simple implicit notation. There is no need for explicit *constructors* or *definitions* as may be required in conventional object oriented languages.

```

65.  A1={"a":5, "b":2,
      "c":"something"}

```

```

{
  "a": 5,
  "b": 2,
  "c": "something"
}

```

**A1** is an instance of Object, with initial values given, in lines with Object-Oriented (OO) terminology. The qualification of members (whether it be a value or a function) is coded with traditional `(.)` operator. The result is an object meant for further processing. However, members of this object are now accessible using *A1.a*. For example:

```

66.  A1.a

```

```

5

```

```
67. A1.b
```

```
2
```

```
68. A1.c
```

```
something
```

It is also possible to use member names in quotes as indexes, as given below:

```
69. A1["a"]
```

```
5
```

```
70. A1["b"]
```

```
2
```

```
71. A1["c"] ="anything"
```

```
anything
```

The following example demonstrates these properties:

```
72. factorial(5)
```

```
120
```

The qualified member is used as an argument to a function:

```
73. factorial(A1.a)
```

```
120
```

Assignments are straightforward. The type of the variable value is taken implicitly from the right hand side (RHS) value.

The variable represented by the left hand side (LHS) name, is dynamically recast to the type of the assigned new value.

For example, in the object A1, *c* which was initialized as a string before, can now be used as a place holder for a *function*:

```
74. A1.c=function(a){return(a+8)}
```

```
(a) 1
```

The output indicates that this is a function with a single parameter *a*.

Now the A1.c represents a function, which can be called as given below.

```
75. A1.c(3)
```

```
11
```

Similarly, b in A1 can be assigned to the SIN function with no extra coding:

```
76. A1.b=SIN
```

Now, we can call A1.b as a function, which then gives the SIN value of 30, in radians.

```
77. A1.b(30)
```

```
-0.9880316240928618
```

A similar example of object assignment and member access is given below.

```
78. B1={"aa":2, "f1":"apple",  
"f2":"peach"}
```

```
{  
  "aa": 2,  
  "f1": "apple",  
  "f2": "peach"  
}
```

The members can be qualified with the initial LHS ids for reference, as usual:

```
79. B1.aa
```

```
2
```

```
80. B1.f1
```

```
apple
```

```
81. B1.f2
```

peach

## 4.g.i. GLOBAL ASSIGNMENTS USING &lt;&lt;&lt;

The following example demonstrates the use of attribute assignment operator <<< to assign a *set* of functions to an attribute of an operator:

---

```
82. ["orange"] <<< [SIN, COS]
```

---

orange

Now this assigned a global variable called *orange*, that now contains a *set* of functions SIN and COS.

Please note that SIN and COS take input in radians. DSIN and DCOS take input in degrees.

This *set* of functions in *orange* is applied to a *set* of numbers 1..4, in *Example 83*.

---

```
83. 1..4@orange
```

---

Number	SIN	COS
1	0.8414709848078965	0.5403023058681398
2	0.9092974268256817	-0.4161468365471424
3	0.1411200080598672	-0.9899924966004454
4	-0.7568024953079282	-0.6536436208636119

Also note that *orange* is now a global variable in the environment for easy use (as a *set* or *collection of functions*, which then becomes a powerful object to use to apply multiple functions at once).

## 5. FUNCTIONS

In most languages, a function like SIN takes one input and gives a scalar output. In *z*<sup>3</sup>, the same SIN function behaves in more powerful ways. If given a scalar SIN will give a scalar result like other languages, but if given an arbitrary *set* of inputs, *z*<sup>3</sup> will give a *similar set of outputs*.

Now, more interestingly, *functions* can be:

- (1) *Simple member functions of a set*, as in objects as member functions, or
- (2) *Set of functions*, that provides an elegant organization.

## 5.a. Set of Functions

It is possible to use the expression in (*Example 84*) as a one line computation, or decomposed into multi-line computation (*Commands 85 and 86*).

---

```
84. 1..4@["x^2", "x*2", "x+2"]
```

---

The example below, demonstrates the declaration of three functions, each with one parameter *x*, collected as a *set* and assigned to **A** for easy reuse.

---

```
85. A=["x^2", "x*2", "x+2"]
```

---

POWER(x, 2)	x*2	x+2
-------------	-----	-----

In *z*<sup>3</sup>, the data provided to such *sets* of functions can be a *set* (or *set of sets*) of any breadth and depth.

The *set* of three functions are now invoked on a range of values 1..4 below:

---

```
86. 1..4@A
```

---

x	x <sup>2</sup>	x*2	x+2
1	1	2	3
2	4	4	4
3	9	6	5
4	16	8	6

Note that explicit looping constructs are avoided. The logic represented by the *set* of functions is applied to *each data* value in a sequence 1..4.

This style of applying a *set of data* to a *set of functions* demonstrates *z*<sup>3</sup>'s functional approach (i.e., *writing WHAT is to be computed, and hiding HOW it is computed*).

**Most library functions in *z*<sup>3</sup>, evaluates to a scalar value or a Set of values, as and when needed.**

The built-in **FOREACH** function (same as **FOR** function) can also be used to achieve combinatorial arguments and *set* of functions in a straightforward manner<sup>5</sup>.

87. **FOREACH (1..2, 2..4, "z=x\*3\*y")**

x	y	z
1	2	6
1	3	9
1	4	12
2	2	12
2	3	18
2	4	24

It is clear that the data *set* given as arguments are used from left to right<sup>6</sup>.

The first *set* 1..2 behaves as the outer loop index values, and the secondary *set* 2..4 behaves as the inner increments, for **x** and **y** values respectively, which are associated from left to right.

88. **FOR (1..2, 2..4, "z=x\*3\*y")**

x	y	z
1	2	6
1	3	9
1	4	12
2	2	12
2	3	18
2	4	24

As the following example clearly demonstrates that **FOREACH** is a rich function that takes data *set* in multiple forms, and *sets* of functions collected in a *set*:

<sup>5</sup> Since lowercase 'for' is a standard keyword in some platforms, FOREACH function is provided to side-step any conflict in case of case-sensitivity.

89. **FOREACH (INTS (3) , [SIN,COS] )**

Number	SIN	COS
1	0.8414709848078965	0.5403023058681398
2	0.9092974268256817	-0.4161468365471424
3	0.1411200080598672	-0.9899924966004454

90. **FOR 1..3 SIN**

Number	SIN
1	0.8414709848078965
2	0.9092974268256817
3	0.1411200080598672

The parenthesis and commas can be dropped as in the previous example, if it would not cause a syntactic ambiguity.

91. **FOR 1..4 "x\*x"**

x	TEMP1 <sup>7</sup>
1	1
2	4
3	9
4	16

## 5.b. Simple Reusable Function Declarations

A function can be declared as follows:

92. **Y1:=u\*t+0.5\*a\*t\*2**

Function Y1 with parameters: (u,t,a) is defined as u\*t+0.5\*a\*t\*2

<sup>6</sup> Any extra parameters are simply ignored, while using the matching values to compute the functions.

<sup>7</sup> Please note that unnamed functions are given temporary names (like TEMP1) in outputs.

Instantly, a function Y1 is created, with parameter u, t and a, with parameter <sup>8</sup> names which are automatically detected by z^3, in order of their appearance.

If there are similar names existing in the environment, z^3 will rename the function.

#### 5.b.i. COMBINATORIAL ARGUMENTS

The following gives an example of *series* and *combinatorial arguments* being used to replace the use of a spreadsheet to do such calculations!

93. **FOR(1..3,2,3,Y1)**

u	t	a	Y1
1	2	3	8
2	2	3	10
3	2	3	12

Let's check what would happen if more data values were provided as parameters, than that were defined in the function Y1 (which expects only 3 parameters u, t and a).

94. **FOR(1..3,2,3,8,Y1)**

u	t	a	-	Y1
1	2	3	8	8
2	2	3	8	10
3	2	3	8	12

As expected, the fourth parameter value 8 is ignored and the computation is completed with the first three parameters of the data *set*.

Data can be simply listed following the function with a space as a separator as below.

<sup>8</sup> Global variables can be accessed from the inside of simple function definitions. However, to keep simplicity of variable names in local scope, any global variable up to 3 characters

95. **FOR 1..3 2 3 8 Y1**

If the user prefers to list the data in a parenthesis, data may be listed with a comma (,) operator as a separator. Consider an example of counting of PRIMES, up to a certain number. With z^3 notation, we can repeat this process for any *set* of numbers. In the example below, series of odd numbers (from 1 to 10) are created, and the count of primes that are within 1 to that number are then calculated.

96. **FOR 1..10..2 "COUNT(PRIMES(x))"**

x	COUNT(PRIMES(x))
1	0
3	2
5	3
7	4
9	4

The ranges can now be expanded from 1..10 to 1..100000..10000, demonstrating powerful ways of combining z^3 notations to achieve complex expressive calculations, without sacrificing simplicity and scalability..

97. **FOR 1..100000..10000 "COUNT(PRIMES(x))"**

x	COUNT(PRIMES(x))
1	0
10001	1229
20001	2262
30001	3245
40001	4203
50001	5133
60001	6057
70001	6936
80001	7837
90001	8714

in length referenced inside the body of the function would need an \_ (underscore) prefixed to its name.

### 5.c. Set \$, \$\$, \$\$\$ and \$\_ Member Functions

A *set* of functions can be invoked in every element in a *set* by using the powerful **.\$** (*dollar*) member function.

```
98.      EVENS (4) . $ ("x+2")
```

```
2      4      6      8
```

In this case, four even numbers are generated by the EVEN function, which are then incremented by 2.

```
99.      1..5.$ ("1/x")
```

```
1      0.5      0.3333333333333333      0.25      0.2
```

The above output lists the reciprocals of numbers 1..5.

Remember that **.\$** is not quite the same as *.map* functions that you may be familiar with, since *.map* operations only operate on the children in the first level of the array or *set*. On the other hand, **.\$** *operates on every element in the set, recursively*.

```
100.     SUM (EVENS (4) . $ ("x+2" ) )
```

```
20
```

The result of **.\$** is also a *set*, and can be fed into further member functions to operate on, such as SUM shown above.

Now, consider the following command to add numbers from 1 to 10.

```
101.     SUM (1..10)
```

```
55
```

The member function (**.\$\_**) (*dollar-underscore*) can be used to apply an aggregate function (such as SUM) to the entire set, resulting in a single functional result.

```
102.     1..10.$_ (SUM)
```

```
55
```

The member function (**.\$\$**) (*dollar-dollar*) applies the *set* of functions provided on *each row (by row)* of the *set*.

```
103.     MAGICSSQUARE (4) . $$ (SUM)
```

```
34
34
34
34
```

The member function **.\$\$\$** (*dollar-dollar-dollar*) works *along columns*, compared to (**.\$\$**) that operates *on rows*. The member function **.\$\_** (*dollar-underscore*) function works across the entire *set*.

```
104.     1..10.$_ (SUM)
```

```
55
```

### 5.d. Set Functions and Set Programming

The use of *sets* as collections of statements and function calls creates an interesting possibility of program segments that is self-explanatory.

```
105.     V:=[a,b,[SIN(a),COS(b),SIN(b),COS(a)]]
```

```
Function V with parameters:(a,b) is defined
as [a,b,[SIN(a),COS(b),SIN(b),COS(a)]]
```

The above function definition creates a function when called gives the input a and b, as well as the result, all as ONE *set* in the output!

This creates the possibility of holding data *sets* that carry inputs and outputs of a process for reporting uses or further analysis. Such composites create a rich expression possibility for calculations in  $z^3$ .

```
106.     V(1,2)
```

```
1
2
0.8414709848078965 -0.4161468365471424 0.9092974268256817 0.5403023058681398
```

Here, the results shows inputs 1 and 2 (*a* and *b* in the *set program*), followed by the array of results from

the the computation of the *set of functions*  $[SIN(a), COS(b), SIN(b), COS(a)]$  in the *set program*.

Such *functions defined as set*, gives unique expression power to programmers. In the example in *Command 106*, the input is effectively carried along with the output.

Let us consider some powerful ways to create functions.

---

107. **F1 := [a+b, a-b]**

---

Function F1 with parameters: (a,b) is defined as [a+b, a-b]

The function F1 is now defined, and calling it is simple.

---

108. **F1 1 2**

---

3      -1

Another example that demonstrates the expressive power:

---

109. **F2 := [a, b, a+b]**

---

Function F2 with parameters: (a,b) is defined as [a,b,a+b]

---

110. **F2 5 2**

---

5      2      7

---

111. **F2 (3, 9)**

---

3      9      12

---

112. **F2 2\*3 5**

---

6      5      11

As can be clearly seen above, the outputs carry the inputs as well as the results, creating absolutely interesting possibilities of handling data, as well as avoid unnecessary complexity in programming.

---

113. **F3 := [SUM(a..b..c)]**

---

Function F2 with parameters: (a,b,c) is defined as [SUM(FROMTO(CONCAT(a,b),c))]

This effectively creates a function F3 that can add up series between a and b, of any interval c.

---

114. **F3 1 10 2**

---

25

The sum of 1+3+5+7+9 is obtained as 25 by calling F3 with the parameters 1, 10 and 2.

The following shows another interesting use of *set Programming*, coupled with `||` function definition (Section 4.e).

---

115. **1..4 | x^2+x^3 |**

---

2      12      36      80

Here the function  $x^2+x^3$  itself is a *function expression*. The result of this function is applied on the series 1 to 4.

### 5.e. Advanced computation of lists

Trigonometric calculations are useful in a wide varied domains, such as electrical engineering, map projections etc. These require computation of lists. Series of values thus generated are heavily used during further computation, as inputs, as well as loop control variables; but most languages fail to support quick and easy generation of common collections and series.

$z^3$  makes the generation of such lists incredibly easy, starting with x..y..z notation such as in 1..10..3.

Generation of commonly used angles, in radians or degrees, can be easily achieved as given below. Terms like DEG360BY45 can give easy listing of 360degrees divided by 45degree segments etc. for easy use. Please note the use of the TRANSPOSE operator  $\sim$ , used here for vertical display of the generated *sets*. Similarly rad2pi by 4 (case does not matter) can be used to divide radians into appropriate pieces. These generated values are automatically enabled for units. It helps tremendously with trigonometric calculations etc.

---

116. **DEG360BY45~**

---



0°
45°
90°
135°
180°
225°
270°
315°
360°

117. **RAD2\*PiBY4~**

0rad
1.5707963267948966rad
3.141592653589793rad
4.71238898038469rad
6.283185307179586rad

More examples and details of Series Generation are given in Appendix IV Series Generation.

The series computed can be used as in cases like:

118. **DEG360BY45@DSIN**

Number	DSIN
0°	0
45°	0.7071067811865475
90°	1
135°	0.7071067811865476
180°	1.2246467991473532e-16
225°	-0.7071067811865475
270°	-1
315°	-0.7071067811865477
360°	-2.4492935982947064e-16

## 5.f. Series computation

The design of z^3 attempts to provide natural language interfaces, with terse and powerful notations.

Hence SUM 1.4 can also be implied by the following natural language expressions.

119. **ADD 1 to 4**

10

1 to 4 implies all numbers from 1 upto 4 (i.e., 1, 2, 3 and 4).

Note that this is not the same as ADD 1 4 which should give the result 5.

A series by an increment can be expressed using the *x to y by z* notation.

120. **ADD 1 to 4 by 2**

4

Obviously, the series here can also be represented as ADD(1..4..2)

## 6. BUILT-IN FUNCTIONS IN Z^3

z^3 has powerful prebuilt function libraries. This collection is continuing to grow daily; and already these number into the thousands.

These range from functions in Mathematical, Statistical, Financial, Engineering, Medical, Database, and many other domains.

Several of these standard library functions are documented at <http://wiki.zcubes.com>

Consider some interesting available functions.

### 6.a. Permutations and Combinations

Permutation relates to the act of arranging all the members of a collection into some sequence or order; whereas Combination is a way of selecting items from a collection, such that the order of selection does not matter.

z^3 comes with functions to list permutations and combinations of items, as well as techniques to count and list them.

121. **PERMUTATIONS (1 . . 3)**

1	2	3
1	3	2
2	1	3
2	3	1
3	1	2
3	2	1

All possible permutations of 1, 2 and 3 are shown above.

#### 122. PERMUTATIONS (1 . . 3 , 2)

1	2
1	3
2	1
2	3
3	1
3	2

All possible permutations of 1, 2 and 3, in *sets* of two elements, are shown above.

#### 123. COMBINATIONS (1 . . 3)

1  
2  
3

All possible combinations of 1, 2 and 3, of *one* element are shown. Next, all possible combinations of 1, 2 and 3, of *two* elements are shown.

#### 124. COMBINATIONS (1 . . 3 , 2)

1	2
1	3
2	3

#### 6.a.i. COMMON NUMBER SERIES

EVENS is a function that returns x even numbers, for a requested x.

#### 125. EVENS 4

0 2 4 6

The series thus generated can be input to functions as SUM, or vice versa.

#### 126. SUM (EVENS (4) )

12

#### 127. EVENS (SUM (4) )

0 2 4 6

The following creates a jagged collection of 1, 2, 3 and 4 even numbers.

#### 128. EVENS (1 . . 4)

Limit	EVENS
1	0
2	0 2
3	0 2 4
4	0 2 4 6

#### 6.a.ii. SIMPLE NUMBER STATS

STATS is a powerful function that applies a lot of Statistical Functions on a series of numbers.

#### 129. STATS (1 . . 100)

COUNT	100
SUM	5050
AVERAGE	50.5
VAR	841.6666666666666
STDEV	29.011491975882016
VARP	833.25
STDEVP	28.86607004772212
MIN	1
MAX	100
MEDIAN	50.5

MODE	#N/A

Let us look at a larger collection of numbers.

---

130.      **STATS (1..1000)**

---

COUNT	1000
SUM	500500
AVERAGE	500.5
VAR	83416.66666666667
STDEV	288.8194360957494
VARP	83333.25
STDEVP	288.6749902572095
MIN	1
MAX	1000
MEDIAN	500.5
MODE	#N/A

It scales-up for more complex needs easily, by considering the STATS function call for 1, 301, 601, and 901 in a series:

---

131.      **FOR 1..1000..300 "STATS (1..x) "**

---

Now, *Example 96* is scaled below to a larger range of data.

---

132.      **FOR 100..1000..300**  
**"COUNT (PRIMES (x) ) "**

---

x	TEMP1
100	25
400	78
700	125
1000	168

$z^3$  scales-up to handle larger range (though limited by your computers capacity), as shown in the following computation:

---

133.      **FOR 1...10000000**  
**"COUNT (PRIMES (x) ) "**

---

x	TEMP1
1	0
2	1
4	2
8	4
16	6

32	11
64	18
128	31
256	54
512	97
1024	172
2048	309
4096	564
8192	1028
16384	1900
32768	3512
65536	6542
131072	12251
262144	23000
524288	43390
1048576	82025
2097152	155611
4194304	295947
8388608	564163

Note the “three dots” operator between 1 and 10000000, which signifies the generation of a geometric series.

Please try:

FOR 1...10000000 "COUNT(PRIMES(x))"

as a first attempt to test the capacity of your device.

### 6.a.iii. SET OPERATIONS

Several operations are provided to operate on *sets*. Some examples with set-theoretical operations are given below:

---

134.      **UNION 1..3 4..5**

---

1
2
3
4
5

---

135.      **DIFFERENCE 1..5 1..3**

---

4
5

---

136.      **INTERSECTION 1..5 1..3**

---

1
2
3

## 7. Z<sup>3</sup> SIMPLE EXAMPLES

z<sup>3</sup> language, while being based on global standards, is *unlimited in scope* by being open to extension. It does not take a single approach to problem expression and solution, but *many approaches*, which result in highly flexible possibilities of terse and verbose expressions based on user skill and style.

Several example real-world problems are described in sections below.

### 7.a. Sets and Related Structures

#### 7.a.i. MATRICES

Set (or *sets of sets*) of complex dimensions can represent conventional matrix definitions in an effortless manner. z<sup>3</sup> provides a collection of powerful matrix functions and manipulation capabilities.

##### 7.1.1.1. Matrix Generation

With z<sup>3</sup>, a wide variety of matrices can be generated with ease.

In *Section 4.c (Matrix Operator(| |))*, generation of a simple *set* is described. For example,

137. **|4,2|**

0	0
0	0
0	0
0	0

|4,2| generates a simple 4x2 matrix.

Known types of matrices of required size can be generated using the MATRIX (or MATRIXWITH) function. For example, it is very common to fill a matrix with “positive” or “zero” or “negative” values as needed as follows:

138. **MATRIXWITH(4, "positive")**

42.675436730496585	20.18217903096229	42.50673889182508	9.618869726546109
91.79219712968916	92.56706861779094	79.73511724267155	26.351151429116726
84.20511102303863	17.219695332460105	34.37458740081638	37.503470783121884
1.1728932848200202	10.188973206095397	36.16558900102973	27.639518934302032

139. **MATRIXWITH(4, "negative")**

-42.675436730496585	-20.18217903096229	-42.50673889182508	-9.618869726546109
-91.79219712968916	-92.56706861779094	-79.73511724267155	-26.351151429116726
-84.20511102303863	-17.219695332460105	-34.37458740081638	-37.503470783121884
-1.1728932848200202	-10.188973206095397	-36.16558900102973	-27.639518934302032

140. **MATRIXWITH(4, "zero")**

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

A matrix of size 4 of positive integers is generated below.

141. **MATRIX (4, "positive:integer")**

85	57	100	65
47	100	48	86
18	28	78	92
95	2	35	100

Another example of a matrix of size 4 of negative integers is equally easy.

142. **MATRIX (4, "negative:integer")**

-100	-39	-76	-82
-44	-32	-8	-4
-36	-88	-73	-66
-73	-87	-27	-25

Similarly, a matrix of 4x4 size of integers.

143. **MATRIX (4, "integer")**

-2	-40	-64	-84
88	88	-66	30
66	-16	64	-20
-45	96	88	-87

Similarly, a matrix of 4x4 size of Boolean values (0 or 1).

144. **MATRIX(4, "logical")**

0	0	1	1
0	1	1	0
0	0	1	0
1	0	0	1

145. **MATRIX(4, "alternant",1..10, "[i,j]")**

0 0	0 1	0 2	0 3
1 0	1 1	1 2	1 3
2 0	2 1	2 2	2 3
3 0	3 1	3 2	3 3

146. **MATRIX(4, "alternant",1..10, "i-j")**

0	-1	-2	-3
1	0	-1	-2
2	1	0	-1
3	2	1	0

More special matrices can also be generated as described below.

#### 7.1.1.2. Hilbert Matrix

The Hilbert matrix is a square matrix with entries being the unit fractions.

For example,  $H_{ij} = 1 / i+j-1$ .

So, 2x2 Hilbert matrix is

1	1/2
1/2	1/3

For n, it is a square matrix nxn with the values as

1,	1/2,	1/3,	1/4,	. . .,	1/n
1/2,	1/3,	1/4,	1/5,	. . .,	1/n-1
1/3,	1/4,	1/5,	1/6,	. . .,	1/n-2
. . .					
1/n,	1/n-1,	1/n-2,	. . .,		1/2n-1

In  $z^3$ , simply calling MATRIX function with arguments "hilbert" and size will provide the result:

147. **MATRIX("hilbert",2)**

1	0.5
0.5	0.3333333333333333

148. **MATRIX("hilbert",4)**

1	0.5	0.3333333333333333	0.25
0.5	0.3333333333333333	0.25	0.2
0.3333333333333333	0.25	0.2	0.1666666666666666
0.25	0.2	0.1666666666666666	0.14285714285714285

#### 7.1.1.3. Hermitian Matrix

Hermitian Matrix (or self-adjoint matrix) is a square matrix with complex entries that is equal to its own conjugate transpose, that is, the element in the  $i$ -th row and  $j$ -th column is equal to the complex conjugate of the element in the  $j$ -th row and  $i$ -th column, for all indices  $i$  and  $j$ . In mathematical representation:

$$a_{ij} = \overline{a_{ji}} \text{ or } A^T = \overline{A}$$

Here is an example:

149. **MATRIX("hermitian",3)**

74	79+i39	96+i36
79-i39	52	-20+i48
96-i36	-20-i48	77

#### i. Hankel Matrix

The Hankel matrix (or Catalecticant matrix), named after Hermann Hankel, is a square matrix with constant skew-diagonals (positive sloping diagonals), e.g.

$$A_{i,j} = A_{i-1,j+1}.$$

150. **MATRIX("hankel",3)**

0.5082476452709008	0.8938533218532763	0.8938533218532763
0.8938533218532763	0.8938533218532763	0.5082476452709008
0.8938533218532763	0.5082476452709008	0.17844765487260217

#### 7.a.ii. TOEPLITZ MATRIX

The Toeplitz matrix or diagonal-constant matrix, named after Otto Toeplitz, is a matrix in which each descending diagonal from left to right is constant. The Hankel matrix above is closely related to the Toeplitz matrix (which is an upside-down Hankel matrix).

$$A_{i,j} = A_{i+1,j+1} = a_{i-j}.$$

For instance, the following matrix is a Toeplitz matrix in  $z^3$ :

```
151.    MATRIX("toeplitz",3)
```

```
0.08108029455585108  0.7705726775647403  0.06162740141092149
0.08108029455585108  0.08108029455585108  0.7705726775647403
0.7705726775647403  0.08108029455585108  0.08108029455585108
```

```
152.    MATRIX("toeplitz",4,1..4)
```

```
1  2  3  4
1  1  2  3
2  1  1  2
3  2  1  1
```

#### 7.1.2.1. Hadamard Matrix

Named after the French mathematician Jacques Hadamard, a square matrix whose entries are either +1 or -1, and whose rows are mutually orthogonal, is called a Hadamard Matrix.

In geometric terms, this means that every pair of two different rows in a Hadamard matrix represent two perpendicular vectors.

In combinatorial terms, it means that every pair of rows have matching entries in *exactly* half of their columns and mismatched entries in the remaining columns.

```
153.    MATRIX("hadamard",3)
```

```
1 1 1 1
1 -1 1 -1
1 1 -1 -1
1 -1 -1 1
```

#### 7.1.2.2. Vandermonde Matrix

Vandermonde Matrix, named after Alexandre-Théophile Vandermonde, is a matrix with terms of a geometric progression in each row, i.e., an  $m \times n$  matrix.

$$V = \begin{bmatrix} 1 & \alpha_1 & \alpha_1^2 & \dots & \alpha_1^{n-1} \\ 1 & \alpha_2 & \alpha_2^2 & \dots & \alpha_2^{n-1} \\ 1 & \alpha_3 & \alpha_3^2 & \dots & \alpha_3^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha_m & \alpha_m^2 & \dots & \alpha_m^{n-1} \end{bmatrix}$$

$$V_{i,j} = \alpha_i^{j-1}$$

In  $z^3$ , the following is an example of Vandermonde matrix:

```
154.    MATRIX("vandermonde")
```

```
1  0.098985958378762
   0.009798219956162004
1  0.8950120634399354  0.801046593703011
1  0.9542551881168038  0.9106029640478366
```

```
155.    MATRIX("vandermonde",4,2)
```

```
1  2  4  8
1  2  4  8
1  2  4  8
1  2  4  8
```

```
156.    MATRIX("vandermonde",4,1..4)
```

```
1  1  1  1
1  2  4  8
1  3  9  27
1  4  16  64
```

#### 7.1.2.3. Upper and Lower-Triangular matrix and Symmetric matrix

In Upper Triangular Matrix, all elements under its diagonal are zero. In Lower-Triangular Matrix, all elements over the main diagonal are zeroes. In Symmetric Matrix both sides of the diagonal elements are filled, but with elements around the main diagonal symmetric in value.

```
157.    MATRIX("upper-triangular",6)
```

```
-20  -74  9  66  32  57
0  52  -47  60  26  -87
```

0	0	-28	-31	-49	-70
0	0	0	-18	-70	63
0	0	0	0	59	9
0	0	0	0	0	69

158. **MATRIX("lower-triangular", 6)**

-4	0	0	0	0	0
10	5	0	0	0	0
-40	-82	10	0	0	0
-7	100	99	-74	0	0
34	39	46	17	87	0
68	-4	65	57	0	10

159. **MATRIX("symmetric", 6)**

15	95	-31	30	-5	14
95	-13	98	-35	70	-33
-31	98	-29	48	87	90
30	-35	48	73	16	-72
-5	70	87	16	98	68
14	-33	90	-72	68	97

The hyphen between upper-triangular is optional.

#### 7.1.2.4. *Pascal Matrix*

The elements of the symmetric Pascal matrix are the **binomial coefficients**, i.e.

$$S_{ij} = \binom{n}{r} = \frac{n!}{r!(n-r)!}, \text{ where } n = i + j, \quad r = i.$$

In other words:

$$S_{ij} = {}_{i+j}C_i = \frac{(i+j)!}{(i)!(j)!}.$$

160. **MATRIX("pascal", 5)**

1	1	1	1	1
1	2	3	4	5
1	3	6	10	15
1	4	10	20	35
1	5	15	35	70

#### 7.a.iii. *MATRIX SIZES*

SIZE function can be used to find the size of sets.

161. **SIZE(|4x5|)**

4	5
---	---

A second parameter can be used to indicate the size is to be obtained *in a specific dimension*.

162. **SIZE(|2x3|, 0..1)**

Matrix	Dimension	SIZE						
<table> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>0</td></tr> </table>	0	0	0	0	0	0	0	2
0	0	0						
0	0	0						
<table> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>0</td></tr> </table>	0	0	0	0	0	0	1	3
0	0	0						
0	0	0						

Ther result gives the size along dimensions 0 and 1, of an array of size 2x3.

#### 7.a.iv. *MATRIX OPERATIONS*

The following code is used to generate a 3 by 3 *set* filled with 3. The RANDOM function is then called on each of the members using the \$ function fills each element with random values within 0 through 3. As you can see MX is then assigned with the result.

163. **MX=ARRAY(3,3,3)**

3	3	3
3	3	3
3	3	3

164. **MX=ARRAY(3,3,3).\$(RANDOM)**

0.10359240020625293	0.08560038451105356	0.39424868300557137
0.7169222580268979	2.0910016105044633	0.5476922818925232
0.12873755511827767	0.31074305064976215	2.0048518725670874

The matrix MX is now added along a row using MROWOP using the + operator.

165. **MROWOP(MX,"+",true)**

0.5834414677228779	0.10359240020625293	0.1891927847173065
0.5834414677228779		
3.3556161504238844	0.7169222580268979	2.8079238685313612
3.3556161504238844		
2.4443324783351272	0.12873755511827767	0.4394806057680398
2.4443324783351272		

This result is interesting, as much as it is powerful. The *true* value as the second parameter indicates that the *cumulative* and *running* result as each element is

operated on (as it applies columns in each row) are listed. Only *cumulative* result is provided if the third parameter is empty is *false*.

166. **MROWOP (MX, "+", false)**

0.5834414677228779 3.3556161504238844 2.4443324783351272

The same can be done for each column, with the top row indicating the *cumulative* result, and rows below this row showing results (when each element was added by row to the next element) in the column.

167. **MCOLOP (MX, "+", true)**

0.9492522133514285 2.487345045665279 2.946792837465182  
0.10359240020625293 0.08560038451105356 0.39424868300557137  
0.8205146582331508 2.176601995015517 0.9419409648980945  
0.9492522133514285 2.487345045665279 2.946792837465182

168. **MATRIXPACK (1..5, 2)**

1 2  
3 4  
5

MATRIXPACK splits the matrix into elements of sizes given as parameter. Here it divides matrix MX into pieces of 2 elements.

#### 7.a.v. MATRIX ARITHMETIC OPERATIONS

Simple Matrix addition, multiplications, negation, etc. can be obtained using MATRIX related functions such as below.

169. **MMULT ([1,2,3], [[4], [4], [2]])**

18

Note the use of brackets for the vertical matrix as in [[4],[4],[2]] in *Example 169*.

1..3\*\*3 is a simple notation to duplicate a *set*, by a requested number of times, indicated by \*\*. In this case, 1..3 will be replicated 3 times.

MMULT then operates on the two *sets* as below.

170. **MMULT (1..3\*\*3, 1..3\*\*3)**

6 12 18

6 12 18  
6 12 18

MMULT, does scalar multiplication of the argument is a scalar, with the matrix that is provided in the other argument.

171. **MMULT (1..10, 2)**

2 4 6 8 10 12 14 16 18 20

MATRIXADD conducts simple addition of matrices.

172. **MATRIXADD (1..5, 1..5)**

2 4 6 8 10

MATRIXNEGATE multiplies each element by -1, or effectively negates the elements.

173. **MATRIXNEGATE (ARRAY (4, 4, 10))**

-10 -10 -10 -10  
-10 -10 -10 -10  
-10 -10 -10 -10  
-10 -10 -10 -10

MEQUAL checks each element in a *set* to see if it matches a provided value, in this case a 10x10 matrix filled by 2, is checked against 2.

174. **MEQUAL (ARRAY (10, 10, 2), 2)**

true

#### 7.b. Vector Operations

*Dot Product or Scalar Product* of matrices can be conducted on vectors represented as matrices or *sets* using the DOTPRODUCT (also called SCALARPRODUCT) function.

175. **DOTPRODUCT (1..3, 4..6)**

32

Similarly, *Cross Product or Vector Product* of matrices can be conducted on vectors represented as



matrices or *sets* using the CROSSPRODUCT function.

---

176. **CROSSPRODUCT (1..3,4..6)**

---

-3      6      -3

The functions CROSSPRODUCT and VECTORPRODUCT are the same.

---

177. **VECTORPRODUCT (1..3,4..6)**

---

-3      6      -3

### 7.b.i. MATRIX DETERMINANTS

In linear algebra, the **determinant** is a special value associated with a square matrix.

For example, in a matrix that represents the coefficients of a *System of Linear Equations*, its determinant provides important information about the matrix. The system has a unique solution exactly when the determinant is nonzero; when the determinant is zero there are either no solutions or many solutions.

Determinants occur throughout mathematics. In some cases they are used just as a compact notation for expressions that would otherwise be unwieldy to write down.

For instance, the determinant of the matrix:

---

178. **A = [2 2 1;1 3 4; 2 6 2]**

---

2	2	1
1	3	4
2	6	2

**|A|** has the value as:  
 $(2 \times 3 \times 2 + 2 \times 4 \times 2 + 1 \times 1 \times 6) - (1 \times 3 \times 2 + 2 \times 4 \times 6 + 2 \times 1 \times 2) = -24$

In  $z^3$ , the determinant of a matrix **A** is denoted as **det(A)**, **det A**, or **DET(A)**.

---

179. **det (A)**

---

-24

In  $z^3$ , determinants of any size square matrix is easily calculated, as for the matrix **x** below (generated using deal member function).

---

180. **x=|3|.deal()**

---

0.5169654679484665	0.831032874295488	0.5080022979527712
0.3894023841712624	0.8241141976322979	0.8453823360614479
0.20151550299488008	0.4140409689862281	0.052092665107920766

---

181. **det (x)**

---

-0.036501362405503224

Determinant of an identity matrix is 1, as indicated in *Example 182*.

---

182. **det (IM(4))**

---

1

The determinant of a randomly generated 3x3 matrix is given below.

---

183. **det (|3x3|.deal())**

---

-0.0306124444199811

### 7.c. Matrix Rotations

Matrix rotation can be achieved by the MATRIXROTATE function.

---

184. **MATRIXROTATE (|4|,1)**

---

0	1	0	0
0	0	1	0
0	1	0	0
0	0	1	0

---

185. **MATRIXROTATE (|4|,2)**

---

0	0	1	0
0	1	0	0
0	0	1	0
0	1	0	0

---

186. **MATRIXROTATE (|5|,4)**

---

0	0	0	0	1
---	---	---	---	---

0	1	0	0	0
0	0	1	0	0
0	0	0	1	0
1	0	0	0	0

A simpler  $z^3$  notation for the same is given below, using member functions.

---

187. **|5|.rotate(4)**

---

0	0	0	0	1
0	1	0	0	0
0	0	1	0	0
0	0	0	1	0
1	0	0	0	0

### 7.c.i. SIMPLE MATRIX MERGING WITH FUNCTIONS

---

188. **ARRAY(10,10,10)**  
**.merge(ARRAY(10,10,10))**  
**.\$(RANDOM).print()**

---

Here two 10x10 *sets*, filled with 10 are merged cell by cell and the result is merged together.

To enter lines with soft line-break, use *Shift+Enter*, instead of simple *Enter*.

It makes it simple to provide member functions indented under the main object as given above.

### 7.c.ii. ACROSS MATRICES MERGING WITH FUNCTIONS

In  $z^3$ , *.across()* member function is a powerful operation that applies a *specific function* or *set of functions on each pair of elements* on the two input matrices.

Consider a simple 3x3 identity matrix.

---

189. **IM(3)**

---

1	0	0
0	1	0
0	0	1

The *.across()* function applies SUM to the two identity matrices.

---

190. **IM(3).across(IM(3),SUM)**

---

2	1	1	1	0	0	1	0	0
1	2	1	0	1	0	0	1	0
1	1	2	0	0	1	0	0	1
1	0	0	2	1	1	1	0	0
0	1	0	1	2	1	0	1	0
0	0	1	1	1	2	0	0	1
1	0	0	1	0	0	2	1	1
0	1	0	0	1	0	1	2	1
0	0	1	0	0	1	1	1	2

A more powerful example of applying a *set* of functions (SUM and AVG in the following case) on the each combination of elements of input matrices is shown next.

---

191. **IM(3)**  
**.across(**  
**IM(3),**  
**[SUM,AVG]**  
**)**

---

2	1	10.5	10.5	10.5	0	0	0	0	10.5	0	0	0	0
10.5	2	1	10.5	0	0	10.5	0	0	0	0	10.5	0	0
10.5	10.5	2	1	0	0	0	0	10.5	0	0	0	0	10.5
10.5	0	0	0	0	2	1	10.5	10.5	10.5	0	0	0	0
0	0	10.5	0	0	10.5	2	1	10.5	0	0	10.5	0	0
0	0	0	0	10.5	10.5	10.5	2	1	0	0	0	0	10.5
10.5	0	0	0	0	10.5	0	0	0	0	2	1	10.5	10.5
0	0	10.5	0	0	0	0	10.5	0	0	10.5	2	1	10.5
0	0	0	0	10.5	0	0	0	0	10.5	10.5	10.5	2	1

### 7.c.iii. QUICK MULTIPLICATION TABLES

Consider the following use of the across function to generate a Multiplication Table in Command 192.

In this case, the *.across* function is used in such a way as to operate on every pair of numbers between 1 and 10. This is a simple demonstration of the *.across* operation, (which can also be used on an array of functions as indicated previously) to create useful

collections of results, such as those that can be used for contour plotting etc.

192.

```
1..10.across(1..10,PRODUCT).transpose()
```

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

Using the power of *sets*, you can get the multiplication factors of any one number (using the [ ] element accessing capability). For example, the following command gets the multiplication table for the number 6 (note the 0 based indexing of *sets* as used).

193.

```
1..10
```

```
.across(1..10,PRODUCT)
```

```
.transpose()[0][5]
```

```
6
12
18
24
30
36
42
48
54
60
```

## 7.d. Puzzles and Other Interesting Computations

There are several special problems and puzzles that are pre-solved in z<sup>3</sup> for enthusiasts, to analyze various case scenarios and to slice and dice such results.

### 7.d.i. MAGIC SQUARE

In recreational mathematics, a Magic Square is an arrangement of numbers (usually integers) on a square grid, where the numbers in each row, the numbers in each column, and the numbers in the forward and backward main diagonals, all add up to the *same number*.

A magic square has the same number of rows as it has columns, and in conventional math notation, "n" stands for the number of rows (and columns) it has.

Thus, a magic square always contains  $n^2$  numbers, and its size (the number of rows and columns it has) is described as being "of the order n".

The smallest nontrivial case, a  $3 \times 3$  grid that is a magic square of order 3 is shown next.

194.

**MAGICSQUARE (3)**

```
2      7      6
9      5      1
4      3      8
```

In z<sup>3</sup>, magic squares of any size can be created using the **MAGICSQUARE** function. The parameter can be given as a series, such as **MAGICSQUARE(3..15)** to have a series of **MAGICSQUAREs** of sizes 3 to 15.

195.

**MAGICSQUARE (4)**

```
1      15      14      4
12     6       7       9
8      10      11      5
13     3       2      16
```

The sum of every row, column and diagonal, which should be a constant value, is called the *Magic Constant* or *Magic Sum*, *M*. Every normal magic square has a unique constant determined solely by the value of *n*, which can be calculated using this formula:

$$M = \frac{n(n^2 + 1)}{2}.$$

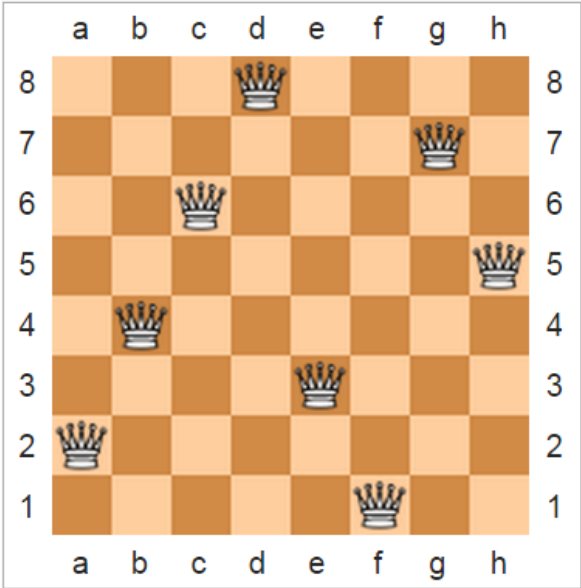
The row wise and column wise sum of the magic square elements show that for an m of 4 rows and columns,  $4*(4^2+1)/2 = 34$  would be the magic constant. *Commands 196 and 197* indicate that row-wise and column-wise summations of elements add up to a Magic Constant of 34. The `.$$` and `$$$` member functions of *sets* are used to compute the row and column based sums.

196.	MAGICSQUARE (4) . \$\$ (SUM)
34	
34	
34	
34	

197.	MAGICSQUARE (4) . \$\$\$ (SUM)
34	
34	
34	
34	

7.d.ii. N-QUEENS PUZZLE

Chess composer Max Bezzel published the *Eight Queens Puzzle* in 1848. Franz Nauck published the first solutions in 1850, and also extended the puzzle to the *n*-queens problem, with *n* queens on a chessboard of *n*×*n* squares. Since then, many mathematicians including Carl Friedrich Gauss



have worked on both the eight queens puzzle and its generalized *n*-queens version.

**Eight queens puzzle** is the problem of placing eight **chess queens** on an 8×8 chessboard so that no two queens attack each other. Thus, a solution requires that no two queens share the same row, column, or diagonal.

The eight queens puzzle is an example of the more general ***n*-queens problem** of placing *n* queens on an *n*×*n* chessboard, where solutions exist for all natural numbers *n* with the exception of *n*=2 and *n*=3.

In computer programming, the solution to this problem is considered a classic involving problem-solving, algorithms, and data structures. This approach has several applications in scheduling, distributed systems, networking, etc.

There are two explicit solutions for *n*=4 and the solutions can be obtained in *z*<sup>3</sup> as follows:

198.	NQUEENS (4)
1	_ Q _ _
	_ _ Q _
	Q _ _ _
	_ _ Q _
2	_ _ Q _
	Q _ _ _
	_ _ Q _
	_ Q _ _

In *z*<sup>3</sup>, the user can use the NQUEEN function in several ways to get the solution, as well as explore and discover patterns associated with these solutions.

199.	NQUEENS ()
1	Q _ _ _ _ _
	_ _ _ Q _ _
	_ _ _ _ _ Q
	_ _ _ _ Q _

The result will be 92 solutions to *8Queens* problem, the first and last are given in the following:

```

|_|_|Q|_|_|_|_|
|_|_|_|_|_|Q|_|
|_|Q|_|_|_|_|_|
|_|_|_|Q|_|_|_|
.
.
.
.
.
.
92
|_|_|_|_|_|_|Q|
|_|_|_|Q|_|_|_|
|Q|_|_|_|_|_|_|
|_|_|Q|_|_|_|_|
|_|_|_|_|_|Q|_|
|_|Q|_|_|_|_|_|
|_|_|_|_|_|Q|_|
|_|_|_|_|_|Q|_|
|_|_|_|_|Q|_|_|

```

More interestingly, you can compute and get in a range of data values for different sizes as follows:

---

200. **NQUEENS (1..8)**

---

It scales-up easily, and consider sizes of chessboards with sizes 8..12.

---

201. **NQUEENS (8..12)**

---

The results of *Example 201* range from 352 solutions for 8 queens upto 14200 Solutions for 12 queens. In simpler cases, solutions could be sparse as follows.

---

202. **1..4@NQUEENS**

---

```

1
1
|Q|

1 Solutions
2
0 Solutions
3
0 Solutions
4
1
|_|Q|_|_|
|_|_|_|Q|
|Q|_|_|_|
|_|_|Q|_|

2
|_|_|Q|_|

```

```

|Q|_|_|_|
|_|_|_|Q|
|_|Q|_|_|

```

2 Solutions

For practical purposes, the results become too large when it gets to a size of 19 queens.

7.d.iii. **BIRTHDAY PROBABILITY**

In probability theory, birthday probability is a simple yet interesting problem. The history of this problem is obscure. Possibly, Harold Davenport or Richard von Mises proposed what we consider today to be the *birthday problem*.

This problem solves the probability of at least two of the  $n$  people in a room sharing a birthday. In a group of  $n$  people, there are  $365n$  possible combinations of birthdays.

The simplest solution is to determine the probability of no matching birthdays and then subtract this probability from 1.

When  $n \leq 365$ :

$$p(n) = 1 - \left(\frac{364}{365}\right)^{C(n,2)} = 1 - \left(\frac{364}{365}\right)^{n(n-1)/2}$$

In a random group of 23 people, there is actually about a 50–50 chance that two of them will have the same birthday. Sample solutions for the following problems in  $z^3$  are as follows:

---

203. **BIRTHDAYPROBABILITY (23, 365)**

---

0.5131345029080766

This is known as the birthday paradox. In a room of 75 there's a 99.9% chance of two people birthday matching.

In a group of 6 people, the following gives how many of them celebrate their birthday in the same month?

---

204. **BIRTHDAYPROBABILITY (6, 12)**

---

0.7745997705740058

The decimal value is equivalent percentage value of 78%.

In a list of 40 people, it is 88%.

#### 205. BIRTHDAYPROBABILITY (40)

0.8866177230044445

#### 206. BIRTHDAYPROBABILITY (10..100..10)

Count	BIRTHDAYPROBABILITY
10	0.12721138320197134
20	0.41972174869639556
30	0.7061121304328839
40	0.8866177230044445
50	0.9666783241671075
60	0.9925402093017472
70	0.9987278316204606
80	0.9998347350723671
90	0.9999836455875729
100	0.9999987671582301

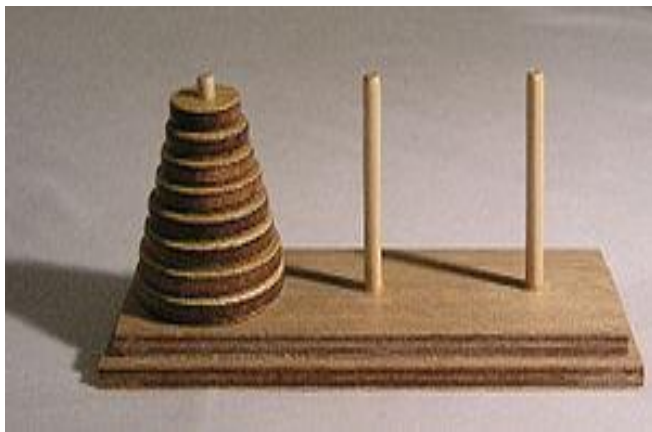
#### 7.d.iv. TOWERS OF HANOI

The Tower of Hanoi puzzle was invented by the French mathematician Edouard Lucas in 1883. This puzzle is also known as *Towers of Brahma*.

The Tower of Hanoi problem is isomorphic to finding a Hamiltonian path on an  $n$ -hypercube.

Suppose three rods, as shown in the figure, and several disks with different sizes which can slide onto any rod.

The disks are arranged in ascending order of size on one rod, with the smallest one at the top in a stack, such that it makes a conical shape.



The objective of the game is to move all the disks onto a different pole with the following conditions:

- Only one disk can be moved at a time
- Only the uppermost disk can be moved from any stack.
- The smaller disk should always occupy the upper position of each stack at all times.

The puzzle can be solved in seven moves for three disks. The minimum number of moves required to solve a Tower of Hanoi puzzle is  $2^n - 1$ , where  $n$  is the number of disk.

Hence the number of steps can be expected to increase rapidly, with the number of disks. Using  $z^3$ , number of steps to can be calculated for varying  $n$ :

#### 207. 1..100..5@"2^n-1"

n	$2^n - 1$
1	1
6	63
11	2047
16	65535
21	2097151
26	67108863
31	2147483647
36	68719476735
41	2199023255551
46	70368744177663
51	2251799813685247
56	72057594037927940
61	2305843009213694000
66	73786976294838210000
71	2.3611832414348226e+21
76	7.555786372591432e+22
81	2.4178516392292583e+24
86	7.737125245533627e+25
91	2.4758800785707605e+27
96	7.922816251426434e+28

When  $n=3$ , we expect 7 steps.

#### 208. 3@"2^n-1"

n	TEMP1
3	7



4	16
5	32
6	64
7	128
8	256
9	512
10	1024

7.d.v. FLOYDS TRIANGLE

Floyd’s triangle is the collection of natural numbers arranged in a right triangle to the left. It is named after Robert Floyd. Each line in Floyd’s triangle has one more element than the previous row, and has consecutive numbers from the left in each row. The nth row in the Floyd Triangle sums to  $n(n^2 + 1)/2$ , same as the constant of an  $n \times n$  magic square.

In z^3, Floyd’s triangle displays as follows, with the first parameter indicating the number of rows to be displayed, and the second number indicating the limit of the natural number to be displayed.

213. FLOYDSTRIANGLE (20 , 34)

1									
2	3								
4	5	6							
7	8	9	10						
11	12	13	14	15					
16	17	18	19	20	21				
22	23	24	25	26	27	28			
29	30	31	32	33	34				

214. FLOYDSTRIANGLE (1 . . 3)

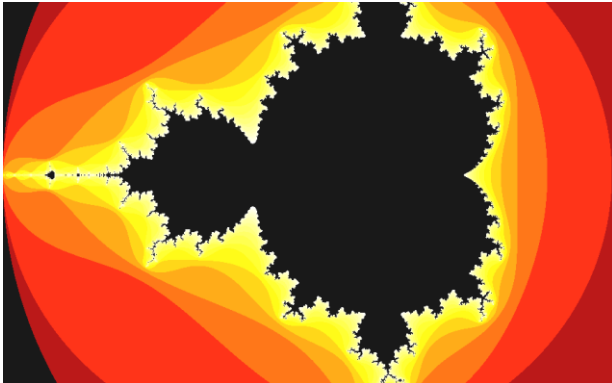
NumberOfRows	FLOYDSTRIANGLE
1	1
2	1 2 3
3	1 2 3 4 5 6

7.d.vi. FRACTALS-MANDELBROT

Compared to the Euclidean geometry, which has a long history of more than 2000 years, *Fractal* geometry is very new. Benoit Mandelbrot's famous book *The Fractal Geometry of Nature* was published relatively recently, in 1982. Nature is full of fractals, like trees, river networks, lightning bolts and blood vessels etc. Hence, fractal patterns tend to look extremely familiar and *natural*.

Fractals are *infinitely complex patterns that are self-similar across different scales*. This property is called “self-similarity”. Fractals form a *never ending pattern*, created by repeating a simple process over and over, in an ongoing feedback loop.

215. FRACTAL (20)



In z^3, user can get varying accuracy of fractals, by setting the parameter of the FRACTAL call.

The following call will create FRACTAL diagrams for an accuracy of 10, 100 and 1000. The quality of

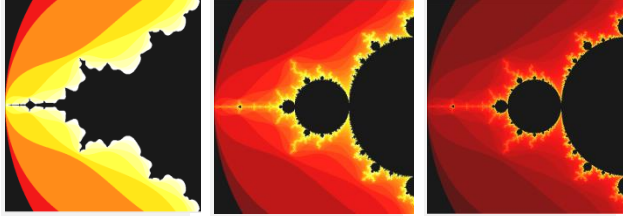


the generated fractal improves with the accuracy used, as evident from the generated examples given below.

---

216.      `[ [10,100,1000] ] @FRACTAL`

---



Mandelbrot Set is the set of points in the complex plane with the sequence  $(c, c^2 + c, (c^2 + c)^2 + c, ((c^2 + c)^2 + c)^2 + c, (((c^2 + c)^2 + c)^2 + c)^2 + c, \dots)$ , where the result does not approach infinity. The Julia Set is closely related to Mandelbrot Set.

The Mandelbrot Set is obtained from the quadratic recurrence equation:

$$z_{n+1} = z_n^2 + c, \text{ (with } z_0 = 0),$$

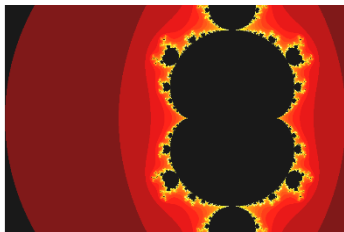
where points  $c$  in the complex plane for which the computed value of  $z_n$  does not tend to infinity.

The colors represent points that remain bounded within a limit for such recursive calls.

---

217.      `FDZ3 ()`

---



FDZ3 gives the FRACTAL generated for  $z_{n+1} = z_n^3 + c$ .

#### 7.d.vii. LISSAJOUS

*Lissajous Curve* is a parametric plot of the harmonic system. It is also called *Bowditch Curves*. This family of curves was investigated by Nathaniel Bowditch, an American mathematician in 1815, and later in more detail by Jules Antoine Lissajous in 1857.

Lissajous used sounds of different frequencies to vibrate a mirror. A beam of light reflected from the mirror, was allowed to trace patterns which depended on the frequencies of the sounds – in a setup similar to projectors used in today's laser light shows.

Lissajous figures often appeared as props in science fiction movies made during the 1950's. It has serious applications in physics, astronomy, and other sciences today.

Technically, Lissajous figure is the intersection of two sinusoidal curves, the axes of which are at right angles to each other. Mathematically, this translates to a Complex harmonic function:

$$x = A \sin(at + \delta), \quad y = B \sin(bt),$$

The appearance of a figure is highly sensitive to  $a/b$ , the ratio of  $a$  and  $b$ .

According to the ratio value, the shapes of the figures change in interesting ways.

For a  $a/b$  ratio=1, the figure is an ellipse.

For  $a=b$ ,  $\delta = \pi/2$  radians, the figure is a circle.

For  $\delta = 0$ , the figure is a line.

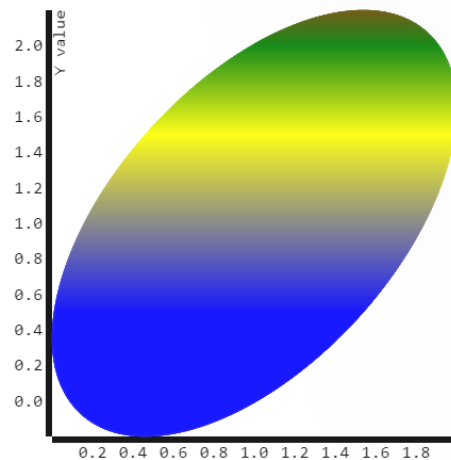
For  $a/b = 2$ ,  $\delta = \pi/4$ , the result is a parabola.

The Lissajous curve gets more complicated for other ratios, which are closed only if  $a/b$  is rational.

---

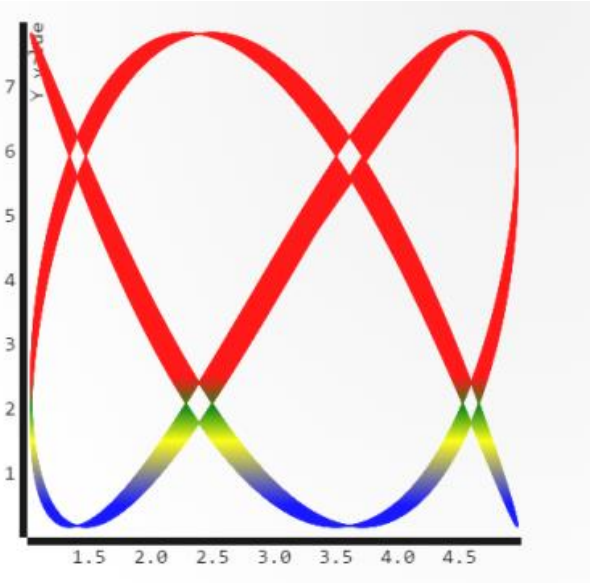
218.      `LISSAJOUSCURVE ("ellipse")`

---



LISSAJOUSCURVE function can be given upto 8 parameters packed in a *set* indicating a1, b1, c1, d1, a2, b2, c2 and d2, or a string like "ellipse", "parabola" etc.

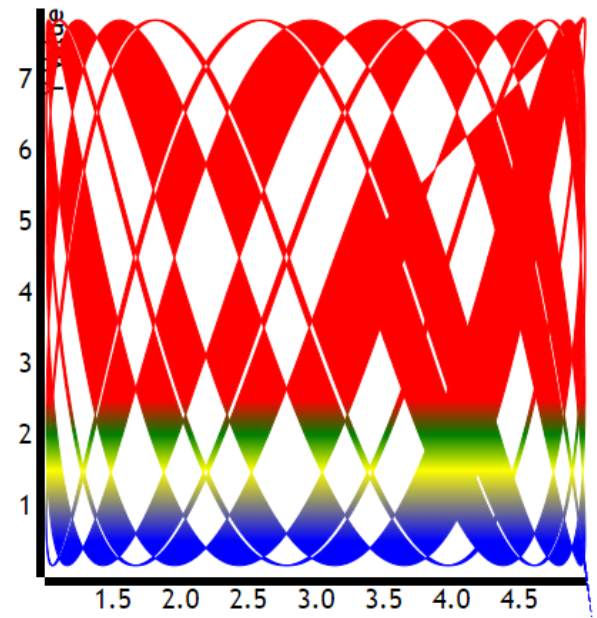
219. **LISSAJOUSCURVE ([2,3,3,3,4,5,3,4])**



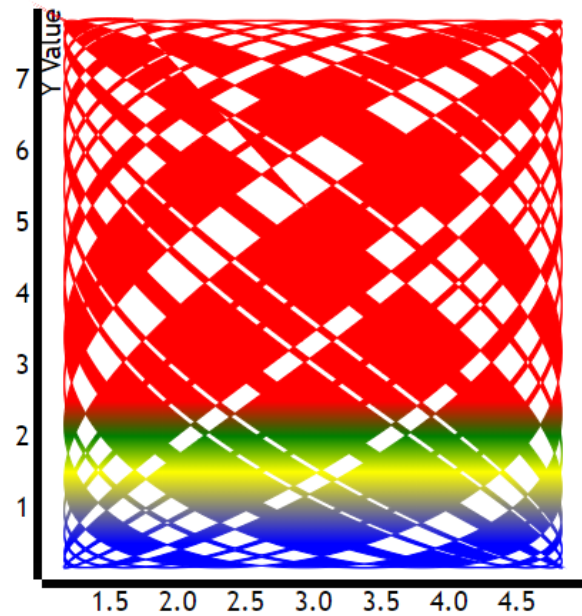
With the interactive parameter changing technique, right click on the parameter that you want to change on the ZOS display lines.

Use the range controls that appear to generate a variety of such curves as given below.

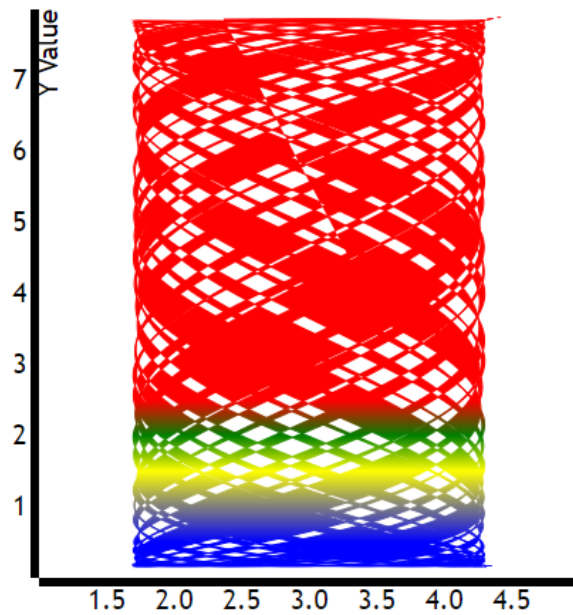
220. **LISSAJOUSCURVE ([2,61,3,3,4,5,3,4])**



221. **LISSAJOUSCURVE ([2,70,3,3,4,5,3,4])**

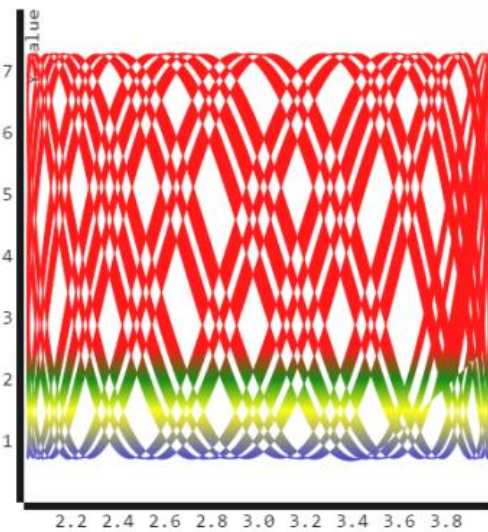


222. **LISSAJOUSCURVE ([2,79,3,3,4,5,3,4])**



LISSAJOUS Curves are fascinating regarding the types of curves you can generate by simply changing the parameters.

```
223.    LISSAJOUSCURVE ([1,-3,32,3,-
4,52,-4,4])
```

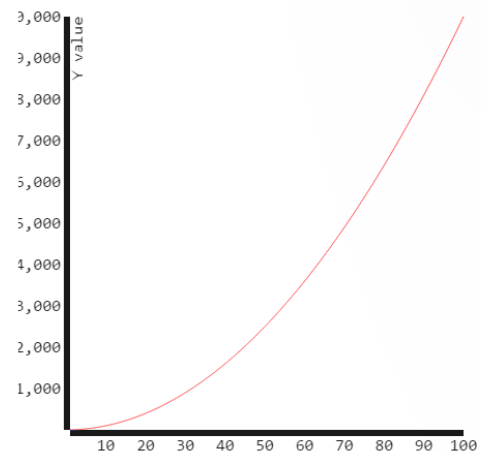


### 7.d.viii. GRAPHING DATA CURVE

A plot is used for representing a dataset graphically - usually showing the relationship between two or more variables.

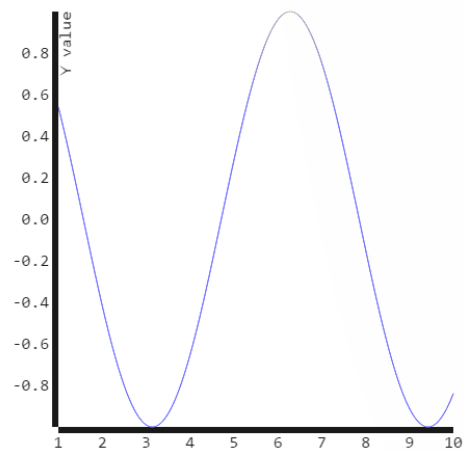
Graphs of functions are used in mathematics, sciences, engineering, technology, finance, and many other areas.

```
224.    1..100@"x^2" .graph()
```



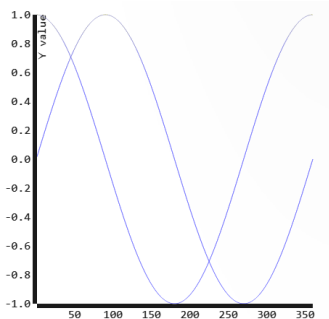
Two axes are used to plot a graph. The horizontal axis is called the x-axis and the vertical one is the y-axis.

```
225.    1..10..0.008@COS .graph()
```



In  $z^3$ , user can also use multiple functions to plot a graph showing values from both:

226. `1..360@[DSIN,DCOS] .graph()`



## 7.e. Financial Functions

A variety of functions are available in  $z^3$ . The functions generally follow the functions available in Spreadsheets, Databases, etc.

For example, PMT function finds the payment per period for a loan, based on constant payments and a constant interest rate.

The general syntax for this is:

`PMT(Rate,NoPaymentPeriods,PresentValue,FutureValue, Type)`

The Type parameter can be 0 or 1, to indicate the payment at the start or end of the period.

227. `PMT(0.132/12,15,50000,100000,1)`

-9695.799429757533

228. `PMT(0.132/12,15,50000,100000,0)`

-9802.453223484865

Now, we can really rev things up to analyze a variety of inputs with a single command. 15..60.5 will effectively find all combinations of arguments, with the NoPaymentPeriods for 15, 20, 25, 30, 35, 40, 45, 50, 55 and 60.

229. `PMT(0.132/12,15..60..5,50000,100000,0)`

Rate	NoPaymentPeriods	PresentValue	FutureValue	Type	PMT
0.011000000000000001	15	50000	100000	0	-9802.453223484865
0.011000000000000001	20	50000	100000	0	-7296.235646349717
0.011000000000000001	25	50000	100000	0	-5795.499074576284
0.011000000000000001	30	50000	100000	0	-4797.4963045857485
0.011000000000000001	35	50000	100000	0	-4086.7629976833496
0.011000000000000001	40	50000	100000	0	-3555.5659024443326
0.011000000000000001	45	50000	100000	0	-3144.0522641536077
0.011000000000000001	50	50000	100000	0	-2816.309590311666
0.011000000000000001	55	50000	100000	0	-2549.483727385898
0.011000000000000001	60	50000	100000	0	-2328.337900528343

Similarly, the future value of an annuity can be calculated using the FV function, given the rate, number of payment periods, payment per period, optional present value and a type flag indicating the payment to be done at the start or end of the period.

230. `FV(0.05/12,14,-1500,1)`

21577.278753101153

On the other hand, PV function gives the present value, for an optional future value. This can easily be followed by changing the parameters to try out different scenarios, such as changing the type parameter.

231. `PV(0.07/12,20*12,2500,0)`

-322456.2662406346

232. `PV(0.07/12,20*12,2500,0..1)`

Rate	NoPaymentPeriods	Payment	FutureValue	Type	PV
0.005833333333333334	240	2500	0	-322456.2662406346	
0.005833333333333334	240	2500	1	-322456.51384268	

To find the number of days of in the coupon period that contains the settlement date, use the COUPDAYS function.

233. `COUPDAYS`  
`(DATE(2012,1,1),DATE(2013,1,1),1,1)`

366

COUPDAYBS gives the number of days from the beginning of a coupon period until its settlement date.

234. **COUPDAYBS (DATE (2008, 6, 1), DATE (2009, 1, 1), 2, 1)**

152

COUPDAYSNCR gives the number of days from the settlement date to the next coupon date..

235. **COUPDAYSNCR (**  
**DATE (2012, 1, 6) ,**  
**DATE (2013, 6, 6) ,**  
**1, 1**  
**)**

152

Several functions also available in z<sup>3</sup> to calculate financial values, of which some are indicated below.

- XIRR - To calculate internal rate of return.
- IRR - To calculate the internal rate of return of a cash flow stream associated with an investment.
- MIRR - To find the value of the modified internal rate of return for a particular cash flows.
- XNPV - To find the net present value for a schedule of cash flows .
- NPV - To calculate the net present value of an investment.
- SYD - To find the depreciation of an asset for a given time period .
- EFFECT - To calculate the effective annual interest rate.

Combination of these financial functions with combinatorial arguments, provides the ability to do flexible analysis for a variety of inputs as below.

The example shows the possibility of analysis by varying the parameters over ranges of dates, frequencies, etc.

This can be a powerful analytical or teaching tool in professional and educational settings.

236. **COUPDAYSNCR (**  
**DATE (2008, 1, 12, 1) ,**

**DATE (2009, 11, 1) ,**  
**1, 2, 1**

)

Listing of dates in a series can be achieved by the # operator for easy use in date and financial functions.

Settlement	Maturity	Frequency	Basis	COUPDAYSNCR
1/1/2008	11/1/2009	1	1	305
1/1/2008	11/1/2009	2	1	121
2/1/2008	11/1/2009	1	1	274
2/1/2008	11/1/2009	2	1	90
3/1/2008	11/1/2009	1	1	245
3/1/2008	11/1/2009	2	1	61
4/1/2008	11/1/2009	1	1	214
4/1/2008	11/1/2009	2	1	30
5/1/2008	11/1/2009	1	1	184
5/1/2008	11/1/2009	2	1	184
6/1/2008	11/1/2009	1	1	153
6/1/2008	11/1/2009	2	1	153
7/1/2008	11/1/2009	1	1	123
7/1/2008	11/1/2009	2	1	123
8/1/2008	11/1/2009	1	1	92
8/1/2008	11/1/2009	2	1	92
9/1/2008	11/1/2009	1	1	61
9/1/2008	11/1/2009	2	1	61
10/1/2008	11/1/2009	1	1	31
10/1/2008	11/1/2009	2	1	31
11/1/2008	11/1/2009	1	1	365
11/1/2008	11/1/2009	2	1	181
12/1/2008	11/1/2009	1	1	335
12/1/2008	11/1/2009	2	1	151

For example, (#2/2/2012-2/12/2012) will give this series of dates as in the following example.

237. **(#2/2/2012-2/12/2012)@DAYSINYEARTILLDATE**

Date	DAYSINYEARTILLDATE
2/2/12	33day
2/3/12	34day
2/4/12	35day
2/5/12	36day
2/6/12	37day

2/7/12	38day
2/8/12	39day
2/9/12	40day
2/10/12	41day
2/11/12	42day
2/12/12	43day

## 7.f. Statistical Functions:

In mathematics, statistics is the body of science that deals with the analysis, interpretation, presentation and organization of data. z^3 provides a large collection of statistical functions that perform most of the common Statistical calculations from simple min, max, mean, median and mode calculations to more complex Statistical Distribution and Probability tests, Distributions, Frequency, Rank, Deviation, Variance, Trend Lines, etc.

**238. MAX(-10..0,13..25)**

25

**239. MIN(-29..10)**

-29

**240. AVEDEV(-15..22)**

9.5

**241. VARA(10,15,20,25,false)**

92.5

**242. CORREL([(-10)..(-3)], [20..28])**

0.8366600265340756

**243. FISHER(0.1..0.4..0.1)**

0.10033534773107562    0.2027325540540821  
0.3095196042031118    0.42364893019360184

**244. KURT([-40..30,35..60..0.7])**

-1.1999999999999988

**245. WEIBULL(143,180,170,true)**

3.019806626980426e-14

Using z^3, user can conduct a variety of Statistical Tests.

These include:

ANOVA SINGLE FACTOR  
BARTLETT'S TEST  
CHITEST  
COCHRAN'S Q TEST  
DURBIN WATSON TEST  
FTEST  
FRIEDMANN TEST  
KENDALL'S TAU TEST  
KRUSKALWALLI'S TEST  
KSTESTCORE  
KSTESTEXPONENTIAL  
KSTESTNORMAL  
LEVENE'S TEST  
MANNWHITNEY U TEST  
MOODSMEDIAN TEST  
RIEMANN ZETA TEST  
SHAPIRO-WILK TEST  
SIGN TEST  
SPEARMAN'S RHO TEST  
TTEST  
TTEST PAIRED  
TTEST TWO SAMPLES EQUALVARIANCES  
TTEST TWO SAMPLES UNEQUALVARIANCES  
WILCOXON RANK SUM TEST  
WILCOXON SIGNEDRANK TEST  
ZTEST  
ZTEST TWO SAMPLE FOR MEANS

More info is given at <http://wiki.zcubes.com>

## 8. APPENDICES

### 8.a. Appendix I Operators

The following operators are used in the  $z^3$  language:

<code>+, -, *, /, ^, %</code>	Arithmetic Operators
<code>   </code>	Array Function and Creation Operator
<code>.., ...</code>	Arithmetic and Geometric Series Creation
<code>@</code>	Apply to
<code>#</code>	Series or Special Case Qualifier for Dates, Calci Cells, and Sequences, etc.
<code>&lt;&lt;&lt;</code>	Member or Variable Assignment
<code>()</code>	Function Call
<code>[]</code>	Set Creation
<code>{ }</code>	Object Set
<code>["key"]</code>	Set Object Membership
<code>.</code>	Member Function Dereferencing.
<code>.mf</code>	Member Function
<code>\$(function, parameters)</code>	Element-wise Function Application

<code>.\$\$ (function, parameters)</code>	Row-wise Function Application
<code>.\$\$\$ (function, parameters)</code>	Column-wise Function Application
<code>.\$_ (function, parameters)</code>	Cumulative Function Application (all)

### 8.b. Appendix II: Simple Set and Objects

#### 8.b.i. SET

Simple *Set* are declared using the conventional `[]` notation, and items are then accessed using the common `[index]` notation.

#### 8.b.ii. ASSOCIATIVE SET/OBJECTS

In  $z^3$ , *associative set* are distinct from *simple set*, in both declaration and qualification syntax.

*Associative set* can be declared using `{ }` notation, with members within indicated using `(:)` as the separator between the member id and the value. While accessing such a member, conventional operator `(.)` is used to reference it.

In the following example all these characteristics are demonstrated elegantly as a data and a function are associated.

```

246.
A1={"a":8,"g":function(x){return(x+5)}}

{
  "a": 8
}

247.
A1.g(5)

10

```



## 8.c. Appendix III: Javascript and z<sup>3</sup>

z<sup>3</sup> uses Javascript as a backbone and provides full functionality and power of Javascript to pass through.

### 8.c.i. USING SET MEMBER FUNCTIONS

For example, any *set*, including matrices, can be operated on using any of the *set member functions*.

This can be triggered even with simple javascript commands as follows. Note the repeated application of additional member functions. In the following, *.seq()* fills a *set* with a sequence, and *.explode()* expands an element into a *set* having elements from 1 to that element.

```
248.      new
Array(8).seq().explode(1).print()
```

```
[
  [
    ],
    [1
    ],
    [1,2
    ],
    [1,2,3
    ],
    [1,2,3,4
    ],
    [1,2,3,4,5
    ],
    [1,2,3,4,5,6
    ],
    [1,2,3,4,5,6,7
    ]
]
```

```
249.      new
Array(8).seq(2).explode(1).print()
```

```
[
  [1,2
  ],
  [1,2,3
  ],
  [1,2,3,4
  ],
  [1,2,3,4,5
  ],
  [1,2,3,4,5,6
  ],
  [1,2,3,4,5,6,7
  ],
  [1,2,3,4,5,6,7,8
  ],
]
```

```
[1,2,3,4,5,6,7,8,9
]
```

```
250.      new Array(8).seq().print()
```

```
[0,1,2,3,4,5,6,7]
```

```
251.      new Array(8).seq(2).print()
```

```
[2,3,4,5,6,7,8,9]
```

```
252.      new Array(8).seq(2,2).print()
```

```
[2,4,6,8,10,12,14,16]
```

```
253.      new
Array(8).seq(2,2).explode(2).print()
```

```
[
  [2,3
  ],
  [2,3,4,5
  ],
  [2,3,4,5,6,7
  ],
  [2,3,4,5,6,7,8,9
  ],
  [2,3,4,5,6,7,8,9,10,11
  ],
  [2,3,4,5,6,7,8,9,10,11,12,13
  ],
  [2,3,4,5,6,7,8,9,10,11,12,13,14,15
  ],
  [2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17
  ]
]
```

An identity matrix of size 5 is generated, filled with random numbers and then cleared and filled with 3, and is printed below.

```
254.      IM(5).deal().clear(3).print()
```

```
[
  [3,3,3,3,3
  ],
  [3,3,3,3,3
  ],
  [3,3,3,3,3
  ],
  [3,3,3,3,3
  ],
  [3,3,3,3,3
  ]
]
```



```
radpiy10@[SIN,COS] .graph()
```

## 8.d. Appendix IV Series Generation

Creating series of items to be the input for further processing is a powerful aspect of z<sup>3</sup>.

These can range from arithmetic series, geometric series, other prepacked series, alphabet series, and date series.

### 8.d.i. ARITHMETIC SERIES

<Start>..<>end>..<>interval>

notation is adopted for arithmetic series.

e.g.:

```
1..100..2
```

### 8.d.ii. GEOMETRIC SERIES

<Start>...>end>...>index>

notation is adopted for arithmetic series.

e.g.:

```
1...100...2
```

### 8.d.iii. PREPACKED SERIES

Several standard situations like angles in radians of degrees are indicated as below.

The terms can be composed using the pattern, (the numbers can be changed as you need), in a self-evident manner:

```
rad360by4
deg180by3
deg360by4@SIN
radpi/2by3
radpiy13
radpiy3
radpiy2
deg180by5
deg360by1@SIN
deg360by180@SIN
deg360by180@SIN
radPiby13@LOG
rad16*Piby16
```

e.g.:

```
SIN(radpiy2)
```

### 8.d.iv. DATE SERIES

A series of dates can be generated using # symbol followed by the data range. #<Start>->End > Dates.

e.g.:

```
#1/1/2001-1/31/2013
```

### 8.d.v. ALPHABET SERIES

A series of letters can be generated using # symbol followed by the letter range. #<Start>->End Letters.

e.g.:

```
#a-z
#A-D
```

## 8.e. Appendix V Member Functions

The following is a listing of member functions for *sets*, *strings*, *functions*, *objects* and *other key objects*. By convention, *member functions* follow lowercase notation in general, while *primary functions* follow uppercase notation. The intellisense capability on the command-line assists users with listing of functions and arguments, simply by pressing *Ctrl+Space* using the keyboard. Set member functions are given under the Array object in the following table, in an interchangeable manner.

Object	Member Function	Parameters
String	endsWith	Str
Array	PAD	
Array	headingset	Array
Array	some	fun /*, thisp*/
Array	shuffle	
Array	map	callback, thisArg
Array	IntArrayToString	
Array	ZCompareArrays	Arr
Array	ZMap	Fnc
Array	ZFoldRight	fnc, start
Array	ZFoldLeft	fnc, start

Array	ZExistsObject	x
Array	ZFilter	fnc
Array	ZRandomElement	
Array	IsArray	
Array	ZMapR	fnc
Array	table	
Array	calci	
Array	transpose	IncludeHeader
Array	column	
Array	columns	
Array	row	
Array	rows	
Array	cell	Row, Column, Width, Height
Array	cells	
Array	accumulate	Total
Array	cumcolumns	
Array	cumrows	
Array	cumcolumn	
Array	cumrow	
Array	accumulatewith	CumulateFunction, CurrentResult
Array	cumcolumnswith	
Array	cumrowswith	
Array	cumcolumnwith	
Array	cumrowwith	
Array	stringlist	
Array	tofunctions	
Array	setHeadings	Headings
Array	headings	
Array	istype	
Array	t	
Array	copy	
Array	extract	
Array	zip	OtherArray
Array		
Array	unzip	
Array	zero	ValueInstead, PreserveStructure
Array	random	Base,Numbers
Array	rand	
Array	pad	Length, PadString
Array	dim	

Array	seq	StartIndex, By
Array	explode	StartIndex, By, RecurseTillLevel
Array	implode	RecurseTillLevel
Array	unimplode	RecurseTillLevel
Array	specialprint	Trimmed,TabLevel
Array	print	Trimmed,TabLevel
Array	cartesianproduct	IsWithoutFlatten
Array	make1to2d	ReplaceOriginal
Array	twod	
Array	is2d	
Array	is1d	
Array	rowpush	OtherArray
Array	count	FirstLevelOnly
Array	mergecolumns	OtherArray
Array	clone	
Array	rowconcat	OtherArray
Array	mergerows	
Array	colconcat	
Array	columnconcat	
Array	rowlengths	Function
Array	deal	Within, ManyInEach
Array	nth	Nth, Count
Array	first	Count
Array	first	
Array	second	
Array	third	
Array	fourth	
Array	fifth	
Array	sixth	
Array	seventh	
Array	eighth	
Array	ninth	
Array	tenth	
Array	eleventh	
Array	twelfth	
Array	thirteenth	
Array	fourteenth	
Array	fifteenth	
Array	sixteenth	
Array	seventeenth	
Array	eighteenth	

Array	nineteenth	
Array	twentieth	
Array	hundredth	
Array	thousandth	
Array	millionth	
Array	last	Count
Array	lastelement	Count
Array	firstelement	Count
Array	pastefolds	MidOnly
Array	mid	From, Count
Array	few	
Array	any	Count
Array	isTrue	
Array	isFalse	
Array	rest	Start, Count
Array	otherthan	ArrayWithElementsToExclude
Array	where	Term
Array	spliteach	SplitExpression, RetainSplitterAlso InResult
Array	slices	SliceExpression
Array	core	
Array	nicejoin	JoinString, EndString, SubArrayString
Array	fjoin	HeadLength, JoinString
Array	funcjoin	HeadLength, FindString, FirstString, MidString, LastString
Array	injoin	JoinWith
Array	merge	OtherArray, Function
Array	across	OtherArray, Function
Array	across	OtherArray, Function
Array	insert	Value, AfterLastFlag
Array	pairmatch	AtFoldValue, AtReverseFoldValue, StartFrom
Array	fold	AtFoldValue, AtReverseFoldValue
Array	filter	Function
Array	plot	Mode
Array	removeByVal	Value

Array	graph	Mode
Array	car	
Array	cdr	
Array	head	
Array	tail	
Array	equal	Array, CheckLength, StartFrom
Array	equalvalues	Array, CheckLength, StartFrom
Array	compare	
Array	pack	
Array	multisort	
Array	clean	Expression, ReplaceWith
Array	is	Thing, IsNot
Array	isnull	
Array	isnotnull	
Array	match	Expression
Array	matchcolumn	Expression, Column
Array	matchrow	Expression, Row
Array	matchindex	Expression, IndexThenFromMatch
Array	matchvalue	Expression, IndexThenFromMatch
Array	include	
Array	notininclude	Item
Array	flatten	
Array	forward	Function, StartValue
Array	backward	Function, StartValue
Array	rotaterows	NumberOfSteps
Array	rotatecolumns	NumberOfSteps
Array	parts	NumberOfParts, SpecificPart
Array	half	
Array	halves	
Array	thirds	
Array	fourths	
Array	firsthalf	
Array	secondhalf	
Array	flipparts	
Array	rotate	NumberOfSteps
Array	exec	
Array	maprow	Function
Array	mapper	Function

Array	maplist	fun
Array	maprow	Function
Array	mapper	
Array	maplist	fun
Array	i	PreviousArray
Array	__\$	
Array	ri	
Array	ri	
Array	ci	
Array	\$	
Array	\$\$	
Array	\$\$\$	
Array	\$ _	
Array	\$x	
Array	x\$	
Array	\$X	
Array	X\$	
Array	\$CELLS	
Array	\$R	
Array	\$C	
Array	\$A	Parameter
Array	\$dth	
Array	\$diag	
Array	\$d	
Array	ids	
Array	rowcount	
Array	colcount	
Array	size2d	
Array	size	
Array	cube	
Array	slides	
Array	o	
Array	flatten	
Array	remove	
Array	removewith	
Array	objects	
Array	eval	
Array	set	
Array	setrow	Row, Array
Array	setcolumn	Column, ColumnValues
Array	flip	

Array	reverselevel	Level
Array	shiftlevel	Level, NumberOfTimes
Array	fillwith	
Array	branch	
Array	branchvalues	
Array	clearcopy	FillWith
Array	clear	FillWith
Array	filtermatches	MatchIdenticalMatr ix, OnlyMatches
Array	replace	ExpressionArrayOrV alues, ReplaceWith
Array	replicate	Count
Array	unwrapleaf	
Array	appendfunction	Function
Array	\$\$F	
Array	across	OtherArray, Function
Array	pair	Value, OnRight
Array	except	
Array	ntimes	Function, NumberOfIterations , Accuracy, Converge
Array	converge	Function, NumberOfIterations , Accuracy, Converge
Array	repeatntimes	Function, NumberOfIterations , Accuracy, Converge
Array	pieces	Width, Function
Array	foldl	Function, StartSeed
Array	foldr	Function, StartSeed
Array	partitiononcondi tion	TakeDropOrAllFlag, Function, Parameter
Array	filteronconditio n	TakeDropOrAllFlag, Function, Parameter
Array	collectwhile	
Array	suchthat	
Array	collect	
Array	takewhile	
Array	dropwhile	
Array	collectwhileasve ctor	
Array	suchthataasvector	
Array	collectasvector	

Array	takewhileasvector	
Array	dropwhileasvector	
Array	splitwhile	
Array	splitwhileasvector	
Array	det	Array
Array	adjoint	Array
Array	inverse	Array
Array	determinant	
Array	adjoint	
Array	inverse	
Array	addsequence	InFront, StartFrom, OptionalSequenceArray
Array	addrow	NumberOfRows
Array	addcolumn	NumberOfColumns
Array	insertrow	Index, NumberOfRows
Array	insertcolumn	Index, NumberOfColumns
Array	deleterow	Where
Array	deletecolumn	Where
Array	ar	
Array	ac	
Array	dr	
Array	dc	
Array	ir	
Array	ic	
Array	bindcolumn	
Array	filteronrow	Condition, ExtractColumns, FilterOnColumn
Array	filteroncolumn	Condition, ExtractColumns, FilterOnColumn
Array	aggregate	Columns, Function, Params
Array	lookup	
Array	reversesort	Function
Array	printf	StyleString, JoinString
Array	atindex	
Array	data	
Array	result	
Array	type	
Array	checktype	TypeArray, ForceCheckOnVariables

Array	numbers	ForceCheckOnVariables
Array	drop	
Array	keep	
Array	nullifyobjects	Recursive
Array	ZJSON	Recursive
Array	makekeyarray	Recursive
Array	atnode	Function, SubtractByArray, ScaleByArray, DoNotShowIndices
Array	nodeindex	
Array	indices	Function, SubtractByArray, ScaleByArray, DoNotShowIndices, RowArray
Array	xy	FunctionArray, OffsetArray, ScaleArray, GiveIndicesAlso, DoCentering
Array	xypanel	FunctionArray, OffsetArray, ScaleArray, GiveIndicesAlso
Array	tablelookup	RowValueMatch, ColumnValueMatch
Array	t	
Array	c	
Array	r	
Array	uncrostab	UptoColumn
Array	crostab	RowSet, ColSet, PageSet, DataSet
Array	findcellref	Values
Array	setaxis	Axis, ColumnValues
Array	joincolumnswith	ArrayOfJoinCharacters, IsRepeat
Array	joinrowswith	ArrayOfJoinCharacters, IsRepeat
Array	notwithinlimits	LimitArray, IncludeEdges
Array	withinlimits	LimitArray, IncludeEdges
Array	concatall	LimitArray, IncludeEdges
Array	deepcopy	
Array	async	Iterator, CallBack
Array	value	Function
Array	of	
Array	truefalse	IsCheckTrueFunctionList, IsCheckFalseFunctionList, DoFlattenFirst

Array	fixat	Index, Fix
Array	prefix	
Array	suffix	
Array	chunks	ChunkSize
Array	add	Thing
Array	inc	
Date	format	mask, utc
Function	\$	
Function	merge	array, args
Function	argumentNames	
Function	update	array, args
Function	curry	
Function	delay	timeout
Function	defer	
Function	argumentaslist	
Function	wrap	wrapper
Function	fury	AvoidArguments
Number	units	
Number	fuzzy	
Number	getfuzzy	
Number	larger	OtherNumber
Number	smaller	OtherNumber
Number	normalizeunits	OtherUnits
Number	add	OtherNumber
Number	subtract	OtherNumber
Number	multiply	OtherNumber
Number	divide	OtherNumber
Number	div	OtherNumber
Number	power	OtherNumber
Number	m	
Number	a	
Number	s	
Number	d	
Number	di	
Number	p	
Number	makeunits	String
Number	compareto	OtherNumber
Number	equals	OtherNumber

Number	notequals	OtherNumber
Number	string	
Number	setunit	ToUnits
Number	convert	ToUnits
Number	eval	
Number	replicate	Count
Number	isin	Array, IgnoreCase
Number	trim	
Number	slice	
Number	withinlimits	LimitArray, IncludeEdges
Object	print	
Object	makecopy	
Object	core	
Object	keyprint	ElementSplit, LineSplit, QuoteKey, BracketWrap
String	reverse	
String	eval	
String	trim	
String	trimend	
String	trimbegin	
String	trimstart	
String	toInt	
String	duplicate	NumberOfTimes
String	encrypt	Seed
String	decrypt	Seed
String	endsWith	str
String	startsWith	str
String	inarray	Array, IgnoreCase
String	isin	
String	cube	
String	equation	Replace
String	replicate	Count, JoinWith
String	pieces	Each
String	uniteval	
String	tagvalues	Tags
String	tokens	Splitter
String	clean	Expression, ReplaceWith
String	insert	index, string
String	splitat	AtArray
String	substringindices	SubString

## 9. HOW TO WORK WITH ZCUBES

Here are the links to help you on working with ZCubes and learn more about ZCubes features.

[http://wiki.zcubes.com/Learn\\_ZCubes](http://wiki.zcubes.com/Learn_ZCubes)





*www.zcubes.com*

*Do It All!*