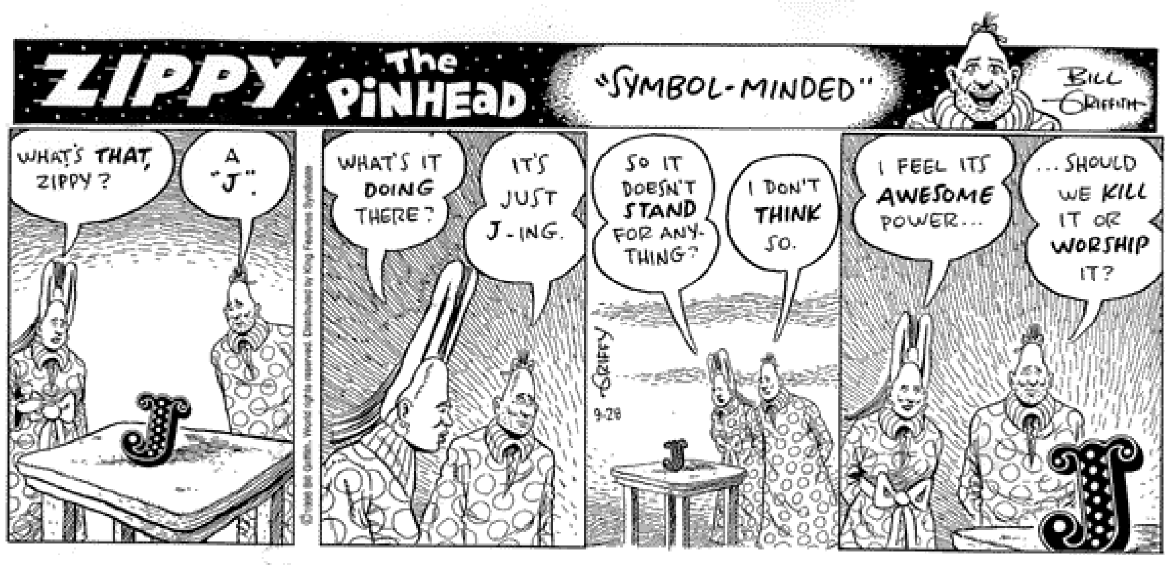
Finexec

Financial & Actuarial Computing

with the J Programming Language

(Notation as a Tool of Thought – Kenneth E. Iverson)

William Szuch



1 Abstract

The purpose of this note is to introduce Finexec (see: http://www.finexec.com.au) and the J Programming Language as a tool for, educational, financial and actuarial computing. J is the modern successor to APL (A Programming Language), a language that was developed in the 1960s. APL suffered from its use of an unorthodox character set which did not sit well with ASCII text displays and keyboards.

J is truly a new language by APL's author, Kenneth E. Iverson, and implemented by his son Eric and Roger Hui. J is more than just an ASCII-fied APL but retains the same mathematical principles. The syntax of J is simple: all verbs (functions) have the same priority and parentheses are the only way to alter execution order. J's components are named using English grammar terminology. Function are called verbs, adverbs and conjunctions modify the action of a verb and data is referred to as nouns. (ie. a verb carries out an action on a noun).

J is an executable mathematical notation and consists largely of 118 primitives or parts of speech (73 verbs, 11 adverbs, 27 conjunctions, 7 other).

We explore through a series of simple examples of how the J can change the approach and thinking to be applied to compound interest and life table calculations using only the primitives defined in the J.

Contents

2 Why the J Programming Language?

J is a powerful array programming language suitable for dealing with actuarial problems in finance, life and non-life insurance and for carrying out complex calculations, data analysis, simulation and modelling.

The power of J is in its notation and syntax that allows one to focus on the problem rather than get submerged in the technicalities of a programming language. It is particularly strong in the mathematical, statistical, and logical analysis of data and for building new and better solutions to old problems.

J is portable and runs on Windows, Unix, Mac, PocketPC handhelds and smartphones, both as a GUI and in a console. J software and systems can be installed and distributed for free which allows for the sharing of tools and ideas across a community of J users.

J uses English grammar terminology in preference to terms used in other programming languages and mathematics. For example, the + sign in mathematics might be considered as a function but in J it is referred to as a verb. Other English grammar terms used in J are: alphabet, word, sentence, noun, adverb and conjunction.

The concept behind an array based programming language is that its operations apply at once to an entire array. This makes it possible to think and operate whole data(array), without having to resort to explicit loops of individual scalar operations.

J array programming language primitives express ideas about data manipulation, often providing easier methods and improved prospective to a problem.

J can also be used as a simple desk calculator and is an excellent way to start learning J.

2.1 J Dictionary and Vocabulary

The J Dictionary is the primary source of information that defines the J Programming Language. The J Vocabulary is a list of primitives defined in J Dictionary.

2.2 Learning J

There are a number of tools available to start learning J. These can be found at the Jsoftware website and include books, essays, articles, tutorials etc.

2.3 J Application Library (JAL)

The J application library consists of:

- the base library, which has utilities required for the development system. This is always included with the distribution.
- the addons library, almost all of which are optional additions. The one exception is the J Http Server (JHS) addon, which is also included with the distribution.

3 Some Features of J

3.1 Standard ASCII Character Set

J uses the standard ASCII character set instead of the special APL set.

3.2 J Primitives

A J primitive is a word whose meaning is assigned by the J language.
J has a rich set of primitives to manipulate data and verbs. (See Appendix 2)

A mnemonic one- or two-character spelling of primitives.

Verbs can be modified, combined and defined using facilities such as adverbs and conjunctions.

3.3 Array Oriented

Single entities, lists, tables and arrays are treated in a very systematic and general manner. The elements of arrays can themselves be arrays. Some simple examples.

```
   1+1
2

   1 + 1 2 3
2 3 4

   1 2 3 + 1 2 3
2 4 6
```

Define A as an array.

```
  [ A =: i. 2 3      NB. This means create an array of 2 rows, 3 columns incrementimg by 1 at each step.
0 1 2
3 4 5

   1+A
1 2 3
4 5 6

   1 2 3+"1 A    NB. + has been modified by the rank conjunction “.
1 3 5
4 6 8

   1 2+"2 A     NB. + has been modified by the rank conjunction “.
1 2 3
5 6 7
```

3.4 No Order of Execution Hierarchy

No order of execution hierarchy among function. Execution is right to left.

```
    8 % 2*(2+2)
1
```

3.5 Negative Numbers

In J a negative number is represented with an underbar symbol, different from the minus symbol eg: _12.123. It is part of the number like a decimal point.

3.6 Ambivalent Verbs

Use of ambivalent verbs that can take a right argument of right and left and right arguments.

   ^1   NB. e^1
2.71828

   10^2   NB. 10^2
100

3.7 Adverbs and Conjunctions

Use of adverbs and conjunctions to modify, compose and combine verbs enabling a rich set of operations based on a small set of verbs.

/ (insert adverb)
```
   sum =: +/
   sum 1 2 3 4 5
15

   minmax =: (<./;>./)
   minmax 1 2 3 4 5
┌─┬─┐
│1│5│
└─┴─┘
```

@ (atop conjunction)
```
  minmaxa=: >@minmax
  minmaxa 1 2 3 4 5
1 5
```

3.8 Trains - Hooks and Forks

A train is an isolated sequence of verbs.
A train of two (hook) or three (fork) verbs produces a verb and (by repeated resolution), a verb train of any length also produces a verb.

```
        HOOK                  FORK              CAPPED FORK

     g       g           g         g             g      g
    / \     / \         / \       / \            |      |
   y   h   x   h       f   h     f   h           h      h
       |       |       |   |    / \ / \          |     / \
       y       y       y   y    x y x y          y    x   y
```

For example, a simple fork containing three verbs:

```
   mean =: +/  %  #
```

3.9 Boxed Data

Boxed data that can have numeric and text entities.

```
 123;'asd';12.01;'this is text'
```

```
┌───┬───┬─────┬────────────┐
│123│asd│12.01│this is text│
└───┴───┴─────┴────────────┘
```

3.10 Tacit Programming

A feature of J is tacit (or functional) programming that does not require the explicit mention or definition of arguments to a verb (ie: function) and the use of assignments to assign names to the verb. For example, verb mean defined as a fork is a tacit verb that calculates mean without reference to the arguments (ie: data) which can be a numeric atom (ie: scalar), list (ie: vector), table (ie: array) or multi-dimension array.

```
   (+/ % #) 1 2 3 4 5
3

   ((+/ % #),mean) 1 2 3 4 5
3 3

  mean i. 3 5
5 6 7 8 9

   mean”1 i. 3 5
2 7 12
```

J tacit is:

- self-contained
- suitable for modular design
- automatic optimization (supported by special code)
- elimination of unwanted side effects
- no name conflicts

Using tacits and explicits to define more complex verbs.

3.11 R Statistical Package

J has an interface to R, i.e. R for its statistical functions and J as an expressive application development language that can call R directly.

3.12 Jdatabase (jd)

Jdatabase (Jd) is a relational database management system (RDBMS) from Jsoftware that is implemented in J. Jd is a package that is added to J with the Package Manager. Jd extends J so that you have the best of two worlds: J with an integrated, high-performance, columnar data base system.

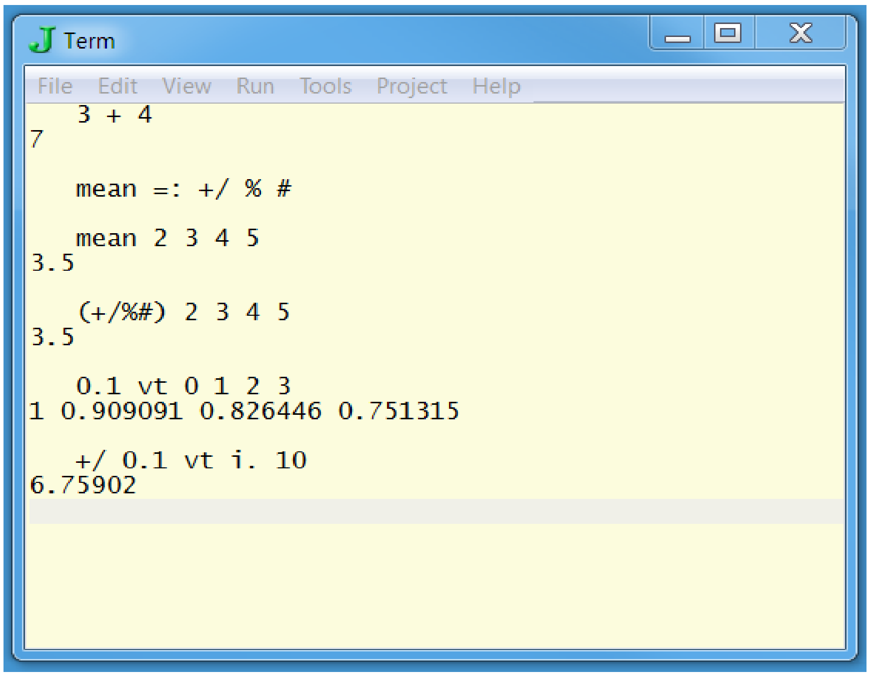
Jd is geared towards storing and analyzing large amounts of data. Jd is free for non-commercial use.

Jd lives openly and dynamically in the J execution and development environment, so that the full power of J is available to the application developer. For example, Jd columns are mapped to J nouns, so built-in J primitives can apply directly to the data.

It works well with large tables (millions of rows to billions), multiple tables connected by joins, structured data, numerical data, and complex queries and aggregations.

3.13 J Execution Window

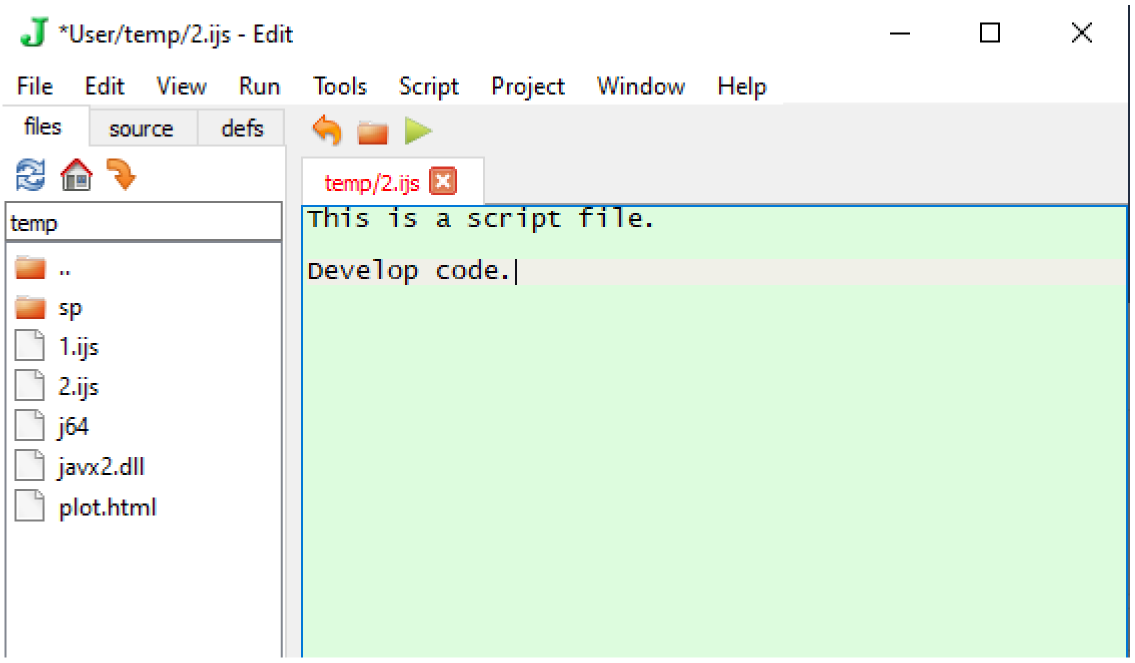
The examples in the note are as executed in a J session execution window. Lines that are entered for execution are indented three spaces and the J answers start at the margin. The following diagram shows a J execution window with some simple examples being executed.



```
   3 + 4
7

   mean =: +/ % #

   mean 2 3 4 5
3.5

   (+/%#) 2 3 4 5
3.5

   0.1 vt 0 1 2 3
1 0.909091 0.826446 0.751315

   +/ 0.1 vt i. 10
6.75902
```

3.14 J Script Window

The J script window provides an environment for editing J script files that can be loaded for a project.



```
This is a script file.

Develop code.
```

3.15 J User Community

An active community of J users that provide support and help with queries.

3.16 J Library

The J library includes:

- J Application Library (JAL): System Library and Addons.
- Miscellaneous script files.
- Phrases: useful J expressions that can be cut and pasted into your programs.

4 Compound Interest – A Cash Flow Approach

A cash flow approach to understanding basic financial calculations and solving problems can be adopted using J. This is made possible through the use of the J interactive programming language. Many financial calculations can be regarded as comprising of the following three parts:

(1) amount and time of the cash flow
(2) selecting a suitable simple or effective rate of interest and time unit
(3) probability that the cash flow is paid or received

In basic finance the probability of the cash flow being paid or received.is usually ignored. This is considered in other categories such as life contingencies.

Definitions are built around defining:
S = simple interest rate for a time unit
E = effective interest rate for a time unit
Ev = list of variable effective interest rates for time units
C = amount of cash flow
T = period of the cash flow in time units

The choice of an effective interest rate and time unit for a calculation should be carefully considered as this will impact the approach to solving the problem. The time unit is used to determine the timing of the cash flow. A well defined effective interest rate and time unit can make it easier to construct and solve the problem.

There is no one right approach and this will be determined on the how the cash flow and interest rate are defined for the problem. One approach for compound interest problems is to select the time unit that applies to the effective interest rate.
For example, if the effective interest rate is 6.0% yearly a time unit of a year is appropriate.

In this section we introduce the simple verbs v, vt , vvt and pvcf and show how they can be applied to compound interest calculations including variable effective interest rates which are not usually used because of the additional calculation complexity especially with spreadsheets. The verbs have been programmed as tacits and therefore have no dependencies and resolves to J primitives.

4.1 The verb v – Present Value Factor

The verb v is defined as follows:

Present value of 1 payable in 1 time unit for an effective interest rate E.
Syntax: v(E)
E = effective interest rate for one time unit

The verb v is said to be monadic as it takes only a right argument E.
The verb takes data as a single item, list, table and array and the tacit code has no loops.

```
v =: [: % 1 + ]
```

4.1.1 Examples.

E as a single item

```
   v(0.05)
0.952381
```

E as a list

```
   v(0.03 0.04 0.05)
0.970874 0.961538 0.952381
```

E as at table

```
   v(0.03 0.04 0.05,:0.1 0.12 0.15)
0.970874 0.961538 0.952381
0.909091 0.892857 0.869565
```

4.2 The verb vt – Constant Effective Interest Rates: Present Value Factors

The verb vt is defined as follows and is an extension the verb v:

Present value of 1 payable in a period of T time units
for a constant effective interest rate E over the period.
(E)vt(T)
E = constant effective interest rate per time unit over a period
T = number of time units in the period

The verb vt is said to be dyadic as it takes left and right arguments E and T. All J verbs must take a right argument and verbs that only take a right argument are called monadic. As with the verb v no loops.

```
vt =: ([: % 1 + ])~ ^"_ 0 ]
```

4.2.1 Examples

```
   (0)vt(0)
1

   (0.05)vt(10)
0.613913

   (0 0.05 0.1 0.15)vt(10)
1 0.613913 0.385543 0.247185

   (0.05)vt(0 5 10)
1 0.783526 0.613913

   (0 0.05 0.1 0.15)vt(0 5 10)
1        1        1        1
1 0.783526 0.620921 0.497177
1 0.613913 0.385543 0.247185
```

4.3 The verb vvt – Constant and Variable Effective Interest Rates: Present Value Factors

The verb vvt is defined as follows and is an extension of verbs v and vt:

Present value of 1 payable in a period of T time units for a constant or variable effective interest rates Ev assuming a constant time unit.

Syntax: (Ev)vvt(T)
Ev : list of effective interest rates for the constant time unit with the last rate
extended to time T
T : list of periods in time units

The verb vvt is dyadic as it takes left and right arguments Ev and T and is built from smaller tacits. Again, no loops.

```
vvt =: [: % */"1@:((1 + ((([ {. ] , (# {:))~"_ 0) (1 + <.))) ^ [: (#&1@:<. , ] - <.)"0 ])
```

4.3.1 Example 1 – Present Value Factors

The verb vvt is used to calculate one or a list of present value factors for $1 for a constant or variable effective interest rates Ev over a range of times T. The list of present value factors can then be used in further calculations.

```
   (0.1)vvt(1)
0.909091

  (0.1)vvt(0.5 1 1.5 10.5)
0.953463 0.909091 0.866784 0.367601

  0.1 0.05 vvt 0.5 1 1.5 2 10.5
0.953463 0.909091 0.887182 0.865801 0.571885
```

Brackets can be used for clarity and the result saved (ie: as R1) for more calculations.

```
   R1 =: (0.1 0.05 0.04) vvt (0 1 2 3)
   R1
1 0.909091 0.865801 0.832501
```

We can extend the application of vvt to a new verb by using the J bond conjunction & to fix a left or right argument as follows Ev&vt or vt&T. For example we can fix the effective interest rate at 5% and define vt05 which can be used as monadic verb for variable times T.

```
   vvt05 =: (0.05)&vvt
   vvt05 1 2 3 4
0.952381 0.907029 0.863838 0.822702
```

Similarly, we can fix variable times and then use the effective rates as the right argument.

```
   vvtT =: vvt&(0 1 2.5 5 10)
   vvtT(0.1 0.04)
1 0.909091 0.857151 0.777095 0.638715
```

4.4 Example 2 – Periodic Cash Flows

A simple extension of vvt using only the J primitives i. + and / is used to calculate a list of present value factors for periodic cash flows and the present value of the cash flow.

The J primitive verb i. generates a list of integers for example:

```
   i. 5
0 1 2 3 4
```

The J primitive verb + and adverb / are combined to form the new verb +/ which is used to calculate the sum of a list, for example:

```
  +/ 2 3 4 5
14
```

The simple annuity certain of N level periodic payments of $1 per time unit commencing after D time units can be calculated as follows:

```
   +/ (Ev)vvt(D+i.N)
```

Payment in arrears of 1 time unit:

```
   +/ (0.1)vvt(1+i. 10)
6.14457
```

Payments in advance:

```
  +/ (0.1)vvt (i. 10)
6.75902
```

Payments in arrears 2.5 time units:

```
  +/ (0.1)vvt (2.5 + i. 10)
5.32601
```

The interest rate can be varied over the period of the annuity. Assuming payments in arrears of 1 and 2.5 time units and a 10% rate for the first 3 time units, 8% for the next 2 time units and 5% thereafter the value of the annuity certain for 10 payments is calculated as follows.

Payments commence after 1 time unit:

```
   +/ ((3#0.1),(2#0.08),0.05)vvt (1+i. 10)
6.6154
```

Payments commence after 2.5 time units:

```
  +/ ((3#0.1),(2#0.08),0.05)vvt(2.5+i. 10)
5.99957
```

The J primitive # is used to copy its right argument and the comma , is used to append eg:

```
  ((3#0.1),(2#0.08),0.05)
0.1 0.1 0.1 0.08 0.08 0.05
```

It is a simple step to calculate the present value of a variable periodic cash flow (CF) of N payments as follows:

```
   +/ CF * (Ev)vvt(D + i.N)
```

Assuming 10% effective interest rate and payment increase linearly from 1 to 10 the present value is calculated as follows:

```
   +/ (1+i. 10) * (0.1)vvt(1+i. 10)
29.0359
```

If payments decrease from 10 to 1 (ie reversed) the present value is calculated as follows:

```
+/ (|. 1+i. 10) * (0.1)vvt(1+i. 10)
38.5543
```

The J primitive |. is used to reverse a list eg:

```
   |. 1 2 3
3 2 1
```

CF can be any list of periodic payments eg:

```
+/ (1 3 2 4 10) * (0.1 0.05)vvt(i. 5)
16.6102
```

4.5 Example 4 – Loan Repayment Calculations

We show how loan repayments for several scenarios can be calculated using only vvt and some J primitives.

A =: 100000 NB. Amount of loan
N =: 10 NB. Term of loan in years

4.5.1 Scenario 1

Level monthly repayments
10.00% yearly effective interest rate payable monthly

```
  S1 =: 100000 % +/ (0.1%12)vvt (>: i. 12*10)
  S1
1321.51
```

4.5.2 Scenario 2

Level monthly repayments
10.00% yearly effective interest rate payable monthly decreasing by a level 0.5% each year after the first year

```
   S2 =: 100000%+/((%12)*12#(0.1-0.005*i.10)) vt (>: i. 120)
   S2
1249.71
```

4.5.3 Scenario 3

Monthly repayments increasing by 2.5% each year
10.00% yearly effective interest rate payable monthly decreasing by a level 0.5% each year after the first year

First year monthly payments:

```
   S3 =: 100000 % +/ (12# 1.025^i. 10)*((%12) * 12# (0.1 -
0.005*i. 10)) vvt (>: i. 120)
   S3
1133.25
```

4.5.4 Scenario 4

Monthly repayments increasing by 2.5/12% each month
10.00% yearly effective interest rate payable monthly decreasing by a level 0.5% each year after the first year

First monthly payment:

```
   S4 =: 100000 % +/ ((1+0.025%12)^ i.120)*((%12)* 12#(0.1-.005*i.
10)) vvt (>: i. 120)
S4
1119.22
```

4.6 Example 3 – Non Periodic Cash Flows

The present value of a cash flow is simply the present value factors multiplied by the corresponding cash and then summed. This can be shown as:

```
   +/ CF * (Ev)vvt(T)

   +/ (2 5 12 100) * (0.1 0.05 0.025)vvt(0.5 2 5 20)
71.3959
```

4.7 The verb pvcf – Present Value of a Cash Flow for a Constant Effective Rate

This verb is used to illustrate the calculation of Treasury Bond price.

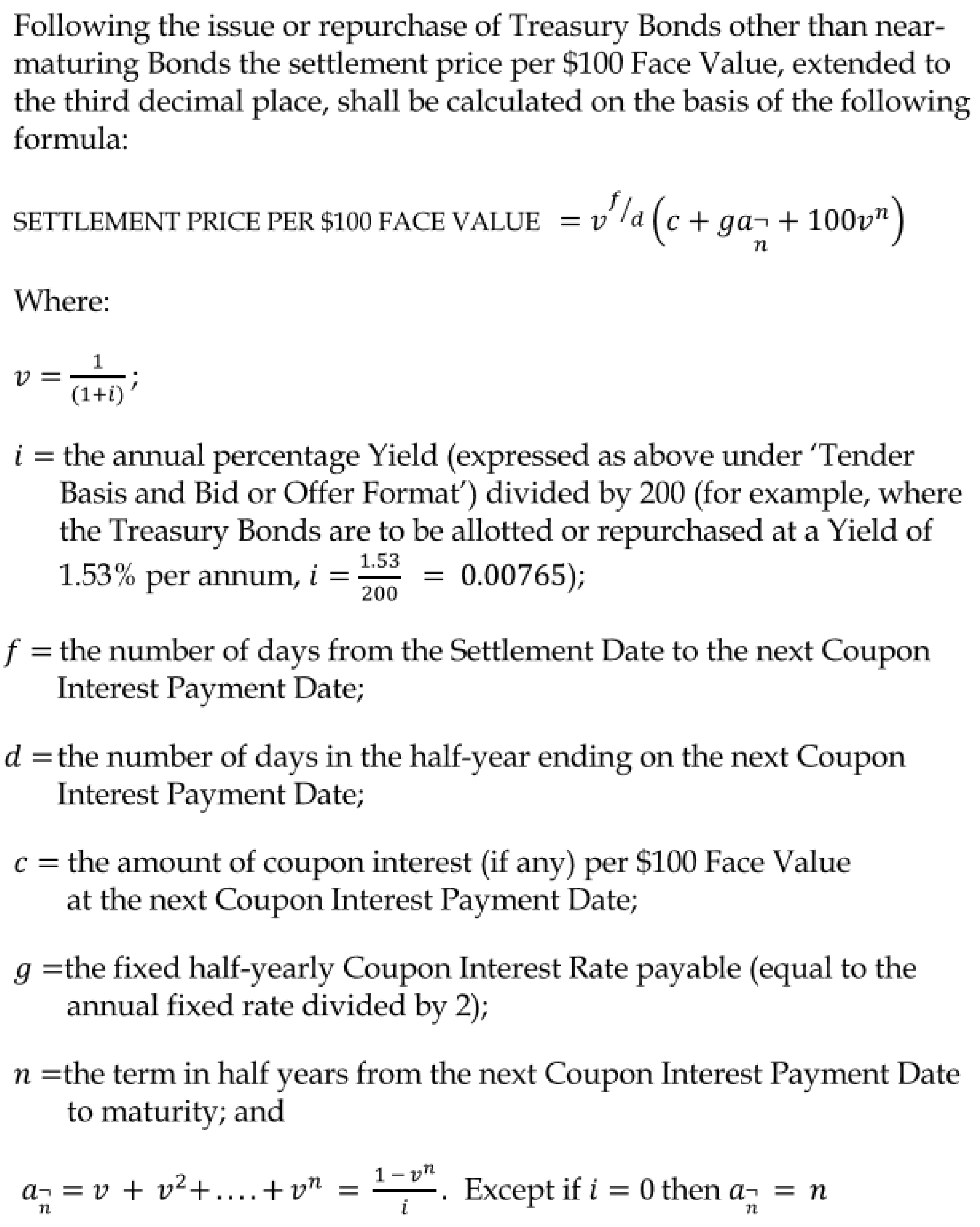
The verb pvcf is defined as follows and is an extension of the verb vt.

Present value of a cash flow C payable at the periods of T time units for a constant effective interest rate E over the period.
syntax:
(E)pvcf(C;T)
E = constant effective interest rate per time unit over the period.
C = amount of cash flow
T = periods in time units of cash flow

```
pvcf=: +/@:(>@:([: 0&{ ]) * [ (,:@:([: % 1 + ])~ ^"_ 0 ]) >@:([: 1&{ ]))
```

4.7.1 Example: RBA Basic Treasury Bond Formula – Settlement Price

From the Australian Office of Financial Management
Information Memorandum – Treasury Bonds (26/7/2019)



Following the issue or repurchase of Treasury Bonds other than near-maturing Bonds the settlement price per $100 Face Value, extended to the third decimal place, shall be calculated on the basis of the following formula:

$$\text{SETTLEMENT PRICE PER \$100 FACE VALUE } = v^{f/d}\left(c + g a_{\overline{n}|} + 100v^n\right)$$

Where:

$v = \frac{1}{(1+i)}$;

i = the annual percentage Yield (expressed as above under 'Tender Basis and Bid or Offer Format') divided by 200 (for example, where the Treasury Bonds are to be allotted or repurchased at a Yield of 1.53% per annum, $i = \frac{1.53}{200} = 0.00765$);

f = the number of days from the Settlement Date to the next Coupon Interest Payment Date;

d = the number of days in the half-year ending on the next Coupon Interest Payment Date;

c = the amount of coupon interest (if any) per $100 Face Value at the next Coupon Interest Payment Date;

g = the fixed half-yearly Coupon Interest Rate payable (equal to the annual fixed rate divided by 2);

n = the term in half years from the next Coupon Interest Payment Date to maturity; and

$a_{\overline{n}|} = v + v^2 + \ldots + v^n = \frac{1-v^n}{i}$. Except if $i = 0$ then $a_{\overline{n}|} = n$

Calculate the price per $100 face value on 1/11/1992 to yield 8.30% pa to maturity for a 12.5% 15 January 1998 bond. Example from RBA Note – 6/9/1995)
(For the preceding equation: i = 0.0415, f = 75, d =184, g = 6.25, n = 10, c = 4.15)

Taking a cash flow approach using pvcf, vt and v:

E =: 0.0415	NB. Effective half yearly interest rate
C =: (11#6.25),100	NB. Cash flow
T =: (75%184)+(i.11),10	NB. Time units

```
   (E)pvcf(C;T)
121.132
```

Using only vt:

```
   +/ C*(E)vt(T)
121.132
```

Using only v:

```
   +/C*(v(E))^T
121.132
```

In full using J primitives

```
  +/C*(([: % 1+])E)^T
121.132
```

5 Life Table

```
Simple examples of using J for life table calulations.
ALT_2015_17_M is a list of qx values from age 0 (See Appendix 1).
ALT_2015_17_M =: 0.00343 0.00027 0.00016 0.00014 0.00011 …
```

5.1 Probability of Survival

```
From Age 0
  */\(1-ALT_2015_17_M)
0.99657 0.996301 0.996142 0.996002 0.995892 …

From Age 65
  */\65}.(1-ALT_2015_17_M)
0.9899 0.979021 0.967273 0.954534 0.940703 …
```

5.2 Expectation of Life

```
Age 0
   0.5+ +/ */\(1-ALT_2015_17_M)
80.4621

Age 65
0.5+ +/ */\ 65}.(1-ALT_2015_17_M)
19.6474
```

The preceding operation can be combined to a tacit defined as:

Calculate the complete expectation of life without extended Qx table.
Qx)exz(Age)
Qx = mortality table starting at age = 0
Age = integer ages

```
ALT_2015_17_M exz 0 65
80.4621 19.6474
```

5.3 Life Annuity

It is a simple matter to calculate a life annuity with payment in arrears from the probability of survival and present value factors.

E = effective annual interest rate

```
Age 0
  +/  ((0.02)vt(1+i. #ALT_2015_17_M)) * */\(1-ALT_2015_17_M)
39.1876

Age 65
  +/  ((0.02)vt(1+i. # 65}.ALT_2015_17_M)) * */\65}.(1-ALT_2015_17_M)
15.2981
```

The preceding operation can be combined to a tacit defined as:

Single life annuity with yearly payments of 1 in arrears.
Syntax:
(Qx;E)ax(Age)
Qx = mortality table starting at age = 0
E = yearly effective interest rate
Age = integer ages

```
   (ALT_2015_17_M;0.02)ax(0 65)
39.1876 15.2981
```

5.4 Stochastics Model

Produce a table of survival periods from age 65 for 100,000 simulations using ALT_2015_17_M. For this example, we develop an explicit dyadic verb that has no loops and generates a table for the simulation.

```
NB. ========================================================
NB. (Qx)stocha(Age;S)
NB. Qx = mortality table – commencing at age 0
NB. Age = age of life (integer)
NB. S = number of simulation
NB. (ALT_2015_17_M)stocha(65;100000)
stocha =: 4 : 0
Qx =. x
'Age S' =. y
rnd01 =. ?.@$&0             NB. Random number generator (tacit)
QAge =. Age}.Qx             NB. Mortlity table from Age
RND =. rnd01 S,#QAge        NB. Random numbers table
*/\"1 QAge<"1 RND           NB. Survival periods
)
```

The result is a table of 100000 rows 36 columns

The first 10 rows of the table:

```
1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
   +/+/(ALT_2015_17_M)stocha(65;100000)
1915767
```

If required this verb could be written as a tacit.

6 Finexec

Objectives of Finexec include:

- Promote learning the J Programming Language for solving problems.
- Develop financial and actuarial computing skills and tools using J.
- Establish a community of Finexec and J users that share and develop tools and ideas for financial and actuarial computing.

For details about Finexec go to: http://finexec.com.au. Finexec is in the early stages of development will be undergoing changes.

7 Comment

We have only briefly explored the power and flexibility of J as applied to simple compound interest and life table calculations. The example calculations have been carried with just one sentence in a J execution window which provides for interactive feedback for carrying out the calculations. Similar power and flexibility can and be achieved from the development of tools and applications for more complicated calculations in financial, life and non-life actuarial work which can be shared across a community of J users. A bit of effort is initially required to learn J but it is well worth it and is intellectually rewarding unlike other computing languages. The best way to learn is to download the system and start using J and studying the lessons and labs provided with the system.

8 References

8.1 Books

Hui R.K.W. and Iverson K.E.
J Dictionary
Iverson Software Inc. Toronto 1998

Howard A. Peelle
Mathematical Computing in J:Introduction (Volume1)
Research Studies Press Ltd. 2004

Thomson N.J.
J: The Natural Language for Analytic Computing
Research Studies Press Ltd. 2001

Rich Henry
J for C Programmers
Copyright Henry Rich 2008

Stokes Roger
Learning J
Copyright Roger Stokes 2015

8.2 Articles

Iverson K.E.
Notation as a Tool of Thought
IBM Thomas J. Watson Research Centre

Quitana Jose Mario
Tacit Programming in Action: A Decade of Experience
Jsoftware Conference 23/24 July 2012 Toronto

8.3 Websites

www.jsoftware.com

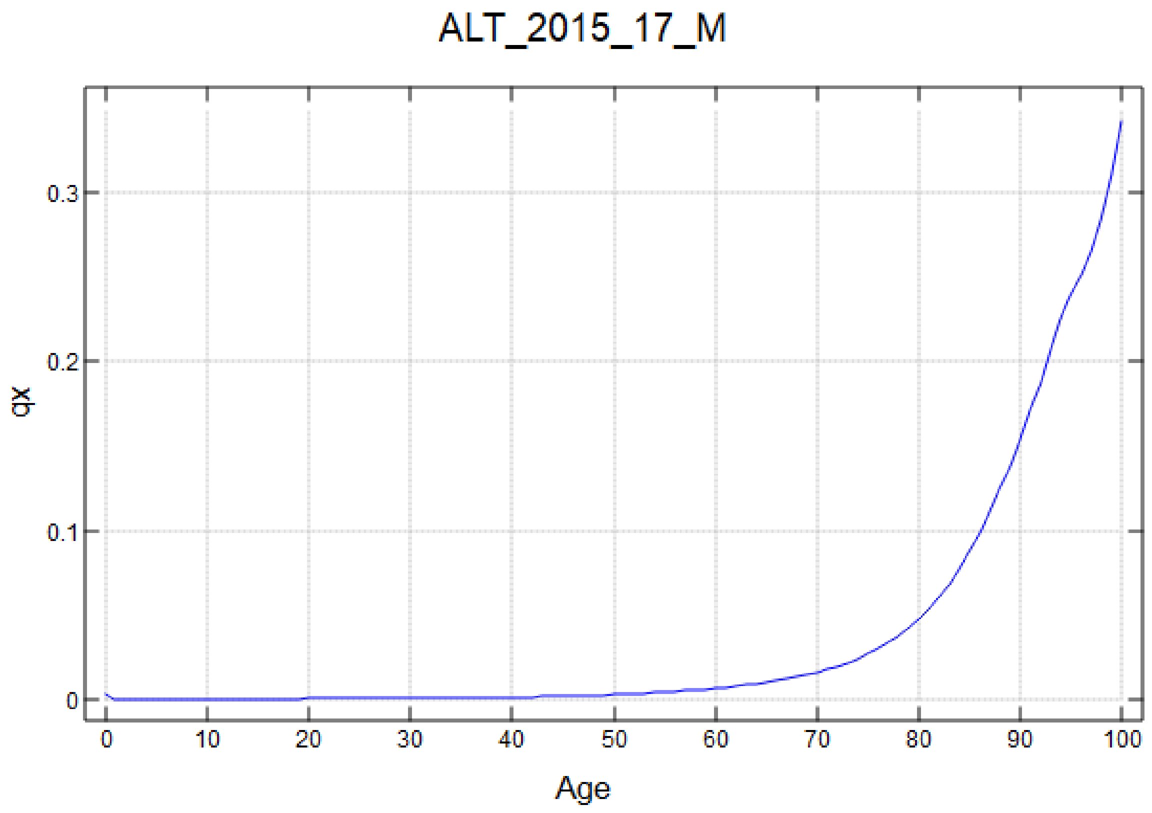
code.jsoftware.com/wiki/NuVoc

www.finexec.com.au

9 Appendix 1 – Life Table ALT_2015_17_M

The qx values for the ALT_2015_17_M table are shown.
Run in a J session to load values.

```
data =. [: ". rplc&(LF;' ') NB. data.ijs

ALT_2015_17_M =: data 0 : 0
0.00343 0.00027 0.00016 0.00014 0.00011 0.00010 0.00009 0.00008 0.00008 0.00008
0.00008 0.00009 0.00010 0.00013 0.00017 0.00024 0.00031 0.00040 0.00047 0.00053
0.00057 0.00059 0.00061 0.00063 0.00064 0.00065 0.00067 0.00069 0.00072 0.00075
0.00079 0.00083 0.00087 0.00092 0.00097 0.00103 0.00110 0.00116 0.00123 0.00132
0.00142 0.00154 0.00166 0.00179 0.00192 0.00206 0.00223 0.00240 0.00258 0.00277
0.00298 0.00321 0.00347 0.00378 0.00412 0.00448 0.00486 0.00527 0.00573 0.00622
0.00676 0.00732 0.00792 0.00857 0.00929 0.01010 0.01099 0.01200 0.01317 0.01449
0.01602 0.01776 0.01972 0.02191 0.02430 0.02699 0.03006 0.03358 0.03758 0.04216
0.04752 0.05366 0.06054 0.06828 0.07722 0.08735 0.09862 0.11090 0.12427 0.13867
0.15409 0.17078 0.18851 0.20688 0.22531 0.23916 0.25101 0.26474 0.28313 0.31095
0.34231
)


NB. load ‘plot’
NB. ('title ALT_2015_17_M';'xcaption Age';'ycaption qx')plot(ALT_2015_17_M)
```



10 Appendix 2 – J Primitives

=	Self-Classify • Equal	=.	Is (Local)	=:	Is (Global)
<	Box • Less Than	<.	Floor • Lesser Of (Min)	<:	Decrement • Less Or Equal
>	Open • Larger Than	>.	Ceiling • Larger of (Max)	>:	Increment • Larger Or Equal
_	Negative Sign / *Infinity*	_.	*Indeterminate*	_:	Infinity
+	Conjugate • Plus	+.	Real / Imaginary • GCD (Or)	+:	Double • Not-Or
*	Signum • Times	*.	Length/Angle • LCM (And)	*:	Square • Not-And
-	Negate • Minus	-.	Not • Less	-:	Halve • Match
%	Reciprocal • Divide	%.	Matrix Inverse • Matrix Divide	%:	Square Root • Root
^	Exponential • Power	^.	Natural Log • Logarithm	^:	**Power** (u^:n u^:v)
$	Shape Of • Shape	$.	Sparse	$:	Self-Reference
~	***Reflex • Passive / Evoke***	~.	Nub •	~:	Nub Sieve • Not-Equal
|	Magnitude • Residue	|.	Reverse • Rotate (Shift)	|:	Transpose
.	**Determinant • Dot Product**	..	**Even**	.:	**Odd**
:	**Explicit** / **Monad-Dyad**	:.	**Obverse**	::	**Adverse**
,	Ravel • Append	,.	Ravel Items • Stitch	,:	Itemize • Laminate
;	Raze • Link	;.	**Cut**	;:	Words • Sequential Machine
#	Tally • Copy	#.	Base 2 • Base	#:	Antibase 2 • Antibase
!	Factorial • Out Of	!.	**Fit (Customize)**	!:	**Foreign**
/	***Insert • Table***	/.	***Oblique • Key***	/:	Grade Up • Sort
\	***Prefix • Infix***	\.	***Suffix • Outfix***	\:	Grade Down • Sort
[	Same • Left			[:	Cap
]	Same • Right				
{	Catalogue • From	{.	Head • Take	{:	Tail • {:: Map • Fetch
}	***Item Amend • Amend*** (m} u})	}.	Behead • Drop	}:	Curtail •
"	**Rank** (m"n u"n m"v u"v)	".	Do • Numbers	":	Default Format • Format
`	**Tie (Gerund)**			`:	**Evoke Gerund**
@	**Atop**	@.	**Agenda**	@:	**At**
&	**Bond** / **Compose**	&. &.:	**Under (Dual)**	&:	**Appose**
?	Roll • Deal	?.	Roll • Deal (fixed seed)		
a.	*Alphabet*	a:	*Ace* (Boxed Empty)	A.	Anagram Index • Anagram
b.	***Boolean / Basic***	C.	Cycle-Direct • Permute	d.	**Derivative**
D.	**Derivative**	D:	**Secant Slope**	e.	Raze In • Member (In)
E.	• Member of Interval	f.	***Fix***	H.	**Hypergeometric**
i.	Integers • Index Of	i:	Steps • Index Of Last	I.	Indices • Interval Index
j.	Imaginary • Complex	L.	Level Of •	L:	**Level At**
M.	***Memo***	NB.	Comment	o.	Pi Times • Circle Function
p.	Roots • Polynomial	p..	Poly. Deriv. • Poly. Integral	p:	Primes
q:	Prime Factors • Prime Exponents	r.	Angle • Polar	s:	Symbol
S:	**Spread**	t.	***Taylor Coeff.*** (m t. u t.)	t:	***Weighted Taylor***
T.	**Taylor Approximation**	u:	Unicode	x:	Extended Precision
_9: to 9:	Constant Functions				

11 Appendix 3 - J Code

The J code for the tacit verbs defined in this note are constructed from only J primitives. The verbs are self-contained (ie. no dependencies) and have no name conflicts. The definitions can be copied to a J session and the examples in the note executed. The tacit verbs are built from smaller verbs.

```
v =: [: % 1 + ]
vt =: ([: % 1 + ])~ ^"_ 0 ]
vvt =: [: % */"1@:((1 + ((([ {. ] , (# {:))~"_ 0) (1 + <.))) ^ [: (#&1@:<. , ] - <.)"0 ])
pvcf=: +/@:(>@:([: 0&{ ]) * [ (,:@:([: % 1 + ])~ ^"_ 0 ]) >@:([: 1&{ ]))

exz =: (0.5 + +/@:(*/\@:(1 - ] }. [)))"_ 0
ax =: +/@:(([: */\ 1 - ] }. [: >@:(0&{) [) * ([: % 1 + [: >@:(1&{) [) ^ 1 + [: i. [: # [: */\ 1 - ] }. [:
>@:(0&{) [)"_ 0

mean=: +/ %  #
```

As an exercise if would be of interest to write functions in Python that emulate the functionality of these verbs and then compare the advantages and disadvantages of the languages.

```
NB. ============================================================
NB. (Qx)stocha(Age;S)
NB. Qx = mortality table – commencing at age 0
NB. Age = age of life (integer)
NB. S = number of simulation
NB. (ALT_2015_17_M)stocha(65;100000)
stocha =: 4 : 0
Qx =. x
'Age S' =. y
rnd01 =. ?.@$&0           NB. Random number generator (tacit)
QAge =. Age}.Qx           NB. Mortality table from Age
RND =. rnd01 S,#QAge      NB. Random numbers table
*/\"1 QAge<"1 RND         NB. Survival periods
)
```

12 Appendix 4 – Comparison of J and Python

As an exercise I have compared the code for the life annuity with payments in arrears using:

- J as a tacit
- J as an explicit
- Python

The calculation reads a mortality table, discount rate and age.

12.1 J Tacit

The J tacit code for ax is shown in Appendix 3 and example in section 5.3
Further examples assuming that mortality table has been loaded.

```
   (ALT_2015_17_M;0.02)ax(0 10 20 30 65)
39.1876 37.0323 34.2887 31.0864 15.2981
```

12.2 J Explicit

The J explicit code for tacit ax can written without loops :

```
NB. ============================================================
NB. ax defined as an explicit
NB. (ALT_2015_17_M;0.02)ax_E(65)
NB. (ALT_2015_17_M;0.02)ax_E(0)
NB. (ALT_2015_17_M;0.02)&ax_E each (0 65)
NB. -------------------------------------------------------
ax_E =: 4 : 0
'Qx E' =: x
Age =: y
T =: # NPX =: */\ Age}. (1- Qx)
V =: % (1+E)^ 1 + i. T
+/ V * NPX
)
```

12.3 Python

The Python code was written by finance students from as part of a project. The students were leaning Python as part of the project and therefore the code is subject to review and change. The mortality is read from an excel file.

```
def annuitycalculator3(client_age,client_gender,investment,i):
  #the calculator is that knowing investment, want to get payment each year
    import math
    import numpy as np
    import pandas as pd
    import matplotlib.pyplot as plt
    v=(1/(math.exp(i)))
# i-discounted rate  v-discounted factor
    summ=0

    df = pd.read_excel('Au_life_table_2015-2017.xlsx')

    age = df['Age'].values.tolist()
    n=np.size(age)-client_age-1
    c_npx=np.zeros(n)
    vi=np.zeros(n)

    if client_gender==1:  #1-male 0-female
        m_lx=df['m_lx'].values.tolist()
        m_qx=df['m_qx'].values.tolist()
        m_dx=np.diff(m_lx)*(-1)


        for i in range(1,n+1):
            c_npx[i-1]=(m_lx[client_age+i]/m_lx[client_age])
            vi[i-1]=(v**i)

    elif client_gender==0:
        f_lx=df['f_lx'].values.tolist()
        f_qx=df['f_qx'].values.tolist()
        f_dx=np.diff(f_lx)*(-1)


        for i in range(1,n+1):
            c_npx[i-1]=(f_lx[client_age+i]/f_lx[client_age])
            vi[i-1]=(v**i)


    summ=c_npx*vi
    lumpsum=sum(summ)
  #lumpsum=investment at client age to ensure getting $1 each year
    payment=investment/lumpsum
  #payment indicates annual payment annuitant can get

    return payment
```

D:\library\Finexec_info\Note\Finexec_J_Programming_0.docx