Principles of the model:

1. computational efficiency
2. simplicity
3. versatility

Description of the model

1. stochastic
2. balance sheet approach
3. discrete
4. multi-period
5. Actual scenarios instead of risk-neutral
6. with management actions
7. phase 0
   1. General Model
8. set up the main structure of the model

I have found a tool called “ecto”, which may be useful. Whether to use it or not depends on whether it is computational efficient.

<http://plasmodic.github.io/ecto/ecto/>

The general model should have the following things:

1. cell (called Variable in the project. Is it better to use the names in “ecto” for our project?)

Our Variable (cell) class should have the following functionality:

1. have member variables recording different properties of the cell. For each cell, pointers (or names) of its precedents and dependents may be included. It should have an indicator showing how the caches are reset (reset upon calculation of each record; or reset upon certain criteria (using CriteriaCell metioned below); or reset upon management actions taken (see below for management actions); or never reset)
2. have caches s.t. it can store multi-dimensional results. Most of the time, the caches will be deleted (reset) at the end of calculation of each record.
3. InputCell

A special type of cell. It has the ability to read data from “Input”/”GlobalSettings”, or OutputCells of other plasms (see below for plasm definition). e.g. dCURR\_POLICY in mVarNameSpace.

1. RandomCell

A special type of cell. It will generate random numbers. These numbers should be recorded and finally stored in local drive.

1. CriteriaCell

A special type of cell. Its return value is always boolean or enum. It is used to check which type does the record belongs to. (e.g. does the policy belong to male or female; the name of insurance product the policy belongs to) e.g. “bTemp” written in the file “script\ scrOutputCriteria.txt”. CriteriaCell can also utilize “AccumTable”, e.g. whether certain policy record’s product name belongs to an Accumulation (a group of products).

1. OutputCell

A special type of cell. This type of cell will not reset upon calculation of each record (it is accumulative). The dimension of its cache has to utilize those criteria set up by CriteriaCells. (e.g. the cache may be a 2-dimensional matrix, with 1 dimension male/female, the other dimension different product names.) Upon calculation of each record, the results in OutputCells will be updated. (You may refer to OutputFormatTable and ‘test\_output\out1.txt”). The dimension of cache of an OutputCell may use one of the OutputFormats, or be set up manually.

One of the dimensions of a typical OutputCell should be time.

1. plasm (directed acyclic graph of cells)
2. Auto-detection of precedents and dependents for all cells within the plasm. In this way, the graph is set up automatically.
3. make sure the graph is not cyclic
4. The final input of a plasm should always be “Input”, ”GlobalSettings”, or OutputCell of other plasms.
5. Important cell values should be able to be recorded upon request. e.g. The names of key cells that need to be stored are listed in a local variable called “lsVars” in mVarNameSpace.fRunModel() function
6. A complicated model may have several plasms joining together. The following is an illustration:

“AA” is a plasm which contains an OutputCell called “output” .

“BB” is also a plasm which contains an OutputCell called “output”, which is different that that of “AA”.

“CC” uses the “output” values from both “AA” and “BB” as inputs. It contains a InputCell called “add” which is a function of values of “output” of “AA” and “BB”. The caches of “add” may be multi-dimensional.

1. For this project, an example of plasm is a certain type of insurance products with similar characteristics. Policy records for these products are loaded, calculated and summarized (aggregated) in OutputCells.
2. It is possible to run different policy records parallelly (using numpy/numba thing I guess?) ; it is also possible to run different scenarios (see 1.1.(5)) parallelly
3. It is possible to set up a thread for each plasm, or even set up several threads for a plasm that contain a lot of policy records, and run parallelly (whether to do this depends on whether it is easy to do and computationally efficient)
4. modules (not python modules. This “module” refers to “a collection of cells, and manifests itself as a single shared object”. In our project, it is “Script::BODY”, but the name/practice may not be that good. How should it be called in our project?)

Several modules can be loaded in a plasm.

1. Input

the “Input” Class should be able to do the following:

1. Read data from all tables. There are several types of tables
2. Tables that contain different records (eg. MPFTable). Every row represents a record for a particular policy or a particular asset.
3. Tables that contain assumptions. (eg. GenTable/KeyGenTable). We should be able to obtain the values easily if keys are provided. There are different types of assumptions:
4. BE(best estimate, i.e. average) assumptions for asset/liability
5. assumptions for shocks (e.g. volatility for interest rates)
6. Tables that contain information for run settings. e.g. GlobalSettings:: odDimVarRanges
7. Tables that contain information for output format (eg. mOutputFormatTable
8. Tables that contain information for “which variables’ data will be included in the reports”. (mReportVarTable)
9. Record Random Generator

As we do not have policy records and asset records, we have to generate them randomly currently.

1. Input
2. different field names (names of different values in a record) (e.g. [“sex”,”product\_name”,”policy\_number”])
3. possible values for each field (column) (eg, “f”/”m” for sex) (these values may have a particular variable type, like “string”, “int” or “double”)
4. the distribution (possibility) of each possible value for each field (eg. Bernoulli distribution (0.4) for “sex”)
5. how much records are needed
6. how are these records be stored (e.g. we want one single file holding all the records for each sex. Therefore, in the output folder, there are 2 files, whose file name are “male” and “female” )
   1. Liability side
7. set up excel models for different insurance products (deterministic)
8. Different approaches of valuation
9. Statutory reserve
10. Cash flow
11. Embedded value
12. Fair Value (use risk-free rate, if time allows)
13. different products
14. Whole Life
15. Simple CI
16. Universal Life
17. Unit link (if time allows)
18. deferred annuity (if time allows)
19. term life (if time allows)
20. different assumptions
21. payment mode: only annually
22. mortality rates
23. interest rates
24. expense
25. commissions
26. inflation rate (may not be needed)
27. assume no reinsurance
28. lapse rates
29. bonus rate
30. dynamic behaviour
31. lapse rate will increase when stock return rate/competitor’s guarantee interest rate goes up
    1. asset side
32. set up excel/python/other language models for different assets (deterministic)
33. Different approaches of valuation
34. Statutory reserve (accounting)
35. Fair Value (market values)
36. Fair Value (PV of future cash flows)
37. different asset types
38. bond (callable or not)
39. share (preferred or not)
40. property
41. different assumptions
42. interest rates
43. expense
44. inflation rate (may not be needed)
45. correlations (ratings)
46. default rates
47. dividend rates
    1. management
48. management actions for different situations, including
49. buy/sell any assets
50. raise/reduce bonus rates
51. buy/sell options
52. currently, new business will not be considered
53. dividends for shareholders/ raise debt/ capital injection
54. assumptions for management actions

i . Management actions are taken at time t based on deterministic results at time t

1. phase 1
   1. General Model
2. integration with database (read/write)
3. Visualization of relationship of Cells in a certain plasm. (e.g. for a specific cell, show visually its precedents and descendants, when click one of its precedents, then set focus on the clicked cell ).
4. Different parts of the model can be stored. (i.e. we can create a new table, provide all the data it needs and store it in our local drive. We can also change a certain table and overwrite it in our local drive.)
5. Visualization of reports (charts).
6. Think of a way to incorporate management actions into the model. The workflow should be:

i . Load Inputs

ii . calculate liability/asset with given information at time t (eg, calculate the items in the balance sheet at time t)

iii . make management actions (eg buy/sell asset) based on the results at time t

iv . generate random variable for time t+1 (eg, generate the forward rate for time t+1)

v . if t is not the end of the study period (eg, 100 years), t=t+1, go to step ii; else, produce reports for the model.

vi . Repeat processes i to v and produce results for different random variables generated

2 tricky problems:

i . When should the caches be reset

ii .Which cells’ values should be stored at time t for calculation of asset/liability (step ii) after the management actions

1. Solve problems arisen from “liability and asset side”
   1. liability side
   2. write the scripts for the model, s.t. the results of the model are the same as that shown in Excel worksheets.
   3. test the running time for the model.
   4. It is most likely that the running time is too much. Model points have to be established (each model point is a group of policies with similar characteristics). Compare the results using model points with results using individual policy records.
   5. asset side
2. write the scripts for the model, s.t. the results of the model are the same as that shown in Excel/Python model.
3. test the running time for the model.
4. phase 2
   1. General Model
5. GUI for the model (if time allows)
6. API for excel (if time allows)
7. multi-processing (if time allows)
   1. liability/asset/management
8. Incorporate management actions into the model. Combine everything together.