# THE SIMULATOR OF INDIVIDUAL DYNAMIC DECISIONS, SIDD

*Introductory Programming Guide*

J. van de Ven
18 July, 2016

This document provides a broad overview of the SIDD program, and a step-by-step introduction to amending the code. No formal programming experience is assumed.

# The Simulator of Individual Dynamic Decisions, SIDD

*User Manual*

## Table of Contents


1 Introduction ..... 2

2 Overview of the Program ..... 2

2.1 Program structure ..... 3

3 Set-up ..... 12

3.1 System requirements ..... 12

3.2 Loading the model onto a new computer ..... 12

3.3 Creating a simulation base ..... 13

4 Extending the Program ..... 15

Introducing health status into the model ..... 15

4.1 Defining and storing model parameters ..... 15

4.2 Reading in new model parameters ..... 16

4.3 Altering model grids to accommodate new characteristic ..... 16

4.4 Alter expectations for utility maximisation ..... 18

4.5 The direct influence of health on simulated decision making ..... 20

4.6 Including health status when simulating a population through time ..... 20

4.7 Debugging ..... 20

Appendix A: Basic Rules of Syntax ..... 22

Appendix B: The Utility Function ..... 24

# 1 Introduction

This document provides an overview of programming code that underlies the *Simulator of Individual Dynamic Decisions* (SIDD). SIDD is designed to implement a structural model of dynamic decisions, focussing predominantly on the consumption/savings and labour/leisure margins. The structural model upon which SIDD is based assumes that individuals make their decisions to maximise the expected lifetime utility of their respective benefit units. Importantly, SIDD is designed to solve lifetime decisions, taking into account uncertainty associated with a wide range of characteristics.

The SIDD program is an advanced implementation of the dynamic programming models that are increasingly popular in the economics literature. Unfortunately, any individual who would like to contribute to this literature is currently faced with the massive technical over-head involved in developing the programming routines that are necessary to implement the analytical approach. This manual is written in the hope that progress in the related field might be accelerated by making the source code that underlies SIDD accessible to a wider group of individuals.

Consistent with the above stated purpose, this manual does not discuss at length technicalities that underlie the program's implementation. Rather, the emphasis here is on supporting an individual to amend the program code to consider issues that cannot be described by the existing model structure. Anyone interested in technical details that underlie the program are referred to **. Similarly, directions in practical use of the program in its existing form are provided in the *User Manual*, which is a companion manual to this paper. The discussion that follows assumes that the reader is familiar with each of these two publications.

This document has been kept deliberately brief to facilitate accessibility, with discussion divided into 3 sections. Section 2 provides an overview of the program structure, including a discussion of the sorts of problems that it is currently well adapted for. Section 3 describes how to set the model up on a new computer, ready for amending the programming code. Section 4 then describes how the existing program code can be altered to extend the sorts of problems that it can be used to explore, with reference to a practical example.

# 2 Overview of the Program

The SIDD program includes all of the files and folders necessary to simulate a population cross-section forward and back in time on the basis of the assumed structural framework. Throughout this document, SIDD is referred to as a "program" and not a "model" because our current interest is confined to issues pertaining to the program code. Implementations of this code to specific social contexts include the NIBAX model (a cohort specific specification for the UK, e.g. van de Ven, 2011), the PENSIM model (a cohort specific specification for Ireland, e.g. Callan *et al.*, 2009), and the LINDA model (a cross-sectional model for the UK, e.g. ***). Projections derived from each of these models are generated by the SIDD program.

SIDD is comprised of four types of file. Key model parameters and summary statistics are contained in a series of Excel files, to aid their manipulation. Other model parameters are saved in text files, denoted with a *.dat extension. All calculations are performed by a master executable file called SIDD.exe, which delegates selected tasks to a series of dynamic link libraries (*.dll). Both the executable file, and the dynamic link libraries are programmed in Intel Visual Fortran, which has the

advantages of being computationally efficient and accessible to people with little previous programming experience. This document provides a brief guide to altering SIDD.exe and the associated *.dll files.

## 2.1 Program structure

*The principal technical programming details that are assumed by the subsequent discussion are summarised in Appendix A. It is recommended that you read Appendix A before proceeding if you have little (or no) formal programming experience.*

The programming code takes a modular form, and makes extensive use of programming subroutines. Although it is beyond the scope of this document to provide a detailed description of the workings of the program, this section provides a basic schematic of how the principal subroutines of the program relate to one another, and what each subroutine does.

A full list of the subroutines of the SIDD program is reported in the table presented below. Figure 1 provides a pictorial schematic of how a typical simulation will run through the program code.

The model enters through the "MASTER" program module. MASTER passes first to the LOAD_DATA module, which loads in model variables with calls to the library EXCEL.dll.

LOAD_DATA creates the subdirectory into which simulation output for the current run is saved. It also allocates model parameters to a series of global variables (held in the GLOBAL_M and GLOBAL_PARAMSTORE modules), a process that is partly undertaken through calls to the COHORT_ALLOCATE subroutine. The model parameters are then checked by a call to the CHECK_DATA subroutine, and any errors are reported by calling the REPORT_ERRORS subroutine.

MASTER then passes the program through to the GRIDS_MASTER subroutine, which organises calls to each alternative part of the model, depending upon the parameters that have been set-up for the simulation. A typical simulation will initially involve the following.

1) AXES_INIT will define the terms of the "grids" into which the state space of the considered decision problem is divided
2) GRIDS_EVAL will organise the total storage space for outputs associated with the grids into a series of segments, which are then passed to SOLVE_GRID
    a. The model is set-up to permit all of the output to be saved into a single file (1.dat), which is stored as a global variable throughout a single simulation. This helps to reduce computation times, by limiting reading-writing to hard disk space. In context of memory limitations, however, grid output can also be chopped up into a series of files, each of which will span a given number of "ages". The calculations necessary to manage these details are performed here.
3) SOLVE_GRID organises the terms of the utility maximisation problem associated with each age, and passes these to TM_GRID_CAL. It then re-packages output received from TM_GRID_CAL for passing back to GRIDS_EVAL.
4) TM_GRID_CAL organises treatment of the optimisation problem at each grid intersection. This subroutine first identifies a number of descriptive characteristics of the optimisation problem. It then considers the optimisation problem of each grid abscissae in turn. Three loops are used to loop over grid abscissae. The outer loop considers each possible discrete

state. The second loop considers each segment defined for net liquid wealth, and is parallelised using OpenMP. The third loop considers each alternative continuous state variable. Within the "state loops" referred to above, are a series of loops that are designed to consider each potential decision alternatives between discrete options (e.g. full-time/part-time/not employed), and within these "decision loops", is a call to CONTINUOUS_CHOICE, which optimises over continuous control variables. Finally, TM_GRID_CAL packages calculated output, and returns it to SOLVE_GRID.

5) CONTINUOUS_CHOICE evaluates which continuous control variables require a solution. It then initialises expectations vectors by calling EXPECTATIONS_UPDATE, identifies maximum bounds on consumption via CONSUMPTION_BOUNDS, and passes the optimisation that is now defined through to the routines in the OPTIMISATION module
6) After solving the lifetime decision problem under GRIDS_EVAL, the model passes to SIM_POPULATION to simulate a reference population through time. SIM_POPULATION works by looping over ages, starting with the highest simulated age and working backwards to conduct the backward simulations, and then switching to start with the youngest simulated age and working forwards for the forward simulations. The requirement to satisfy incentive compatibility conditions complicates backward simulation, relative to forward simulation.
    a. Backward simulations start, for each individual, by initialising states at age *aa* from *aa+1* via STATE_AA_INIT. The decisions for the respective individual at age *aa*, given their initialised state variables, are evaluated by DECISION_EVAL. Given an individual's state variables at age *aa+1* and projected decisions at age *aa*, PROJ_STATES_BACKWARD then projects states back to age *aa*. A loop is then used to identify a combination of states and decisions at age *aa* that are mutually consistent with the individual's states at age *aa+1*.
    b. Forward simulations require only a single call to DECISION_EVAL and then a call to PROJ_STATES_FORWARD, for each individual at each age.
7) The model then passes to secondary analysis routines, typically controlled through ANALYSIS_MASTER before returning to the MASTER program and exiting.

**Table 2.1: Model components**

| File | Module | Subroutine | Task |
|---|---|---|---|
| 1_Master.F90 | NA | NA | *main program entry and exit point* |
| 1a_tester.F90 | NA | NA | NA |
| 2_data_input.F90 | data_input | | *loads in and re-packages model parameters* |
| | | load_data | loads in model parameters |
| | | check_data | checks the loaded parameters for obvious inconsistencies |
| | | report_errors | reports any identified errors to the main program window |
| | | cohort_allocate | re-packages model parameters to the format required for the simulation |
| | | update_xArr | NA |
| 2a_global.F90 | | | *defines model parameters passed as global variables* |
| | global_m | NA | global variables storing a sample of definitional terms |
| | global_paramstore | NA | global variables storing model parameters |
| 2b_simdata.F90 | simulated_data | | *defines global variables that store simulated panel* |
| | | NA | global variables storing simulated panel data |
| | | deallct_sim_data | deallocates variables that store simulated panel data |
| | | load_simpop | allocates variables that store simulated panel data and loads in data for base population |
| 3_model_manage.F90 | grids_calc | | *high-level management of routines that are called by a given simulation* |
| | | grids_master | a master subroutine that manages which other subroutines the model subsequently calls |
| | | geq_eval | search routine to identify endogenous prices and returns for steady state solution to a General Equilibrium |
| | | intertemp_elas | calculates intertemporal elasticity by perturbing rates of return |
| | | Delta_lc | sub-component used to evaluate intertemporal elasticity |
| 3a_axes_init.F90 | axes_init | axes_su | *initialises the axes of the grids that are used to solve the lifetime decision problem* |
| 3d_estimn.F90 | model_estn | | *experimental module for estimating the model structure using the method of simulated moments - not discussed further here* |

*segment 1 of 6*

**Table 2.1: Model components (continued)**

| File | Module | Subroutine | Task |
|---|---|---|---|
| 3e_budget_balance.F90 | budget_balance | | *conducts a search to identify taxes that ensure budget balance between consecutive simulations* |
| | | budget_search | search routine for taxes |
| | | budget_call | evaluates budget balance for given tax rates |
| 4_grids_manage1.F90 | grids_comp | grids_eval | *highest-level of subroutine for solving utility maximisations - cycles over grid components* |
| 4a_global.F90 | | | *global variables used to solve the utility maximisation problem* |
| 4b_grids_manage2.F90 | solve_grids | solve_grid | *second highest level of subroutine for solving utility maximisations - cycles over individual ages* |
| 4c_solve_master.F90 | utility_maxn | | *solves utility maximisation problem for individual grid abscissae* |
| | | tm_grid_cal | identifies existing state combination to assume for each utility maximisation problem |
| | | inner_state | evaluates existing values of continuous state variables |
| | | emp_training_id | calculates employment, leisure, and training combinations for individual (discrete) decision options |
| 4d_abscissae.F90 | abscis | abscissae | *determine weights and abscissae of Gauss-Hermite quadrature* |
| 4e_exp_prelim.F90 | exp_prelim | | *evaluates expectations over discrete state alternatives* |
| | | cont_prelim | initialises global variables in relation to continuous states |
| | | disc_prelim | initialises global variables in relation to discrete states |
| | | axes_prelim | defines axes of grids for current solution problem |
| | | disc_states | defines existing discrete state variables |
| | | disc_exp | defines expectations concerning discrete state variables |
| 4f_search_allcontinuous.F90 | solve_continuous | continuous_choice | *defines the terms required to solve over the continuous control variables, and passes the problem to generic optimisation routines* |

*segment 2 of 6*

**Table 2.1: Model components (continued)**

| File | Module | Subroutine | Task |
|---|---|---|---|
| 4g_valfn_evals | val_fn_evals | | *undertakes calculations necessary to evaluate value function calls* |
| | | expectations_update | determines whether it is necessary to update expectations with reference to tax function calls |
| | | expectations_adjust | updates expectations using tax function calls |
| | | consumption_bounds | determines bounds on consumption following update of expectations using tax function calls |
| | | val_eval | evaluates value function |
| 5_popn_genr.F90 | popn_genr | | *simulates population characteristics forward and backward through time* |
| | | sim_population | master subroutine that initialises population variables and passes model to subroutines that project these characteristics through time |
| | | state_bounds | defines maximum and minimum values for state variables |
| | | decision_eval | uses interpolation methods to identify decisions given a benefit unit's prevailing state variables |
| | | proj_states_forward | projects individual characteristics forward one period, given their simulated decisions and characteristics in the prevailing period |
| | | ws_state_init | initialises "working space" used to model retirement status back through time |
| | | ws_state_update | updates the "working space" used to project retirement status back through time to ensure incentive compatibility |
| | | state_aa_init | initialises characteristics for age aa, when projecting backward from age aa+1 |
| | | proj_states_backward | projects individual characteristics backward one period, given their simulated decisions and characteristics in the prevailing period |
| | | hh_size | projects demographic characteristics forward one period |
| | | hh_size2 | updates demographic characteristics using stored lifetime vectors |
| | | self_emp_innov | updates innovations to match self-employment status to observed status in reference period |
| | | rp_dataload | loads in reference population panel data |

*segment 3 of 6*

**Table 2.1: Model components (continued)**

| File | Module | Subroutine | Task |
|---|---|---|---|
| 5b_sim_globals.F90 | sim_vars | | *global variables for simulating population through time* |
| 6_analyses.F90 | internal_analyses | | *a set of subroutines that conduct secondary analyses on simulated panel data* |
| | | analysis_master | master subroutine to determine which analysis routines are accessed |
| | | analysis_calibn | master subroutine to determine which set of statistics to produce to calibrate the model parameters |
| | | analysis_adult | calibration statistics when adults are explicit - generates results reported through excel file calibn.xls |
| | | analysis_hilevel | analysis routine to produce "high level" statistics reported in excel file hi_level_statistics.xls |
| | | ge_sum_stats | generates a subset of the statistics reported in the file hi_level_statistics.xls |
| | | analysis_deciles | generates decile level statistics reported in file analysis_dec.xls |
| | | cv_analysis | generates Compensating Variations reported in file analysis_dec.xls |
| | | tax_analysis | generates statistics for analysis of tax function, as reported in tax_test.xls |
| | | tax_analysis2 | generates statistics for analysis of tax function, as reported in tax_test2.xls |
| | | income_moments | generates summary income moments reported in income_moments.xls |
| | | DA1 | generates summary statistics reported in file DA1.xls |
| 6a_analysis_tools.F90 | analysis_tools | | *a set of routines that are useful for analysing data* |
| | | min_max_sort | generates a vector that reports indices in ascending order of an input vector y |
| | | var | generates the variance of an input vector y |
| | | weighted_mv | generates the weighted mean and variance of a vector of input data vec |
| | | cum_norm | function for calculating the cumulative normal probability |
| | | inv_cum_norm | function for calculating the inverse of the cumulative normal distribution |
| | | zero_invNorm | search routine used to calculate the inv_cum_norm distribution |
| 6b_sorting.F90 | m_mrgrnk | | *a set of routines that are useful (and efficient) for sorting vectors of numbers* |

*segment 4 of 6*

**Table 2.1: Model components (continued)**

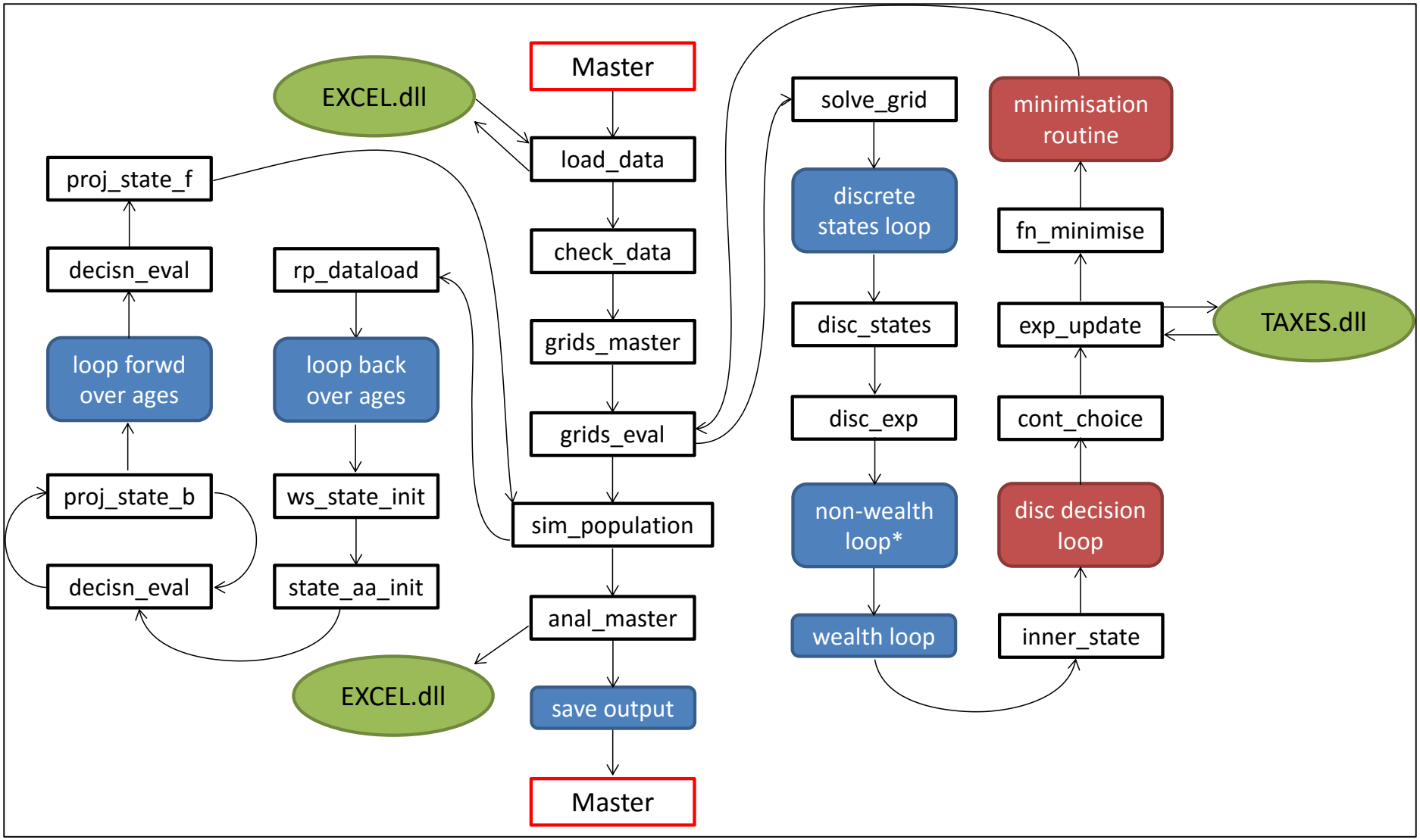
| File | Module | Subroutine | Task |
|---|---|---|---|
| app_BusDekker.F90 | BusDekker | | *Implements the Bus Dekker algorithm to search for a zero of a function* |
| | | zero_search | the Bus Dekker algorithm |
| | | fn_zero | a passing module for defining the function to be set to zero |
| app_Interpolation.F90 | Interpolation | | *a set of subroutines to interpolate over multiple dimensions* |
| | | interpln | undertakes some basic checks before passing an interpolation problem on |
| | | int_disgrid | stripping out interpolation over discrete states |
| | | interp_pass | passes to interpolation, and then conducts extrapolation if necessary |
| | | interp | passes to linear or cubic interpolation methods |
| | | interp_cubic | cubic interpolation |
| | | interp_lin | linear interpolation |
| | | interp_near | nearest neighbour interpolation |
| | | indxoff | working subroutine for identifying off-sets |
| | | getinteger | converts values to integers |
| app_optimisation.F90 | optimisation | | *Contains a series of routines that are designed to optimise over multiple continuous dimensions referring only to value function calls* |
| | | fn_minimise | passes the optimisation problem to the relevant routine - NAG routines are suppressed and can only be accessed if you have the necessary 3rd party license |
| | | objfn | defines the objective function for minimisation |
| | | objfn2 | implements adaptations required by a NAG routine |
| | | objfn3 | implements adaptations required by a NAG routine |
| | | banana | a standard test problem |
| | | powell | implements the Powell search routine, as described in Numerical Recipes |
| | | brent | implements the Brent search routine, as described in Numerical Recipes |
| | | boundary_pnts | identifies boundary points of linear projections through N dimensions |
| | | chk_bound | runs a check for boundary conditions on a specified linear projection |
| | | fminsearch | implements the simplex search method of Lagarias et al. |

*segment 5 of 6*

**Table 2.1: Model components (continued)**

| File | Module | Subroutine | Task |
|---|---|---|---|
| app_other_tools.F90 | other_tools | | *a miscellaneous set of analytical routines* |
| | | time_given_agei_cohort | defines a series of temporal identifiers given age and birth cohort reference number |
| | | age_given_time_cohort | defines a series of temporal identifiers given time and birth cohort reference number |
| | | pension_exist | identifies whether a pension could exist for an individual of given circumstances |
| | | axis_defn | defines axis identifiers |
| | | randnorm | generates a random draw from a normal distribution, based on a random draw from a uniform distribution |
| | | employment_id | identifies employment status for individuals based on other characteristics |
| | | csv_file_write1 | writes output to disk in CSV format |
| | | get_unit | NA |
| | | csv_data_append | appends new data to a CSV output file |
| | | i4_widths | NA |
| | | i4_log_10s | NA |
| | | hh_demog_annual | generates demographics from transition probabilities and random draws |
| tax_coms.F90 | tax_coms | | *constructs vectors for communicating with dynamic link library files that compute tax and benefits outputs* |
| | | tax_prep | packs the vector to send to TAXES.dll |
| | | tax_retn | unpacks the vector retrieved from TAXES.dll |
| | | part_tax_rate | calculates participation tax rates |

*segment 6 of 6*




Note: * denotes parallelised loop using OpenMP. Model entry and exit in red "Master", looping conditions in blue, decision optimisations in red, communication with dll libraries in green

**Figure 2.1: Schematic of program code**

# 3 Set-up

## 3.1 System requirements

SIDD is designed to operate on desktop workstations that use Intel processors and the Microsoft "Windows" operating system. The model will run on minimal system specifications: a 32 bit operating system, with a single core processor, and 4 GB of RAM. The types of problem that the model can be used to explore are very limited on this minimum specification, however, and we recommend the following system specifications: 64 bit operating system, with 12 physical cores, 24GB of RAM and 1TB of hard disk space.

Microsoft Excel is required to analyse summary statistics reported by the model. The model is programmed in the Intel Visual Fortran environment, currently available in the form of *Intel Parallel Studio XE*, or *Intel Fortran Studio XE*.

## 3.2 Loading the model onto a new computer

The model is delivered as a single zip folder. The folder includes two subdirectories: FORTRAN, and MODEL. The FORTRAN subdirectory includes the programming code for the ANALYSIS, EXCEL, SIDD, and TAX routines of the model. The MODEL subdirectory includes all of the files that are required to run the model.

The MODEL subdirectory contains two subdirectories in addition to a set of model files. The subdirectory BASE_FILES contains a separate subdirectory for each "base" specification that you create with the model in which files that are required for the respective base specification are stored (as discussed in the section concerned with "FORM 0" that we return to below). The subdirectory SIMULATIONS will contain a separate subdirectory for each simulation that you run, in which are stored the panel data generated by the model, model parameters, and excel simulation output.

Please follow these steps when installing the model on a new computer:

1. Extract the zipped files from the compressed folder to a subdirectory of your choosing, maintaining the directory structure that we have included with the zipped file
2. In the FORTRAN subdirectory, open up the EXCEL subdirectory, and double-click on EXCEL.sln
    a. This should open the Visual Studio program environment
3. If you can see the "solution explorer" window, then select the purple box
    a. If you cannot see the window, then open it through the "View" drop-down menu
4. In the "Project" drop-down menu at the top of Visual Studio, select "Properties"
5. In the "Configuration" drop-down menu select "All configurations"
6. In the "Platform" drop-down menu select "All platforms"
7. Under the "Configuration Properties", select the "General" category
8. Against the "Output Directory", enter the file location that you have saved the model into
    a. eg: "c:\myfiles\model_lab\model\"
9. Under the "Configuration Properties", select the "Debugging" category
10. Against "Command", enter the location of the file "SIDD.exe"; eg: "C:\MyFiles\MODEL_LAB\MODEL\SIDD.EXE"

11. Against “Working Directory”, enter the same text as under (8)
12. Press the “Apply” button, and then the “Ok” button
13. Under the “File” drop-down menu select “Save All”
14. Under the “Build” drop-down menu select “Configuration Manager”
15. Under the “Active Solution Configuration” select “complete”
16. Under the “Active Solution Platform” select “x64” and press the “Close” button[1]
17. Under the “Build” drop-down menu select “Rebuild Solution”

You should then see some text like:

```
1>------ Rebuild All started: Project: EXCEL, Configuration: Complete x64 ------
1>Deleting intermediate files and output files for project 'EXCEL', configuration
'Complete|x64'.
1>Compiling with Intel(R) Visual Fortran Compiler XE 14.0.1.139 [Intel(R) 64]...
1>CSV_IO.f90
1>comEXCEL.f90
1>0_dll_entry.f90
1>Linking...
1>Starting pass 1
...
1>     ImageHlp.lib(imagehlp.dll)
1>     ImageHlp.lib(imagehlp.dll)
1>Finished pass 2
1>
1>Build log written to
"file://C:\MyFiles\MODEL_LAB\FORTRAN\EXCEL\x64\Complete\BuildLog.htm"
1>EXCEL - 0 error(s), 0 warning(s)
========== Rebuild All: 1 succeeded, 0 failed, 0 skipped ==========
```

18. Redo steps (2) to (17) for the remaining model components, in the following order:
    a. TAXES
    b. ANALYSIS
    c. SIDD

The above ensures that the programming files are all present and working

## 3.3 Creating a simulation base

The model starts with data reported by the Wealth and Assets Survey for a population cross-section of reference adults. The model parameters have been calibrated to match the model to a wide range of summary statistics calculated from survey data sources, with the calibration structured around the year in which the reference population was observed (see van de Ven and Lucchino, 2013, for details). The model comes packaged, ready to project the circumstances of the population cross-section forward and backward through time, to build up a complete life history for each reference individual. It is recommended that this be done, and that the associated data should be defined as the “base” for subsequent simulations. This can be done by following the steps 19 to 29 below

19. Open the MODEL subdirectory, and then open “job file.xls”
20. Ensure that you allow macros to work in Excel
    a. please ask your system administrator if you require assistance with this

[1] If running a 32 bit environment then select “Win32” here.

21. Press ALT+F8
22. Select “SIDD” and press the RUN button
23. Press the “RUN EXISTING JOB FILE” button

This version of the model currently runs in around 9 hours on the recommended system defined in Section 3.1 – the associated simulation creates a new set of base data for analysis, using the full model specification.

24. Re-open “job file.xls”
25. Press ALT+F8
26. Select “SIDD” and press the RUN button
27. Enter “age18_all” into the text-box with the title “name of run to adopt as new base”
28. Enter “base_2006_age18_all” into the text-box with the title “directory name for new base”
29. Press the “CONVERT RUN TO NEW BASE” button

Excel will then work away for a short while, after which you should receive a message confirming that the new base has been created. If you look in the “base_files” subdirectory, you should now see a new subdirectory with the name “base_2006_age18_all”, which includes all of the files defining the base simulation specification.

To test that the model set-up has been successfully completed:

30. Re-open “job file.xls”
    a. This file has been changed since step (24), so that it now references the new base model directory by default.
31. Press ALT+F8
32. Select “SIDD” and press the RUN button
33. Press the “SET UP NEW SIMULATION” button
34. Type “test” in the text-box with the title “Simulation Name”
35. Press the “ENTER” button
36. Tick the box to direct the model to calculate “statistics for equivalised income deciles”
37. Tick the box to indicate that “comparative statistics with the population base” should be evaluated
38. Press the “ENTER AND RUN” button
39. Press the “LAUNCH MODEL” button

The model should then run through once again, in around 6-7 hours. This time, however, the simulations will project only forward through time, taking the population characteristics back in time from the base specification”. When the model is complete, please open the analysis_dec.xls file that is created in the “test” simulation directory, and check that all of the statistics reported in the “differences with base” sheet are close to zero.

# 4 Extending the Program

This section provides a very brief overview of the steps that are involved when introducing a new feature into the model structure. There is no substitute for "hands-on" experience in this regard, and you should allow plenty of time for experimentation in your first few attempts. Although the model structure has been specifically adapted to facilitate the introduction of new features, the structure is complex, and it is therefore exceedingly easy to introduce errors into the framework. Furthermore, once errors have been introduced, they can be very difficult to identify at a later date, and can contaminate simulation results in highly unexpected ways. It is therefore essential that you exercise extreme care whenever adjusting the program structure.

The steps involved to include a new feature in the model, depend upon the feature of interest. Features become increasingly difficult to implement when they alter the uncertainty that people face, or the decisions that they have available. To cover all of the potential issues that you might encounter is well beyond the scope of this introduction. We consequently focus here on a realistic – albeit fairly simple – example. The objective of providing this example is to provide the reader with clues about where they should look in the existing code to guide them in the implementation of more complex structures.

### Introducing health status into the model

Suppose that you wanted to introduce a new characteristic to represent the health status of a reference person into the model. We first need to define what "health status" is to mean in the model. To keep things simple, suppose that we are only interested in two alternative health states (good and bad), where transitions between them are random, and the existing state influences non-discretionary benefit unit expenditure.

Introducing this type of model characteristic involves 7 key steps:

1) Save the existing model structure as a "Draft Model"
2) Define model parameters and store them in associated excel files
3) Read model parameters into the program
4) Alter model grids to accommodate the new characteristic
5) Alter expectations for utility maximisation problem
6) Implement the direct influence of the new characteristic on the decision problem
7) Allow for the new characteristic when projecting the population through time
8) De-bugging model alterations

The first of these steps is important to ensure against some form of critical error during the re-programming process, and to provide a sound basis for comparison during the debugging process. Each of the alternative steps of the analysis is discussed in turn.

## 4.1 Defining and storing model parameters

Suppose that there are 10 model parameters that the model will require to simulate our definition of health status. The 10 parameters could be added to the end of the existing parameters in column AS of sheet "input" in file "Job File.xls", along with descriptions in column AR.

TIP: make sure that you save the new model parameters into both the "input" and "inputA" worksheets of "job file.xls" when you are finished. Otherwise, you risk losing the parameters when someone runs the model macro in future work.

Note that storage of model parameters is not always so straight-forward. It may require you to add worksheets to existing input files, or add new Excel files entirely. This task can consequently become fairly complex, despite the simplicity of the example provided here. You are advised, in complex cases, to look for existing parameters that are similar to those that you are interested in adding, and use the associated code to infer how the task could be dealt with.

## 4.2 Reading in new model parameters

Scrolling to the top of the "job file.xls", you will note that there are comments in the first row of each column of variables stored in the "input" sheet. These comments define the global variables into which the relevant parameters are stored by the model – "cohort_par" in our case. We now need to ensure that our new parameters are loaded into cohort_par.

Open the solution for SIDD in the Visual Studio programming environment. Open file "2_data_input.F90", and ensure that the cursor is at the top of the file, and then do a word search for "cohort_par". You should find a line of code that looks like this:

```
allocate(cohort_par(60))
```

We need to increase the length of cohort_par so that it accommodates the new variables that we have added to Job file.xls. The length of cohort_par must be equal to (or greater than) the index of the last row that you reference in "Job File.xls" minus 1 (as the first row of the Excel file is ignored by the program).

It is now advisable that you check the code by doing the following:

1. save the revised model code,
2. build the solution in the "debug" configuration,
3. clear all break points
4. add a break point at the line in "2_data_input.F90": `call check_data()`
5. set the debugger running to the breakpoint (press F5)
6. check that your new model parameters are at the bottom of the cohort_par vector as expected.

## 4.3 Altering model grids to accommodate new characteristic

The dimensions of the grids upon which the behavioural solutions of the model are based are initialised in the file "3a_axes_init.F90". Comments at the top of subroutine AXES_SU are provided to aid you when you introduce new characteristics into the model. Specifically, you should note that:

```
!  n_grid(t,1,i) - denotes no of points
!  n_grid(t,2,i) - denotes minimum
!  n_grid(t,3,i) - denotes maximum

!  states stacked in following order:
!       cash on hand (w)
!       human capital / persistent wage innovation (y)
```

```
!      temporary wage innovation (e)
!      individual effect on wages - constant (iw)
!      Occupational Pension (s)
!             (s) also used for generic pension annuity from earliest pensionable age
!      Private Pension (p)
!      Own business wealth (ob)
!      BSP (cp1)
!      S2P (cp2)
!      ISA (isa)
!      ...
!      training (t)
!      birth cohort
!      wage offer (household / cohort member)
!       wage offer (spouse - where modelled explicitly)
!      ...
!      self-employment
!      retirement (wage penalty)
!      students at entry to simulation
!      default PP participation state
!      Personal Pension contribution rate when this is discrete
!      pension takeup (receipt) flag
!      education
!      na (a)
!      nk (k)
!
!  This ordering is structured around the following rules:
!
!  1)  continuous variables at top, discrete variables at bottom
!  2)  index of continuous variables from top, with exception of training and birth
cohort
!      training switches between a continuous and discrete state variable
!      birth cohort is discrete in grid solutions, and continuous in population
simulations
!  3)  index of discrete variables from bottom, with exception of wage offer
!      which is not explicitly generated in solution
!  4)  hence, add new continuous variables above training, and new discrete
!      variables below wage offer(s) (indicated by ...)
```

We should therefore add our new (discrete) characteristic below “wage offer”, at the point in the list indicated by “…”, just above “self-employment” in the current structure.

TIP: when adding in a new characteristic it is useful to take note of two alternative factors – (i) any previously implemented statistic that has similar properties to the new statistic, and (ii) the statistic that is immediately below (above) the new statistic for discrete (continuous) variables in the list of modelled variables that is provided above. Often these two factors coincide, which is the case in the current context. We are modelling health status in a very similar way to self-employment – both are discrete states, and both are being modelled exogenously. The “flag” to enable self-employment status is defined in the global vector semp_par(1) – searching for this flag will often help to minimise the risk of missing required alterations to implement a new characteristic in the code.

Search for semp_par(1) in the current file, and you should find:

```
              if ( semp_par(1).gt.0.5 ) then
              !*** self employment

                     ii = ii + 1_4
                     n_grid(i,1,ii) = 2.0
```

```
            n_grid(i,2,ii) = 0.0
            n_grid(i,3,ii) = 1.0
        end if
```

We need to add in the following statistics for our new characteristic, just above the code referred to above:

```
!  n_grid(t,1,i) - denotes no of points
!  n_grid(t,2,i) - denotes minimum
!  n_grid(t,3,i) - denotes maximum
```

This will involve adding in the code:

```
        if ( cohort_par(61).gt.0.5 ) then
        !*** health status

            ii = ii + 1_4
            n_grid(i,1,ii) = 2.0
            n_grid(i,2,ii) = 0.0
            n_grid(i,3,ii) = 1.0
        end if
```

where I have assumed that cohort_par(61) refers to a flag that turns health status on (1) or off (0) in the model. Furthermore, note that the lifetime is divided into "working" and "retired" segments. As health status is likely to apply to both segments of life, it will be necessary to add the above code for both of these segments, which should be evident in the region of code that is referred to above.

## 4.4 Alter expectations for utility maximisation

Close all of the open model files in the solution (by not the solution itself). As the program code extends over 1000's of lines, knowing precisely where to go in every context is not very practical. It is more useful, therefore, to know how to find the places in the code that you are looking for. This detail was provided to us in the previous stage, with the identification that our new characteristic is closely aligned with the modelling of self-employment in the program.

It is now useful to search for all instances in the code where self-employment is treated. Hold CTRL+F to obtain the "find dialog box". Type "semp_par(1)" into the textbox, and ensure that "Entire solution" is in the "Look in" menue.

You should scrutinise each case and consider whether it is relevant to our case as well. The first of these cases is in "2_data_input.F90", and looks like:

```
    if ( semp_par(1).gt.0.5 ) then
    ! self employed

        max_ndim = max_ndim + 1_4
        max_prob = max_prob * 2.0
    end if
```

This indicates that we need to adjust global variables concerning the maximum number of dimensions (max_ndim) and the maximum number of alternative cases to consider for evaluating individual expectations (max_prob) for our new characteristic. This could be done in our case by including:

```
if ( cohort_par(61).gt.0.5 ) then
! health status

       max_ndim = max_ndim + 1_4
       max_prob = max_prob * 2.0
end if
```

Enter this just prior to the self-employment code, following the order convention set out in the comment provided above (page **).

The next relevant case is found in file “4e_exp_prelim.F90”, in subroutine “disc_prelim”, where discrete states are initialised in the routine that solves for utility maximising decisions. The code here is:

```
!*************************************************************
!      self-employment axis
!*************************************************************
if ( semp_par(1).gt.0.1_8 ) then
! self-employment possible

       semp_axis_am1 = 1_4
       if ( current_age.eq.nint(model_par(5)) ) then
       ! if at forced retirement age (next year must be retired)

              semp_axis_a = 0_4
              semp_pts_a = 1_4
       else

              semp_axis_a = 1_4
              semp_pts_a = 2_4
       end if
else

       semp_axis_am1 = 0_4
       semp_axis_a = 0_4
       semp_pts_a = 1_4
end if
disc_axes_a = disc_axes_a + semp_axis_a
nexp_d = nexp_d * semp_pts_a
```

We should reflect this code for our case, again appearing just above the self-employment code as:

```
!*************************************************************
!      health status axis
!*************************************************************
if ( cohort_par(61).gt.0.1_8 ) then
! health status possible

       health_axis_am1 = 1_4
       health_axis_a = 1_4
       health_pts_a = 2_4
else

       health_axis_am1 = 0_4
       health_axis_a = 0_4
       health_pts_a = 1_4
end if
disc_axes_a = disc_axes_a + health_axis_a
```

```
nexp_d = nexp_d * health_pts_a
```

Two key points are of note here. First, we have add a series of new variables here, defined as health_***. It is necessary to do a supplementary search on the relevant semp_*** variables to ensure that the health related variables are all worked into the model properly.

Secondly, the suppression of the self-employment related variables here will not apply for our health related variables, and we therefore need to exercise special care in ensuring that the health related variables are fully worked into the "retired" component of the modelled life course. The best way to do this is to look for another model characteristic that applies to the retired lifetime, and is also similar to our health status. Birth cohort might suffice for this purpose, though it would be necessary to exercise a great deal of care in this regard.

Following each of these threads of enquiry through to their respective conclusions (making extensive use of key-word searches), will substantively address this remaining programming issues required to set-up the utility maximisation problem to take account of the new health characteristics. Note that you should not pursue key word searches into files names commencing with "5" or "6", which concern population simulation and analysis respectively.

## 4.5 The direct influence of health on simulated decision making

This aspect of amending the program is usually particular to the subject of interest. In some cases, it will involve altering the assumed preference relation, in others it will involve adapting expectations only. In our case, it would involve altering the way that the case available for immediate consumption is evaluated. This can be found by identifying where disposable income is evaluated in when describing model expectations. As displayed in Figure 2.1, this is implemented through subroutine EXPectations_UPDATE, found in file 4g_valfn_evals.F90. Searching for "tax_prep" (the module that packages tax inputs, as noted in Table 2.1) in file 4g_valfn_evals.F90, will identify where disposable income is evaluated, and will permit the necessary adjustments to non-discretionary expenditure to be implemented. Implementation of these model changes will often require a bit of experimentation, an issue that is returned to in Section 4.7 below.

## 4.6 Including health status when simulating a population through time

The process of implementing a model change in the subroutines that project micro-data through time usually mirrors that described above for identifying utility maximisation decisions. Key word searches on related variables will identify most of the points in the code that require attention, subject to key issues that are specific to the particular subject of interest (and are therefore usually easily identified).

## 4.7 Debugging

It is crucially important that any amendments to the model code be thoroughly debugged, after they are implemented. A typical cycle of debugging will involve the following:

1. Attempt to build the revised model code in the "Debug" configuration
    a. This step will often reveal a series of errors that are identified by the internal compiler.
    b. Resolve all issues reported by the compiler

2. Delete all break points
3. Run the code through the debugger, adopting a very small grid specification to ensure sensible run-times
    a. This step will often result in the code running a part-way through before it encounters a problem, either due to a critical error (e.g. a division by 0.0), or tripping one of the checks that have been programmed into the structure of the model.
    b. Resolve all issues identified in this way until the model runs all the way through to solve for the complete simulated lifetime.
4. Build the revised model code in the "complete" configuration.
5. Re-run the model, with the new features suppressed.
6. Compare results against the immediately preceding version of the model, saved as a "Draft Model", prior to altering the model code.
    a. These results should be identical. If not, then it is possible to track down unintended complications by running the same problem simultaneously through two debuggers, one using the new code, and the other using the Draft Model code.
7. Re-run the model with the model alteration implemented
    a. If possible, it is best to start with an allowance that should suppress any effects on the simulation. In our case, for example, this could be achieved by setting non-discretionary costs of health status to zero. Any disparities can, again, be identified by comparing simulations under the revised and Draft Model codes.
    b. Ensure that projected effects of the model are broadly sensible.

## Appendix A: Basic Rules of Syntax

- The Fortran code that is provided with the model is organised into four broad structures, which can usefully be thought of as containers.
- The largest container is the "solution", which is a set of files that comprise the basic building blocks that Fortran uses to generate program files
    - The "ANALYSIS solution" can be opened by double-clicking on the file "analysis.sln" in the "FORTRAN\ANALYSIS\" subdirectory.
- The second largest container is the "source file", into which code is written.
    - You can browse through the source files of a solution via the "Source Files" folder of the "Solution Explorer" (which can be seen by selecting "Solution Explorer" from the "View" menu of Visual Studio)
    - To add a new source file to a solution:
        - right click on the Source Files folder
            - select "add", "new item"
            - select "Fortran Free-form File (.f90)"
            - make up a name for the file toward the bottom
            - and press the "Add" button
- Each source file can contain one, or a number, of MODULEs.
    - A MODULE is predominantly a container to organise a number of SUBROUTINEs.
        - e.g. MODULE AA might contain SUBROUTINEs AA1 and AA2; and MODULE BB might contain SUBROUTINEs BB1 and BB2
        - a slight complication arises in relation to "global variables", which is returned to below.
- Most MODULEs are organised as follows:
    - MODULE AA
        - This line denotes the start of the MODULE with the name AA
    - IMPLICIT NONE
        - This line is necessary to avoid easy programming errors – do a google search on it for further detail
    - CONTAINS
        - This line notes that the SUBROUTINES that follow are contained within the MODULE
    - **** SUBROUTINEs then appear here ****
    - END MODULE AA
        - This line denotes the end of the module
- All of the program computations are undertaken by code that is organised within a series of SUBROUTINEs.
    - e.g. SUBROUTINE AA1(x, y) takes a series of inputs x, performs a number of calculations, and then returns a series of outputs y.
        - We would execute this subroutine by entering the following code: call AA1(x,y)
- To use (call) SUBROUTINE XX from within SUBROUTINE YY, either:
    - the two SUBROUTINEs must be organised within the same MODULE, or SUBROUTINE YY must be given access to the MODULE containing SUBROUTINE XX
        - e.g. in the above example SUBROUTINE AA1 could call AA2 by default, but would need to be given access to MODULE BB to call BB1 (or BB2)
- Each SUBROUTINE must be organised as follows (based on the above example):
    - SUBROUTINE AA1(x, y)
        - This line denotes the start of the subroutine, and the variables that are used as inputs and outputs – if the subroutine takes in no explicit inputs, and produces no explicit outputs, then we write "SUBROUTINE AA1()"

- USE BB
  - This line gives SUBROUTINE AA1 access to the SUBROUTINEs contained in MODULE BB
  - This line is only required if you want to access SUBROUTINES (or global variables) stored in MODULE BB
- IMPLICIT NONE
  - This line is necessary to avoid easy programming errors – do a google search on it for further detail
- real (8) :: x, y
  - This line of code refers to the "type definitions", and usually covers a number of lines.
  - A "type definition" tells fortran the explicit nature of the data that each variable contains.
  - The variables that you will most commonly require will be limited to real(8) (a number with a decimal point) and integer(4) (a whole number without any decimals) types.
    - A variable cc is assigned a real type by; real(8) :: cc
    - A variable cc is assigned an integer type by; integer(4) :: cc
    - If cc is a matrix of real numbers with dimension (5,4) (5 rows and 4 columns), then it is assigned by; real(8) :: cc(5,4)
  - You must assign types to all of the variables that are included as inputs and outputs to a given subroutine (e.g. x and y in the example here)
  - You must also assign types to all of the variables that you use within the subroutine, and which are discarded after the subroutine is complete
    - Variables discarded after a subroutine is complete are commonly referred to as "local variables".
- **** You then add in programming code to undertake your desired calculations here ****
- END SUBROUTINE AA1
  - This line denotes the end of your subroutine

The above covers just about everything you will need in relation to program structure. There is, however, one final complication. Fortran requires each variable that is used in any subroutine to be assigned a type (real / integer above). In most cases, the variables that you use will either be explicit inputs / outputs of a subroutine, or will be "local variables" that you don't mind discarding after your desired computations within the subroutine are complete. Nevertheless, there are a number of variables that you might want to make common to a range of subroutines, without needing to repeatedly pass these variables as explicit inputs to each subroutine. Examples in relation to tax and benefits calculations include the number of adults and children in a benefit unit, the employment status of adult benefit unit members, measures of gross income, and so on. This is achieved in the code using "global variables".

- You will find in the set of source files included with the TAX program, one called "2_global.F90".
- If you open this file, then you will see that it includes a module named "global_tax".
  - The MODULE global_tax includes a series of variable type definitions, and no subroutines.

- The variables defined within this module are referred to as “global variables”. This is because it is possible to share them between alternative subroutines without the need for explicit declarations.

- The global variables defined in MODULE global_tax are assigned values within the SUBROUTINE initialise_taxinputs (see Figure 1), found in the source file “1_TaxTools.f90”
  - Note that SUBROUTINE initialise_taxinputs is given access to the global variables by the “USE global_tax” declaration at line 23
- Any SUBROUTINE that subsequently requires access to the global variables need only include the declaration “USE global_tax” in its second line of code (as outlined above for SUBROUTINE structure)

## Appendix B: The Utility Function

This has two components to it. Within-period utility $u$ is a function of total benefit unit consumption $c_{i,t}$ adjusted for effective benefit unit size $\theta_{i,t}$ and leisure time represented by $l_{i,t}$.

represents the consumption-equivalent of leisure and $\varepsilon$ the elasticity of substitution between consumption and leisure.

$$u\left(\frac{c_{i,j}}{\theta_{i,j}}, l_{i,t}\right) = \left(\left(\frac{c_{i,j}}{\theta_{i,j}}\right)^{(1-1/\varepsilon)} + \alpha^{1/\varepsilon} l_{i,t}^{(1-1/\varepsilon)}\right)^{\frac{1}{1-1/\varepsilon}} \quad \textbf{(1)}$$

Within-period utility enters into an intertemporal utility function in the manner represented below. Intertemporal discounting takes a quasi-hyperbolic form, where $\delta$ is the long-run discount factor, and $\beta$ is the excess short-run discount factor. When $\beta = 1$, preferences are time consistent, which implies that – for any given set of circumstances – the same decisions will maximise expected lifetime utility, regardless of when the decisions are made. That is, if an individual could commit to savings and employment decisions that take their evolving circumstances into account for any future age, then they will make the same decisions regardless of their current age. With $0 < \beta < 1$, intertemporal preferences exhibit myopia, which means that people would like to be more patient in the future than will actually be the case. The model assumes that people are ‘sophisticatedly’ myopic, in the sense that they are aware of their own self-control problems and react to them. This can result, for example, in a preference to lock savings away in a pension rather than a bank account, to avoid the temptation of spending the savings prematurely.

$\gamma$ is relative risk aversion, and $\phi_{j-t,t}$ is the probability of surviving $j$ years, given survival to age $t$. $\zeta_a$ and $\zeta_b$ represent the warm glow utility derived from leaving a positive bequest $w^+_{i,t+1}$.

$$U_{i,t} = \left[u\left(\frac{c_{i,t}}{\theta_{i,t}}, l_{i,t}\right)^{1-\gamma} + \beta E_t\left\{\sum_{j=t+1}^{T} \delta^{j-t}\left(\phi_{j-t,t} u\left(\frac{c_{i,j}}{\theta_{i,j}}, l_{i,j}\right)^{1-\gamma} + \left(1-\phi_{j-t,t}\right)\left(\zeta_a + \zeta_b w^{+}_{i,t+1}\right)^{1-1/\gamma}\right)\right\}\right]^{\frac{1}{1-\gamma}} \quad \textbf{(2)}$$