Microsimulation modelling for CEA in Python

April 15, 2019

Microsimulation can bypass the lack of memory in Markov state transition models by tracking individuals and updating the model's transition probabilities and the state value items, costs and HRQoL, based on 'time in state'.

The suggested approach in this tutorial is to first define transition probabilities and state value items as functions that can take time as an argument. Then, the model must be set up to update time and call those functions using each individual's specific time-tracker.

1 Installing Python and necessary packages

This is a step-by-step guide on how to get started with microsimulation models for cost-effectiveness analyses (CEAs) using Python. If you have some experience with for example other programming languages such as R, you will find this relatively easy to follow.If you are completely new to any type of programming, that's ok!

To get started, you need to download Python, and the best way to do this is to download the Anaconda Python platform https://www.anaconda.com/distribution/ Once that is all set up you need to install the necessary packages. This requires opening your computer's command prompt. If you are working on a Windows machine, just click the Windows icon and enter 'cmd' into the search field. In the command prompt you can use 'pip' to install what you need.

The most essential packages will already come with the Anaconda install, but to run the microsimulation we will use the Mesa package.

```
In [1]: # pip install mesa
```

Mesa is a programme that allows you to do agent-based models, which are basically microsimulations where individuals can interact with eachother and their environment (https://mesa.readthedocs.io/en/master/). We will only use parts of Mesas functionality.

Open the 'Spyder' programme located in the Anaconda folder. This is a Python interpreter and is the Python equivalent to RStudio for R. In Spyder you can split up your code into cells by typing '#%%', which can be helpful when working with new code. You run a cell by 'Shift+Enter'.

2 Loading the packages

```
In [2]: from mesa import Agent, Model
        from mesa.time import RandomActivation
        import pandas as pd
        import numpy as np
```

```
import time
from collections import defaultdict
```

Set a seed for the random-number generator to so that each time you run your script later, you will be subject to the same sequence of random numbers. The seed can be set to any number, it has no particular meaning in this case.

```
In [3]: seed = 987654321
```

Define a variable, 'R', as the numpy module which allows stochastic values from distributions, with random numbers determined by the seed. Iterations of the model will continue on this sequence of random numbers unless the Kernel (i.e. the Python engine) is re-started. To re-start (Consoles -> Re-start Kernel)

```
In [4]: R = np.random.RandomState(seed = seed)
```

In this tutorial we will set up the model to run using deterministic values and probabilistic values from the start. We therefore create a variable to tell the functions later which values to use. To run the PSA later we change this variable to 1.

```
In [5]: psa = 0
```

3 Functions for model inputs

3.1 Define the transition probabilities of the model and their functions

In this example we have three discrete health states an individual can be in: 'A', 'B', and 'C' The treatment, Tx, reduces risk of progression from 'B' to 'C' by a relative risk of 0.6

Import external parameter values if needed. Read in with pandas from e.g. csv, Stata, or Excel file.

```
In [6]: # mortality = pd.read_excel('C:\\Users\\larsasp\\python tutorial\\mortality.xlsx')
        # mortality = mortality.set_index('age')
```

Background mortality In this example, all individuals are subject to background mortality from all states.

```
In [7]: def bg_death(age):
            """
            Function returns the probability of death for the general population
            given the age that enters the function.
            Import this from the DataFrame imported from Excel above.
            By assumption, there is no variation in the probability of death
            """
            # Define the probability by locating the value pertaining to the age
            # defined in the external file.
            # pr = mortality.loc[age, 'pr_death']

            # Alternatively, define the probabilities in 10-year brackets:
```

```
        if age < 35:
            pr = 0.0003
        elif 34 < age < 45:
            pr = 0.0004
        elif 44 < age < 55:
            pr = 0.0015
        elif 54 < age < 65:
            pr = 0.004
        elif 64 < age < 75:
            pr = 0.01
        elif 74 < age < 85:
            pr = 0.036
        elif 84 < age < 95:
            pr = 0.15
        elif 94 < age < 105:
            pr = 0.35
        else:
            pr = 0.35

        return pr
```

Transition probabilities between the three health states, A $\rightarrow$ B, B $\rightarrow$ C If an individual recieves treatment, this will on average reduce the annual probability of transitioning from B $\rightarrow$ C

```
In [8]: def a2b(psa):
            """
            Function returns the annual proability of transitioning
            from health state 'a' to 'b', takes the argumment psa which
            returns either the deterministic or proabilistic value
            """
            if psa == 1:
                pr = R.beta(a = 20, b = 80)
            else:
                pr = 0.2
            return pr
```

```
In [9]: def b2c(psa):
            """
            Function returns the annual proability of transitioning from
            health state 'a' to 'b', takes the argumment psa which returns
            either the deterministic or proabilistic value
            """
            if psa == 1:
                pr = R.beta(a = 32, b = 133)
            else:
                pr = 0.20
            return pr
```

```
In [10]: def rr_tx(psa):
             """
             Fucction returns the relative risk of progression from 'B' to 'C'
             if given the treatment, takes the argumment psa which returns either
             the deterministic or proabilistic value
             """
             if psa == 1:
                 pr = R.lognormal(mean = -0.094, sigma = 0.014019)
             else:
                 pr = 0.91
             return pr
```

Now we can define the vectors of transition probabilities pertaining to each state. Although the individuals face different risks of transitions to other health states, for any given individual between two cycles, only one 'event' can come to fruition. This is what really what separates a cohort model from an individual level model. This also introduces 'first order-variability' to the model, which is essentially stochastic noise. This is something to keep in mind when interpreting the results later on.

```
In [11]: def from_A(psa, age):
             """
             Can transition to  {0: stay in A,
                                 1: transition to B,
                                 2: die from background mortality}
             """
             a = [a2b(psa), bg_death(age)]
             b = 1 - sum(a)
             vTP = [b, a[0], a[1]]
             event = R.multinomial(n = 1, pvals= vTP)
             return event

In [12]: def from_B(psa, age, tx):
             """
             If randomised to treatment (randomiser > 0.5):
                 Can transition to  {0: stay in B,
                                     1: transition to C with reduced risk due to Tx,
                                     2: die from background mortality}
             If randomised to control:
             Can transition to  {0: stay in B,
                                 1: transition to C,
                                 2: die from background mortality}
             """
             if tx == 1:
                 a = [b2c(psa)*rr_tx(psa), bg_death(age)]
                 b = 1 - sum(a)
                 vTP = [b, a[0], a[1]]
             else:
                 a = [b2c(psa), bg_death(age)]
```

```
        b = 1 - sum(a)
        vTP = [b, a[0], a[1]]
    event = R.multinomial(n = 1, pvals= vTP)
    return event
```

```
In [13]: def from_C(psa, age):
    """
    Can transition to   {0: stay in C,
                         1: die from background mortality}
    """
    vTP = [(1-bg_death(age)), bg_death(age)]
    event = R.multinomial(n = 1, pvals = vTP)
    return event
```

3.2 Define the costs

We assume here time dependent costs as a function of time in 'B' and 'C' Health state 'A' have no health-related costs Health state 'B' have costs of the treatment Tx, but only year 1 in the health state Health state 'C' have different costs for the years 1, 2, 3 in the health state, and time spent in the state after this

```
In [14]: def cost_b(psa, tx, time_in_b):
    """
    Function returns the treatment cost of Tx if time in the state == 1,
    also takes the argumment psa which returns either the deterministic
    or proabilistic value
    """
    if tx == 1:
        if time_in_b == 1:
            if psa == 1:
                c = R.gamma(shape = 350, scale = 2.3)
            else:
                c = 805
        else:
            c = 0
    else:
        c = 0
    return c
```

```
In [15]: def cost_c(psa, time_in_c):
    """
    Function returns the health related costs of staying in 'C',
    dependent on time spent in the state.
    Also takes the argumment psa which returns either the
    deterministic or proabilistic value.
    """
    if time_in_c == 1:
        if psa == 1:
            c = R.gamma(shape = 500, scale = 20)
```

```
            else:
                c = 10000
        elif time_in_c == 2:
            if psa == 1:
                c = R.gamma(shape = 500, scale = 20)
            else:
                c = 2000
        elif time_in_c == 3:
            if psa == 1:
                c = R.gamma(shape = 100, scale = 5)
            else:
                c = 500
        else:
            if psa == 1:
                c = R.gamma(shape = 80, scale = 3)
            else:
                c = 240
        return c
```

3.3 Define the health-related quality-of-life (HRQoL) weights to assign to the states

Health state 'A' describes an asymptomatic health state, and is by assumption set at 'perfect health' (weight = 1) Health state 'B' also describes an asymptomatic health state but has an associated HRQoL weight of 0.75 in the first cycle spent in the health state if treated, and is by assumption set at 'perfect health' otherwise (weight = 1) Health state 'C' has an associated HRQoL weight of 0.55 year 1, 0.50 year 2, and 0.40 all subsequent years

```
In [16]: def u_b(psa, tx, time_in_b):
             """
             Function returns the HRQoL-weight for time spent in health state 'B',
             and is dependent on treatment status, and time spent in the state.
             Also takes the argumment psa which returns either the
             deterministic or proabilistic value.
             """
             if tx == 1:
                 if time_in_b == 1:
                     if psa == 1:
                         u = R.beta(a = 40, b = 13)
                     else:
                         u = 0.75
                 else:
                     u = 1
             else:
                 u = 1
             return u

In [17]: def u_c(psa, time_in_c):
             """
```

```
        Function returns the HRQoL-weight for time spent in health state 'C',
        and is dependent on time spent in the state.
        Also takes the argumment psa which returns either the
        deterministic or proabilistic value.
        """
        if time_in_c == 1:
            if psa == 1:
                c = R.beta(a = 30, b = 25)
            else:
                c = 0.55
        elif time_in_c == 2:
            if psa == 1:
                c = R.beta(a = 20, b = 20)
            else:
                c = 0.5

        else:
            if psa == 1:
                c = R.beta(a = 20, b = 30)
            else:
                c: 0.40
        return c
```

4 Define the individuals and their characteristics

At this point, we need to actually specify how the model should use the transition probabilities, costs and HRQoL-weights. We need to define their age (which determines the likelihood of all cause mortality), and randomise them to treatment or controll. One way to do this is to create a 'class' to which we can assign characteristics by specifying them as *self*.variablename. In addition to characteristics, we also define 'tracking' variables to store information about the individuals so that we can keep track of time spent in health states, how much cost they have accumulated, and their HRQoL. In this case, we assume that one cycle of the model is one year. This means that we can also interpret the HRQoL as the quality-adjusted life-years (QALYs).

The model's *step function* is structured around if/else statements, which will be controlled against the information that is stored for each individual. This is by no means a computationally efficient way of structuring the model, but it does provide an intuitive way of conceptualising it. At each model cycle (one step) the age, time in the states, and the state-specific costs and HRQoL-weigths are updated and added to the trackers.

```
In [18]: class Individual(Agent):
            """
            Set the basic characteristics, define these as 'self.variable'
            """
            def __init__(self, unique_id, model):
                super().__init__(unique_id, model)
                self.age = 30 # Age at start
                # Randomise to treatment or control
```

```
        if self.unique_id % 2 == 1:
            self.tx = 1
        else:
            self.tx = 0
        """
        Possible health states ('A starting state')
        """
        self.A = 1
        self.B = 0
        self.C = 0
        self.dead = 0
        """
        Set up the individual trackers
        """
        self.cost = 0
        self.hrqol = 1
        self.time_in_b = 0
        self.time_in_c = 0
        self.cycle = 0
        """
        Set up the step function which determines how the
        individuals can transition between the states.
        """
    def step(self):
        if self.A == 1:
            if from_A(psa, age = self.age)[1] == 1:
                self.B +=1
                self.A -=1
                self.time_in_b +=1
                self.hrqol = u_b(psa, tx = self.tx, time_in_b = self.time_in_b)
                self.cost = cost_b(psa, tx = self.tx, time_in_b = self.time_in_b)
            elif from_A(psa, age = self.age)[2] == 1:
                self.dead +=1
                self.A -=1
                self.cost = 0
                self.hrqol = 0
            else:
                self.age +=1
                self.hrqol = 1
                self.cost = 0

        elif self.B == 1:
            if from_B(psa, age = self.age, tx = self.tx)[1] == 1:
                self.C +=1
                self.B -=1
                self.time_in_c +=1
                self.cost = cost_c(psa, time_in_c = self.time_in_c)
                self.hrqol = u_c(psa, time_in_c = self.time_in_c)
```

```
            elif from_B(psa, age = self.age, tx =  self.tx)[2] == 1:
                self.dead +=1
                self.B -=1
                self.cost = 0
                self.hrqol = 0
            else:
                self.age +=1
                self.time_in_b +=1
                self.cost = cost_b(psa, tx =  self.tx, time_in_b  = self.time_in_b)
                self.hrqol = u_b(psa, tx = self.tx, time_in_b = self.time_in_b)

        elif self.C == 1:
            if from_C(psa, age = self.age)[1]:
                self.dead +=1
                self.C -=1
                self.cost = 0
                self.hrqol = 0
            else:
                self.age +=1
                self.time_in_c +=1
                self.cost = cost_c(psa, time_in_c = self.time_in_c)

        update(self)
        self.cycle +=1
```

5 Define the simulation process

At this point we create the class that holds the individuals. Within this class we tell the model in which order the individuals are to be 'activated' at each step, which has no implication in this case as we have no interaction between them. The first part, the '**init**()' tells the model how many individuals to create, and to add these to the 'scheduler'.

```
In [19]: class simulation(Model):
            def __init__(self, N):
                super().__init__()
                self.N = N
                self.cycle = 0
                self.schedule = RandomActivation(self)
                for i in range(self.N):
                    j = Individual(i, self)
                    self.schedule.add(j)

            def advance(self):
                self.schedule.step()
```

6 Define how to collect the data

The model is actually at this point finished, and will run. However, we do need to set up an outer environment to store the data it creates. This is what the function 'update(self)' defined in the Individual class above will do. In this example we store the data in a dictionary, which will have an unique entry for each individual. Each individual's entry will have a list of their stored value at each cycle they are alive.

```
In [20]: #%% Create a dictionary with the model information
         model_dict = defaultdict(list)

         def update(self):
             """
             Function takes each individual's information at each
             cycle and adds it to the outer model dictionary
             """
             if self.hrqol != 0:
                 model_dict[self.unique_id].append([self.cycle,
                                                    self.age,
                                                    self.A,
                                                    self.B,
                                                    self.C,
                                                    self.cost,
                                                    self.hrqol,
                                                    self.tx])
```

7 Prepare the model for both single - and multi run use

As mentioned at the start, it is good practice to decide what you need to do with the model before you start modelling. In most CEA research papers, the main results of interest are derived from the probabilistic output. For this, we have to do all of the post-simulation analysis to filter out the treatment vs control stratified cost-effectiveness results, adjust future outcomes to present value by discounting, and storing of results within the function that runs the model.

```
In [21]: N = 10000 # number of individuals
         cycles = 70 # number of model cycles, here set at 100 years - initial age
         runs = 1 # number of (outer) runs of the model for PSA
         results = {} # dictionary to store the cost-effectiveness output
         dr = 0.035 # discount rate for present value of future output
```

To discount outcomes for each person we need a function that can take the time observed for each individual and pass it through this function to create a vector of discount rates to multiply the individuals' costs and QALYs.

```
In [22]: def discount_function(time):
             """
             Function to create a vector of discount rates.
             """
```

```
            discount = 1 / (1+dr)**time
            return discount
```

Running the code below will then run the model with N individuals, k runs, over i cycles. The individual-level output is collected from the 'model_dict'-dictionary, and the appropriate calculations are done before storing the mean outcomes to the 'results' dictionary.

```
In [23]: start = time.time()
         for k in range(runs):
             model = simulation(N)
             for i in range(cycles):
                 model.advance()

             df = pd.DataFrame([[k] + j for k,v in model_dict.items() for j in v],
                               columns = ['ID', 'Cycle', 'Age', 'A', 'B', 'C',
                                          'Cost', 'QALYs', 'Treatment/control'])
             discounting = discount_function(df['Cycle'])
             df['dCost'] = df['Cost']*discounting
             df['dQALYs'] = df['QALYs']*discounting
             df2 = df.groupby('ID')['A', 'B', 'C', 'dCost',
                                    'dQALYs', 'Treatment/control'].sum()

             tx = pd.DataFrame()
             tx['Treatment/control'] = df.groupby('ID')['Treatment/control'].max()
             individual_output = pd.merge(left = df2, right = tx,
                                          left_index=True, right_index=True)

             treated = individual_output[(individual_output['Treatment/control_y'] == 1)]
             control = individual_output[(individual_output['Treatment/control_y'] == 0)]

             results[k] = {'Costs_Tx': treated['dCost'].mean(),
                           'Costs_ctl': control['dCost'].mean(),
                           'QALYs_Tx': treated['dQALYs'].mean(),
                           'QALYs_ctl': control['dQALYs'].mean()}
         end = time.time()
         exe_time = end - start

         print(exe_time)
```

```
6.861062526702881
```

That's it.

8 Post-simulation analysis

The post-simulation analysis of CEAs has at a minimum three components:

- Incremental cost-effectiveness ratio

- Scatterplot of ICERs
- Cost-effectiveness acceptability frontier

But first, it might be helpful to take a look at the output, and see if it makes sense.

```
In [24]: individual_output.head()
```

```
Out[24]:      A   B   C         dCost     dQALYs  Treatment/control_x  \
         ID
         0    1   6  44  13549.101732  16.298439                    0
         1    2   2  37  15402.959683  13.772649                   41
         2   16  11  34   6555.957290  22.320262                    0
         3    4   6  46  12995.654743  17.550828                   56
         4   16   6  29   7592.396206  20.514678                    0

             Treatment/control_y
         ID
         0                     0
         1                     1
         2                     0
         3                     1
         4                     0
```

We have already made separate DataFrames for the individuals according to treatment status, so from these we can calculate the incremental costs of treatment relative to no treatment

```
In [25]: icer = pd.DataFrame.from_dict(data = results, orient = 'index')
         icer['Inc_Costs'] = icer['Costs_Tx'] - icer['Costs_ctl']
         icer['Inc_QALYs'] = icer['QALYs_Tx'] - icer['QALYs_ctl']
         icer['ICER'] = icer['Inc_Costs']/icer['Inc_QALYs']
         icer
```

```
Out[25]:        Costs_Tx     Costs_ctl   QALYs_Tx  QALYs_ctl   Inc_Costs  Inc_QALYs  \
         0  13417.443778  12967.281085  16.881008  16.948568  450.162693   -0.06756

                   ICER
         0 -6663.131073
```

8.0.1 A few basic plots

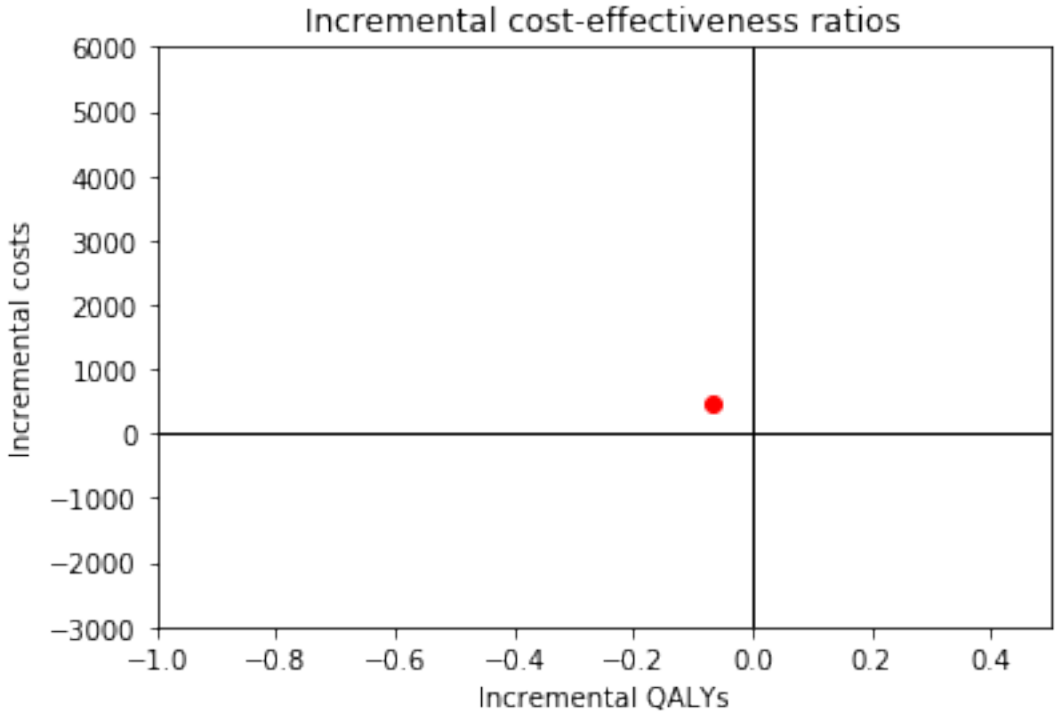
```
In [26]: import matplotlib.pyplot as plt
```

Scatterplot of ICERs

```
In [27]: fig = plt.figure()
         ax1 = plt.subplot(111)
         ax1.scatter(x = icer['Inc_QALYs'], y = icer['Inc_Costs'], color = 'red')
         ax1.set_ylim(-3000, 6000)
         ax1.set_xlim(-1, 0.5)
```

```
ax1.vlines(x = 0, ymin = -3000, ymax = 6000, color = 'black', linewidth = 1)
ax1.hlines(y = 0, xmin = -1, xmax = 0.5, color = 'black', linewidth = 1)
ax1.set_ylabel('Incremental costs')
ax1.set_xlabel('Incremental QALYs')
ax1.set_title('Incremental cost-effectiveness ratios')
```

```
Out[27]: Text(0.5,1,'Incremental cost-effectiveness ratios')
```



CEAC/CEAF

These are a bit more involved, but not really difficult. Just make a vector for willingness to pay (WTP) per QALY, and calculate the incremental net-monetary benefit/net health benefit. Plot proportion of positive iNMBs at each WTP level and plot CEAC. CEAF is the mean iNMBs, and could be plotted in the same.

```
In [28]: WTP = np.arange(0, 51000, 1000)
         iNMB = dict()
         for i in WTP:
             x = (i*icer['Inc_QALYs'] - icer['Inc_Costs'])
             iNMB[i] = x
```

```
In [29]: df3 = pd.DataFrame.from_dict(iNMB, orient = 'index')
         df3.head()
```

```
Out[29]:                0
         0    -450.162693
```

```
1000 -517.722926
2000 -585.283160
3000 -652.843394
4000 -720.403628
```