YSC2254: Modelling and Optimization

Term Project

Applications of Non-Linear Programming and Markov Chains

Submitted By :
Koa Zhao Yuan

Contents

Introduction & Acknowledgements 2

Snakes and Ladders 3

- Implementation 3
- Preliminary Analysis 6
- Snakes and Ladders 7
- Fundamental Matrix - Expected Number of steps to win 10
- Limitations 12
- Extensions 13

Midterm Project Review 14

- Problem Setting 14
- Constraints - Sanity checks 16
- Constraints - Ranges 16
- Other (Non-)linear Constraints 18
- Results & Analysis 20
- Assumptions & Further Studies 22

Conclusion 22

Introduction & Acknowledgements

We've finally come to the end of this module! Throughout the semester, I've learnt about various ways of maximizing or minimizing various objectives. These included areas such as Linear Programming, Integer Programming, Non-Linear Programming (And the painful Lagrange multipliers and Kuhn Tucker Conditions), Decision Trees and Markov Chains.

In this term project report, I will present about 2 project areas: Modelling the Snake and Ladders game using a Markov Chain and reviewing my Midterm Project using Non-Linear Programming methods. Both of them will proceed in a very similar manner: I plan to introduce the problem, then my methodology for modelling these problems. Lastly, I will perform some analysis on the data and derive some meaningful conclusions from it.

The first project will be coded out in R without libraries while the second project will be done using R using the library "nloptr". I tried to use python libraries to do the second project to no avail.

Before beginning this project, I'd like to thank Prof Tim for creating this new course from scratch. It is no trivial task to formulate a new syllabus from scratch, and I think that this course has a lot of potential! While there might be some rough edges to be polished, I'm sure that Modelling and Optimization has the power to find its own niche in Yale-NUS.

With that out of the way, allow me to explain more about my first project on modelling Snakes and Ladders using Markov Chains.

Snakes and Ladders

Do you remember any childhood games you've played in the past, but struggled to understand the underlying stochastic or mathematical mechanisms behind them? For me, one of them is Snakes and Ladders. You roll a dice and move a counter based on the number of steps and hope that you roll the best possible outcome. However, during my younger days, I played this game, thinking that it was simply a game of chance with nothing more. However, it turns out that there's a little more to that. For the purpose of this project, I will assume that this game is played with a single player with 100 squares, subjected to the following mechanics:

- The player places their counter on the first square.
- The player takes turns to roll two fair six-sided dice. Move the counter forward the number of spaces shown on the sum of the 2 dice (ranging from 2-12).
- If the counter lands at the bottom of a ladder, the player moves up to the top of the ladder.
- If the counter lands on the head of a snake, the player slides down to the bottom of the snake.

The player wins when he/she reaches the highest space on the board, 100. To win, the player will need to roll the exact number to get to the last space. If the roll is too high, the player's piece will bounce off the last space and move back. For instance, If a player had four spaces to get to a 100 (square 96) and rolled a 6, the piece will move four spaces to 100, then "bounce back" two spaces to 98.

Now that these are the rules of the game, let's explain why this game can be modelled using a Markov Chain. The reason is because Snakes and Ladders is a stochastic process (due to dice rolling). In addition, the states that the counter can reach are entirely dependent from the previous state. For instance, if the counter was at square 10, it can only reach squares 12 to 22, which represents the possible values of the 2 dice rolls. Therefore, it is possible to formulate a transition matrix based on this. Now that we are done with introducing the problem, let's go ahead to the implementation.

Implementation

Firstly, I created a table representing the probability of the sum of the 2 dice rolls being a certain value. The results are shown in the next page:

Sum of Rolling 2 Dice	Probability
0	$^0/_{36}$
1	$^0/_{36}$
2	$^1/_{36}$
3	$^2/_{36}$
4	$^3/_{36}$
5	$^4/_{36}$
6	$^5/_{36}$
7	$^6/_{36}$
8	$^5/_{36}$
9	$^4/_{36}$
10	$^3/_{36}$
11	$^2/_{36}$
12	$^1/_{36}$

We could ignore the first 2 zeros and create a vector presenting the transition probabilities, which i name *dice_prob*:

```
dice_prob <- c(1/36, 2/36, 3/36, 4/36, 5/36, 6/36,
               5/36, 4/36, 3/36, 2/36, 1/36)
```

The next step was to create a transition matrix. However, if we manually input this, it'd be a 100×100 matrix and that's not very efficient, so let's just automate it. There are 2 cases here:

- Bounce-back mechanic is not relevant (e.g. Square 10)
- Bounce-back mechanic is relevant (e.g. Square 95)
- Counter as at square 100

The first case is much more simpler. If the bounce-back mechanic is not relevant, then that means that if the counter is at the $n-th$ square, it can only reach the $(n+2)$-th to $(n+12)$-th square, barring snakes and ladders. The probability of getting to a square within that range would correspond to *dice_prob*, while the other squares are simply 0.

For the bounce-back case, we know that counter has overshot, which means that it goes beyond square 100. So, what we could do here is to use this formula

$$S_{final} = 100 - St_{over}$$

Where S_{final} represents the final square the counter is in and St_{over} represents the number of squares overshot beyond 100. This can also be expressed equivalently as

$$\begin{aligned} S_{final} &= 100 - St_{over} \\ &= 100 - (S_{Hypo} - 100) \\ &= 200 - S_{Hypo} \end{aligned}$$

Where S_{Hypo} represents the square that the counter would be, given that the board had an unlimited number of squares. With our 2 cases done and dusted, we can create the function for the transition matrix.

The third case is rather trivial. This means that the player has already won the game, so the 100-th square becomes an absorbing state. this means that the $(100, 100)$ entry of the matrix is 1, and the other entries for the transition states of square 100 have a probabiliy of zero.

Below shows the implementation of what I've explained just now in a function. For the transition matrix, I've decided to implement it as a 100×100 matrix:

$$P = \begin{bmatrix} p_{1,1} & p_{2,1} & \cdots & p_{100,1} \\ p_{1,2} & p_{2,2} & \cdots & p_{100,2} \\ \vdots & \vdots & & \vdots \\ p_{1,100} & p_{2,100} & \cdots & p_{100,100} \end{bmatrix}$$

This is where $p_{i,j}$ represents the probability of going from the i-th square to the j-th square in one turn (sum of 2 dice rolls). With all these set in stone, let's create the function that automatically creates the transition matrix:

```
make_trans <- function(){
  trans <- matrix(0, nrow = 100, ncol = 100)
  for (i in 1:99){
    for (j in 1:11){
      if (i + j + 1 > 100){
        # if counter goes beyond the grid
        trans[200 - (i + j + 1), i] <- trans[200 - (i + j + 1), i] + dice_prob[j]
      } else {
        # if counter does not go beyond the grid
        trans[i + j + 1, i] <- trans[i + j + 1, i] + dice_prob[j]
      }
    }
  }
  # last square is absorbing
  trans[100, 100] <- 1
  return(trans)
}
```

Creating the initial state matrix starts from creating a column vector of length 100 and letting its first entry be 1 since the counter starts from the first square:

```
    state <- rep(0, 100)
    state[1] = 1
```

Lastly, we note that the n-step state can be obtained from this formula (Taken from Chapter 17.3):

$$S_n = T^n S_0$$

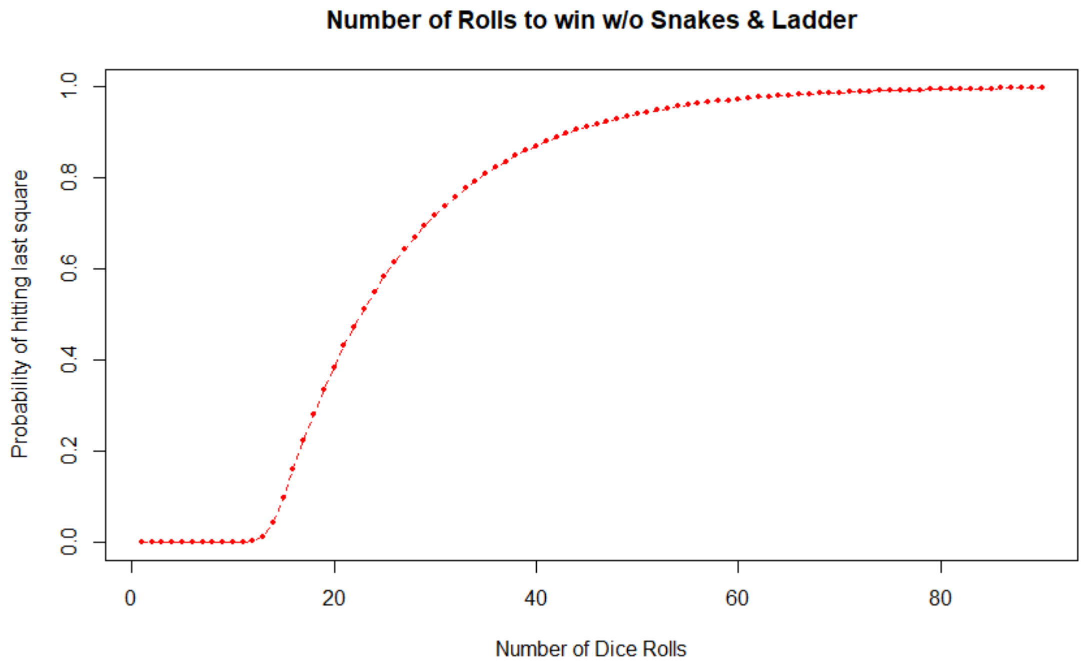
Where S_0 is the initial state column vector, T represents the transition matrix, n represents the number of turns performed (dice rolls) and S_n represents the state after n rolls.

Preliminary Analysis

Before moving to the implementation of Snakes and Ladders, let's perform some preliminary analysis on how many rolls it'll take to win over time. The code is shown below:

```
# run markov chain 90 times
res_initial <- c()
for (i in 1:90){
  state <- transition %*% state
  res_initial <- append(res_initial, state[100])
}
```

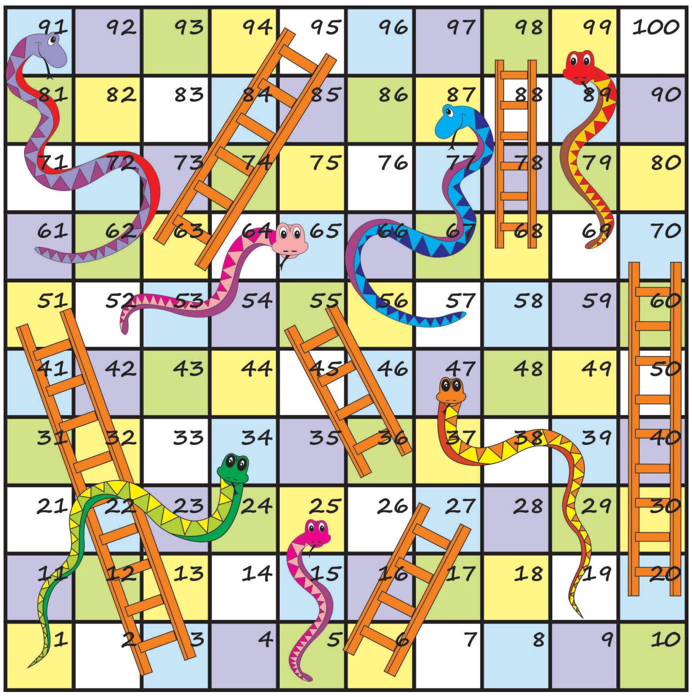
This code snippet does the following: it creates a new vector, *res_initial* and appends the 100-th index of the state vector to the result after every transition from state i to $i+1$, and repeats this for 90 times. In the very end, we can get a plot of the probability of winning against the number of turns done, which is shown below:



As expected, from 0-12 dice rolls, the probability of winning is 0. This is because without any ladders, the furthest square that can be reached is 97 and not 100. However, what happens when snakes and ladders are included? A comparison will be explored in the next section.

Snakes and Ladders

For this section, we refer to the following board for analysis:



In order to implement a transition matrix that reflects snakes and ladders, let's consider what happens to the transition matrix. Consider the snake with head at square 47 and tail at square 19 (As per the diagram). If the counter lands at square 19, nothing happens. However, if the counter lands at square 47, the counter moves to square 19.

Therefore, an insight can be drawn here: the head of a snake, and by extension the base of a ladder is considered a **closed set**, since these squares are unreachable. Hence, we should set all $p_{ik} = 0$, where k is the square containing the snake's head or ladder's base.

However, we must also note that we need to add these corresponding states to the snake's tail and the ladder's top, as this is exactly where the counter is moved to. the only thing left is to apply this as a function:

```
# swapping function for snakes and ladder
transfer_aux <- function(square, old, new, trans){
  temp <- trans[old, square]
  trans[old, square] <- 0
  trans[new, square] <- trans[new, square] + temp
  return (trans)
}

# function to swap states of all 100 squares
# inefficient but gets the job done nicely
transfer <- function(old, new, trans){
  for (i in 1:100){
    trans <- transfer_aux(i, old, new, trans)
  }
  return (trans)
}
```

In this implementation, the variables are:

- **old** - square counter with head of snake, bottom of ladder
- **new** - square counter is moved to (top of ladder, tail of snake)
- **trans** - transition matrix
- **square** - column to perform swapping on (with for-loop), basically the whole row

All that is left is to execute this function over the transition matrix, starting from the snakes:

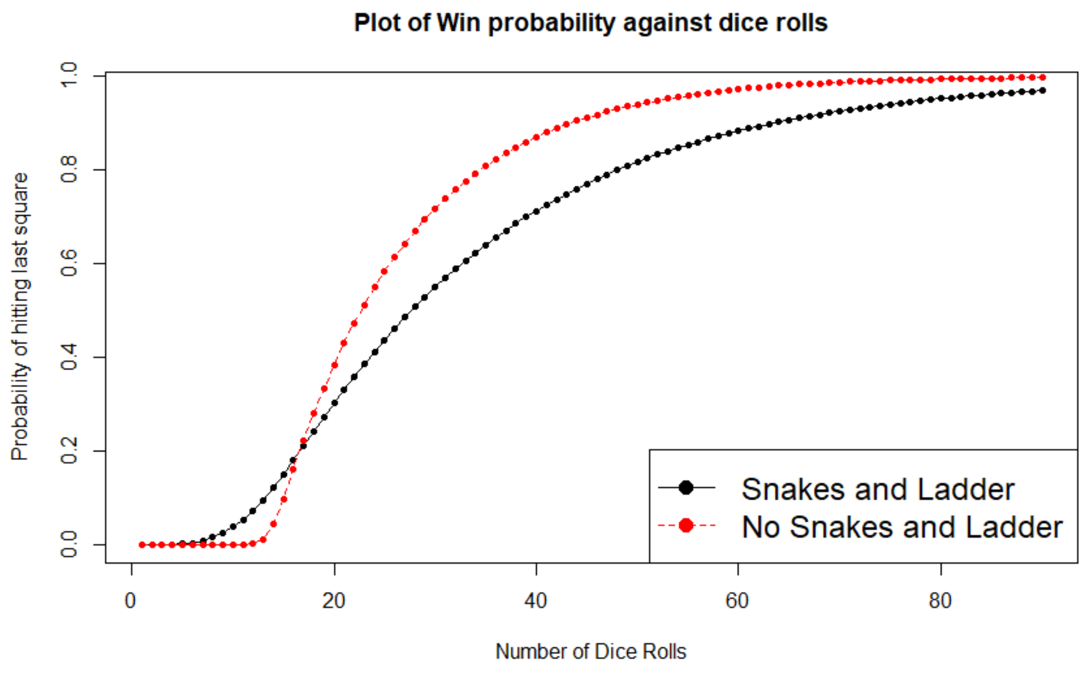
- Snake from square 34 to square 1
- Snake from square 25 to square 5
- Snake from square 47 to square 19
- Snake from square 65 to square 52
- Snake from square 87 to square 57
- Snake from square 91 to square 61
- Snake from square 99 to square 69

Next, we move to the ladders:

- Ladder from square 3 to square 51
- Ladder from square 6 to square 27
- Ladder from square 20 to square 70
- Ladder from square 36 to square 55

- Ladder from square 63 to square 95
- Ladder from square 68 to square 98

With these things set in stone, we can now compare the plots of the probability to win against the number of dice rolls. This is shown below:



Based on this plot, one can see that in the short term ($\leq$ 15 turns), the probability of winning with the snakes and ladders board is higher than the board without snakes and ladders. Reflecting about it, this could be possible because of the ladders in play. If a player is lucky and strikes the best case scenario consecutively, then the counter will be able to step on the base of ladders and not the head of snakes and climb the board rapidly. This explains why the black curve is higher than the red curve initially.

However, as time progresses, we find that the black curve is now below the red curve. There are a few possible reasons for this:

Presence of Snakes

In this case, we note that there are snakes present which can set back a player's progress even though he/she might have went far in the board. Therefore, it is more likely that in the longer term, a player might hit one of these snakes and then have to backtrack to the top, taking additional turns and reducing their probability of hitting the final square.

We also note that in our board, we have more snakes (7) than ladders (6). In other words, if a random square is chosen, it is more likely to set the player back than forward, setting their progress back. This is one possible reason why as the number of rolls, the probability of winning on the board with snakes and ladders is lesser than the board without them.

Creation of Larger Transient states

This reason is a more rigorous analysis on the board than we have done for the previous reason. Before beginning, I reiterate the definition of a **transient** state: if there exists a state j that is reachable from i, but the state i is not reachable from state j.

In our board without the snakes and ladders, this would mean that since squares 1 to 88 are not reachable from squares 89 to 100 (Taking into account the bounce-back mechanic), that would mean that states 89 to 100 are permanently transient. In addition, since one cannot back-track in this mode, this means that (except the last 12 squares) each forward square is transient relative to the previous ones (Square 5 is transient relative to squares $1, 2, 3$), and so on.

However, with the inclusion of the snakes, it is possible to back-track in terms of progress. Hence, this increases the number of **communicable** states in the board, which means that a counter on average spends more rolls within the board before winning. Therefore, this made the black curve lower than the red curve beyond approximately 15 rolls.

Fundamental Matrix - Expected Number of steps to win

While this was not taught in class, we could calculate the expected number of steps to win. In the Lecture notes, it is said that the transition matrix can be segregated into four parts:

$$P = \left(\begin{array}{c|c} Q & R \\ \hline 0 & I \end{array}\right)$$

However, the way I defined my matrix, it actually looks more like this, since my matrix is the transpose of whatever is defined in the lecture notes:

$$P_{coded} = \left(\begin{array}{c|c} Q^T_{coded} & 0 \\ \hline R & I \end{array}\right)$$

Hence, I should take note of this while implementing the code in R. With this, I can easily obtain the fundamental matrix which is

$$N = (I_{99} - Q_{coded}^T)^{-1}.$$

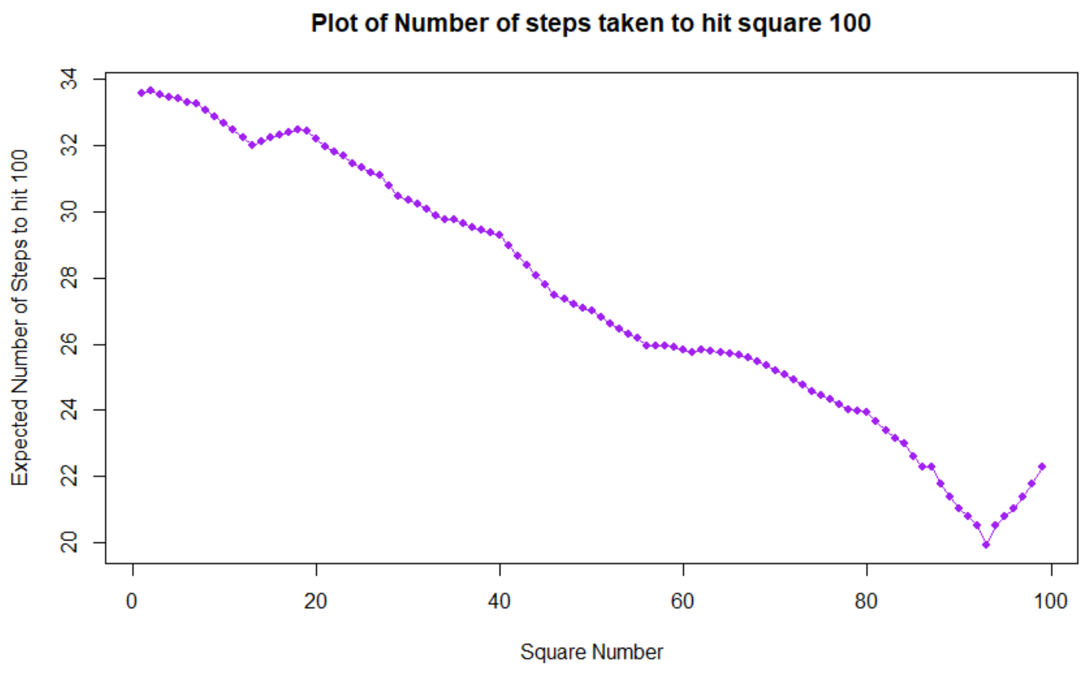
This is correct since in this case,

- Q_{coded}^T are filled with transient states.
- I represents the only absorbing state at square 100
- R represents the vector of the probabilities that given the counter in a certain square, it hits square 100 in one turn.

What we can find is something rather useful: the expected number of steps (turns) taken before being absorbed when starting in square i, where $1 \leq i \leq 99$, which square i is the i-th entry of the vector

$$\mathbf{t} = N\mathbf{1},$$

where $\mathbf{1}$ is a length 99 vector (since N has 99 rows and columns, after having 1 shaved off from the hundredth square) filled with ones. The plot of the expected number of steps taken to hit square 100 against starting square number is shown here:



Interesting insights

Here, a number of insights can be drawn. Firstly, we note that we have an downward trend in the expected number of steps needed to win the game. This is expected, since we are getting closer to square 100. However, we notice some remarkable bumps and small peaks along the way. I'm not finally sure what to make out of it, but i saw a few patterns. These are all referenced from the fact that the most likely dice roll value we can get is 6 with a probability of $\frac{1}{6}$:

- Humps represent squares that are most likely to hit the head of a snake while dips represent squares that are most likely to hit the base of a ladder.
- the larger the humps and ladders, the greater the shift in the number of steps and more likely it is to hit that particular square.

For the first point, it isn't difficult to look at why this is the case. humps mean that more steps are required to hit square 100, which are what snakes generally do. dips mean that less steps are required to hit square 100, which correspond to ladders.

For the second point, we observe that square number 3 is the base of ladder which brings one straight to square 51 which is perhaps one of the most significant shifts in the board. However, the "dip" at square 1 is rather insignificant as seen as it is only square that can hit square 3 with a measly probability of $\frac{1}{36}$. In contrast, looking at square 20, we have a ladder that goes straight up to square 70. However, the corresponding dip in square 14, (most likely square to hit square 20) in one roll is much higher than square 1. This is because the probability of hitting square 20 from 14 is much higher at $\frac{1}{6}$.

The valley seen in squares $89 - 99$ can be easily explained by the bounce-back mechanic that I have implemented in this game. However, something even more surprising is found at square 94: even with a $\frac{1}{6}$ chance to win the game right off the bet, it takes an approximately expected 20 steps to hit square 100. This is very surprising as this means that the counter actually spends a majority of the time bouncing back and hitting snakes without hitting the end.

Limitations

In this project, there are definitely a few limitations. Firstly, it is worth noting that I have restricted the game to only 1 player. Traditionally, snakes and ladders has been a game played by 2 players each time. While this simplification made my Markov chain implementation easier, it also reduced its ability to examine more complex situations.

Secondly, there are only 3 events that are really present in this game: snakes, ladders and dice rolls. This made the game implementation relatively simple and easy, but not very nuanced in general. Perhaps further exploration of this projects could include adding more interesting mechanics such as when landing on a snake, you are not guaranteed to roll, but have a probability to based on another dice roll, etc.

Improvements to these limitations will be discussed briefly in the next section.

Extensions

In this project, I have attempted to apply the concept of Markov Chains to a Snakes and Ladders, a childhood game that I used to play but not really understand really well. I've also gathered some interesting findings about the expected probability to win after a number of turns. I feel that now, if I play this game again, I'd think more about the inner workings of probability here. Some viable extensions of this project are:

- Varying the positions of snakes and ladders and seeing how this affects winning
- Modelling game as 2-player and analysing its fairness (equal chance of winning)
- Introducing new mechanics to the game that makes the Markov chain implementation more interesting
- Computing more things with the fundamental matrix

In general, these extensions are meant to either extend the mechanics of the game or to provide more interesting and diverse analysis to the data obtained. While my implementation provides a rudimentary basis on the game, these extensions look rather interesting too.

Midterm Project Review

In the Midterm project, I tried to maximize perceived utility as a student in Yale-NUS. This was motivated by the fact that time is a real scarcity in school. We try to do everything, but can potentially end up doing absolutely nothing. Now that we've hit the tail end of the semester and internships are coming soon, I think I can drive this project at a different tangent: optimizing my utility as an **intern**!

From the Midterm Project, I realized that a limitation of my implementation was that I stuck purely to linear programming since that was all I knew. This limited my ability to come up with constraints that accurately reflected my real life situation. In this project, I plan to change that with the newfound knowledge learnt from the second half of the semester.

As usual, I want to examine how my utility can be optimized as an intern of Datature. To do this problem, I will attach coefficients to the number of hours spent on work, leisure, sleep, commitments and meals everyday and briefly analyse the results. As usual, I wil be formulating constraints based on my personal experiences and observations on my weekly lifestyle. No data will be pulled from the internet. That being said, formulating these constraints will require a huge introspection on my living habits.

Problem Setting

To begin, allow me to first set up this problem up. Firstly, the sub-script, $i, i \in \{1, 2, 3, 4, 5, 6, 7\}$ attached to my decision variables refers to the i-th day of the week, starting from Monday. For now, I will define 35 decision variables:

- M_i refers to the number of hours spent on meals on the i-th day of the week
- C_i refers to the number of hours spent on compulsory commitments on the i-th day of the week
- L_i refers to the number of hours spent on leisure on the i-th day of the week
- W_i refers to the number of hours spent on work on the i-th day of the week
- S_i refers to the number of hours spent on sleep on the i-th day of the week

As you can see, the higher the coefficient, the higher the utility. I have ranked the coefficients for the following reasons, in decreasing order of utility ranking:

1. Sleep has the highest coefficient, because given my rather static work schedule, I'd like to maximize my rest in order to work in a sustainable manner.

2. Work has the next highest coefficient as I'm obligated to do so. In addition, I want to gain valuable work skills.
3. Leisure would be the next on the list. It's the summer holidays! It's the time to unwind and work on my hobbies.
4. Commitments is next because it's holiday period. I have enough obligations with my work; I do not want to pile up more.
5. Eating is the lowest on the list. I consume my meals alone and i finish it at a very rapid pace.

With this in mind, I wish to maximize the objective function

$$z = \left(\sum_{i=1}^{7} E_i\right) + 1.05\left(\sum_{i=1}^{7} C_i\right) + 1.1\left(\sum_{i=1}^{7} L_i\right) + 1.15\left(\sum_{i=1}^{7} W_i\right) + 1.2\left(\sum_{i=1}^{7} S_i\right).$$

However, because this objective function is a maximizing function and *nloptr* only has a minimizing function, the best that I can do is to multiply this objective function by -1 in R. The implementation of this is shown below:

```
obj <- function(x){
  total = 0
  for (i in 1:35){
    total <- total + (x[i] * (1 + 0.05 * ((i - 1) %% 5)))
  }
  return(total * (-1))
}
```

Here, there is something to note, in R, I cannot express my decision variables as programming variables, but indexes in an array. Therefore, I do not really have a choice but to track them in terms of indexes. Therefore, I've created a table showing the correspondence between these indexes:

	Mon	Tue	Wed	Thu	Fri	Sat	Sun
Meal	1	6	11	16	21	26	31
Commitment	2	7	12	17	22	27	32
Leisure	3	8	13	18	23	28	33
Work	4	9	14	19	24	29	34
Sleep	5	10	15	20	25	30	35

Please refer to this table whenever you encounter confusions to corresponding indexes with decision variables. Now that my objective function is finished, let's move on to some of the constraints.

Constraints - Sanity checks

Before moving on to the constraints, let's cover some sanity checks. The sign constraint is one of them:

$$\textbf{all } S_i, M_i, L_i, C_i, W_i \geq 0.$$

As usual, we also note that in a day, there's only 24 hours. Therefore, we need to include the constraint that the sum of time spent in all 5 activities in a day do not exceed 24 hours: for $i \in 1, 2, 3, 4, 5, 6, 7$,

$$S_i + M_i + L_i + C_i + W_i = 24.$$

With all these checks finished, it's time to go through the constraints relating to range.

Constraints - Ranges

When planning my time, I realized that compared to my time as a student, my schedule in work has becomes a lot more rigid than before. It might be due to the sheer number of hours I commit to weekdays, but I think that it'd be better to express the limitations of the amount of time spent on a certain activity on a given day as a range. Of course, I have to express it in $\leq$ form, which is done already in R. For now, allow me to go through the range constraints.

Meal

My range constraints for mealtimes are as follows:

$$\begin{aligned} 1 &\leq M_1 \leq 1.5 \\ 1 &\leq M_2 \leq 1.5 \\ 1 &\leq M_3 \leq 1.5 \\ 1 &\leq M_4 \leq 1.5 \\ 1 &\leq M_5 \leq 1.5 \\ 1.5 &\leq M_6 \leq 2.0 \\ 1.5 &\leq M_7 \leq 2.0 \end{aligned}$$

There is nothing much that needs to be said here. The amount of time that I spend on meals has not changed at all - it is still very short. Therefore, I do not take too long to eat everyday. However, in the weekends, I do appreciate taking the time to savour the food while away from work.

Commitment

My range constraints for Commitments are as follows:

$$
\begin{aligned}
2 \leq C_1 \leq 3 \\
0 \leq C_2 \leq 2 \\
0 \leq C_3 \leq 2 \\
2 \leq C_4 \leq 3 \\
0 \leq C_5 \leq 2 \\
5 \leq C_6 \leq 7 \\
5 \leq C_7 \leq 7
\end{aligned}
$$

Reflecting about it, I have lower commitments on Tuesday, Wednesday and Friday. This is because I'm not meeting anything in particular. However, Mondays and Thursdays happen to be the days that I gym, so I put it under commitments (Hopefully, I commit to it). My weekends tend to have greater commitment levels to my loved ones and church.

Leisure

My range constraints for leisure are as follows:

$$
\begin{aligned}
1 \leq L_1 \leq 2 \\
1 \leq L_2 \leq 2 \\
1 \leq L_3 \leq 2 \\
1 \leq L_4 \leq 2 \\
1 \leq L_5 \leq 2 \\
5 \leq L_6 \leq 8 \\
5 \leq L_7 \leq 8
\end{aligned}
$$

My justifications here are rather simple. Of course I'd like to have infinite leisure at my disposal, but that's simply impossible. However, one thing's for sure: I'm sure to get more break in the weekends than the weekdays. With that in mind, I simply made my bounds for my weekends higher!

Work

My range constraints for work are as follows:

$$\begin{aligned} 8 \leq W_1 &\leq 10 \\ 8 \leq W_2 &\leq 10 \\ 8 \leq W_3 &\leq 10 \\ 8 \leq W_4 &\leq 10 \\ 8 \leq W_5 &\leq 10 \\ W_6 &= 0 \\ W_7 &= 0 \end{aligned}$$

In terms of work, I know that I'm minimally working 40 hours a week at my company on the weekdays. However, I'm not working in the weekends for sure. Therefore this explains why $W_6, W_7 = 0$. That being said, I can never predict if I will have overtime work or not. Therefore, I have set 2 hours' worth of buffer just in case I have to work overtime.

Sleep

My range constraints for sleep are as follows:

$$\begin{aligned} 7.5 \leq S_1 &\leq 10 \\ 7.5 \leq S_2 &\leq 10 \\ 7.5 \leq S_3 &\leq 10 \\ 7.5 \leq S_4 &\leq 10 \\ 7.5 \leq S_5 &\leq 10 \\ 8.0 \leq S_6 &\leq 11 \\ 8.0 \leq S_7 &\leq 11 \end{aligned}$$

Here, I'd like my sleep to be as abundant and recharging as possible. Minimally, I know that I sleep about 7.5 to 8 hour daily to work productivity. However, I won't complain if I sleep even more. This is why I have put extra sleeping hours in my upper bound just in case I need that extra recharge.

Other (Non-)linear Constraints

In addition to my range constraints, I have a few other constraints that I've defined for the problem. Firstly, in the weekend, the square of the time spent on leisure might be equal or greater to the sum of the time spent in other activities. This equates to 2 constraints: one for Saturday, one for Sunday:

$$E_6 + C_6 + W_6 + S_6 - L_6^2 \leq 0$$
$$E_7 + C_7 + W_7 + S_7 - L_7^2 \leq 0$$

Next, I also want the variable of sleep and meal times to be below a certain amount. In order to do so, I have defined utility functions for calculating the variance of sleep and meals:

```
sleep_var <- function(x){
  # function to calculate sleep variance
  mean_sleep <- (x[5] + x[10] + x[15] + x[20] + x[35] + x[30] + x[35]) / 7
  var <- 0
  for (j in 1:7){
    var <- var + (x[5*j] - mean_sleep) ** 2
  }
  return (var / 7)
}

meal_var <- function(x){
  # function to calculate sleep variance
  mean_meal <- (x[1] + x[6] + x[11] + x[16] + x[21] + x[26] + x[31]) / 7
  var <- 0
  for (j in 1:7){
    var <- var + (x[5*j - 4] - mean_meal) ** 2
  }
  return (var / 7)
}
```

In the very end, I'd like the sleep variance to be less than 8 and the meal variance to be less than 6:

$$\frac{1}{7}\left(\sum_{i=1}^{7}(S_i - \overline{S})\right) \leq 8$$
$$\frac{1}{7}\left(\sum_{i=1}^{7}(M_i - \overline{M})\right) \leq 6$$

I have chosen these variances since I do not really want to fluctuate my sleep time too much. In addition, I do not take too long to eat, and I don't really like inconsistent eating times either. Eating too fast causes indigestion and eating too slowly takes up too much time.

Moving on, my next constraint is on my leisure time: I'd like at least 20 hours of leisure time a week:

$$20 - \sum_{i=1}^{7} L_i \leq 0.$$

Well, this was motivated by a lack of “spare time“ from my weekly schedule. Scheduling everything to the dot is very un-sustainable, and I'd like to do what I can to reverse this.

My next constraint has to do with the total number of hours slept versus the total number of hours slept versus the total number of hours worked. In this case, I'd like it such that the sum of squares of the number of hours spent working and sleeping is related in the following manner:

$$\sum_{i=1}^{7} W_i^2 - 0.85 \sum_{i=1}^{7} S_i^2 \leq 0.$$

This is a weirdly formulated constraint, bu the motivation is simple: I want my sleeping hours to be commensurate to the amount of hours I work. I realized that I get tired really easily, which justifies my extreme need for rest.

The last non-linear constraint that I will introduce is about my weekdays. I'd like to ensure that for my less busy weekdays (Tues, Weds, Fri), the sum of square of the time spent on mealtimes and leisure is more than or equal to the square of the time spent on commitment. In other words, for $i \in \{2, 3, 5\}$,

$$C_i^2 - M_i^2 - L_i^2 \leq 0.$$

Since these are more relaxed days, I prefer for this arrangement. However, Monday and Thursday are significantly more busy days, so I relax this assumption slightly: for $j \in \{1, 4\}$,

$$0.6C_j^2 - M_j^2 - L_j^2 \leq 0.$$

With these numerous constraints set in place, I am ready to see what the Non-Linear optimization method tells me about the kind of intern life I should lead.

Results & Analysis

The first time I ran the algorithm, it gave me these results (rounded to 3 significant figures):

	Mon	Tue	Wed	Thu	Fri	Sat	Sun
Meal	1.41	2.19	0.667	1.61	1.51	1.21	1.35
Commit	2.87	0.743	1.65	1.72	1.66	6.51	5.60
Leisure	2.47	1.92	2.45	1.13	1.09	7.48	7.14
Work	8.14	9.05	10.3	8.95	10.1	1.05	0.866
Sleep	9.12	9.31	8.83	9.68	9.35	8.14	8.45

In a very interesting twist of events, I could not get the algorithm to converge and agree on a certain number. While these numbers shown above look realistic, some constraints were actually violated slightly. I'll name some obvious ones:

- For Saturday, the sum of time spent slightly exceeds 24 hours
- For Saturday and Sunday, the amount of time spent working is non-zero.

If the algorithm could not converge, then that could mean a few things. Firstly, my constraints could be too strict and clashing in such a way that there are no values that obey the entire set of constraints. Alternatively, my constraints could be so relaxed that the algorithm does not know where to converge towards. However, given the sheer number of constraints that I created, I feel that it is the former case.

That being said, I am not very sure about the numerical analysis method conducted by the non-linear optimization algorithm, so I cannot be exactly sure about it. However, this has already been quite a learning experience. Let's go one step further to see whether I can fix this.

Given that the clashes occur at work, I feel that the constraint that might be too strict is the one about the sleep difference:

$$\sum_{i=1}^{7} W_i^2 - 0.85 \sum_{i=1}^{7} S_i^2 \leq 0.$$

Allow me to relax this constraint a little. In order to make it earlier for this constraint to converge, we need to increase the coefficient for sleep to relax this constraint:

$$\sum_{i=1}^{7} W_i^2 - 0.95 \sum_{i=1}^{7} S_i^2 \leq 0.$$

With that done, let's run the algorithm and see whether there are any improvements to the result agreeing with the constraints:

	Mon	Tue	Wed	Thu	Fri	Sat	Sun
Meal	1.10	1.55	1.90	2.45	1.70	0.98	1.06
Commit	2.47	2.42	1.71	3.24	1.66	4.79	5.00
Leisure	2.77	1.79	2.06	1.01	1.58	6.03	8.36
Work	8.87	7.92	8.46	7.72	8.48	0.691	0.392
Sleep	9.31	10.05	10.0	9.52	10.5	11.1	9.12

As you can see, compared to the previous table of values, there is measurably better agreement than before. While it is not completely obeying the constraints that I've set, it's close enough that I can tolerate it. Running this repeatedly might produce slight deviations from results due to a lack of convergence, but the numbers should be close.

After some reflection, perhaps this deviation from expected results could be something about the numerical analysis that has to be done in order to make this problem work. After all, it is a problem in 35 dimensions. However, anything that happens within this numerical algorithm is a black box to me. That being said, this serves as a great learning experience for me to understand the complexities of non-linear optimization algorithms relative to linear ones.

Assumptions & Further Studies

In this particular mini-project, I've expanded on my midterm project by including non-linear constraints which made use of some basic statistics. However, there were a few roadblocks that I encountered in the project.

Firstly, I initially tried to define my objective function as the sum of variances of the utility I got per day, but I quickly realized that that function did not run no matter how many parameters I tweaked. Therefore, I had to work around that by doing a utility function that is linear in nature, but with non-linear constraints. If I had more time, I'd investigate how to make a complex objective function that can converge.

Secondly, I didn't really randomize these constraints, which meant that the schedule I created is purely static. In the future, I hope to be able to create something more dynamic with random variables that make analysis more interesting.

Conclusion

We have finally come to the tail end of this term project. In it, I have modelled the snakes and ladders game using a Markov Chain. While it wasn't really practical, I hope that it provided some interesting findings for you.

Secondly, I have re-looked at my midterm project and implemented non-linear constraints. While the function struggled significantly to find convergence whilst satisfying all constraints, I learnt a lot about the complexities of forming non-linear programs to solve real-life problems.

Going forward, I look forward to using concepts I've learnt from this course in computer science or mathematical problems. Indeed, modelling real situations and optimizing decisions is a very salient concept, but I will remember to wield it with responsibility. Thank you for the course!