Cedric Liang 29674662

Part 1

a. I've decided to build a class called ArrayFloat, with which I've used three instance variables - self.sign, self.exponent, and self.array. This technically isn't a single array in the traditional 'single block of memory' sense of the word, but it's a representation that is much easier to work with in indexing terms, and doesn't change the details of implementing the addition and subtraction algorithms. If it bothers you at all, feel free to imagine adding the sign and exponent as the first two elements every time you see self.array, but also keep in mind that it would result in having to do annoying stuff like self.array[0] every time I want to access the sign.

Noting the above, in this report I'll refer to my array representation of a float with the following format:

$$[array],\ sign,\ exponent$$

The key part is that I have a single character that denotes the signage of my float, and an integer that represents the exponentiation that needs to be applied to elements of the array in order to reproduce the original number.

For example,

- [1, 2, 3] with a sign of "+" and exponent of 0 is equivalent to $+123\ \times\ 10^{0}$
- [1, 2, 3] with a sign of "-" and exponent of 3 is equivalent to $-123\ \times\ 10^{3}$
- [1, 2, 3] with a sign of "+" and exponent of -20 is equivalent to $-123\ \times\ 10^{-20}$

The key to note here is that I ensure that each real number has a unique representation, by implementing procedures for removing and adding leading zeros when required for serialisation and deserialisation.

As such, the array representation of 123 is ONLY $[1,\ 2,\ 3],\ +,\ 0$.

$[1,\ 2,\ 3,\ 0],\ +,\ -1$ is a calculation that would result in the same original float, and it can be a temporary representation of the float if I need to perform an addition with another ArrayFloat with an exponent value of -1, but every procedure I perform always restores the representation back to the correct representation with no leading or trailing zeros.

The ArrayFloat also has the option to set a precision. This is the number of significant figures that is stored. For example, for a precision of 5, the following strings would be converted to the corresponding representations:

- "1.01" → [1, 0, 1], +, -2
- "0001.010000000" → [1, 0, 1], +, -2
- "1.001" → [1, 0, 0, 1], +, -3
- "-0001.0001" → [1, 0, 0 , 0, 1], -, -4
- "1.01001" → [1, 0, 1], +, -2

As such, my representation isn't really storing the position of the decimal at all, only an exponent describing the order of magnitude at which it resides. Conversion is easy as a result - given a representation:

[1, 2, 3 , 7, 1], +, 3

We can quickly derive the place values of each digit by first creating a parallel array of reversed indices for the digit array component:

[1, 2, 3, 7, 1]
[4, 3, 2, 1, 0]

Then, the value of each digit is given by the sum of the value of that parallel array and the exponent. For example, the value of the first 1 is $1 \times 10^{4+3}$, the value of the 2 is $2 \times 10^{3+3}$, the value of the 3 is $3 \times 10^{2+3}$, and so on.

In other words, the decimal point is implied to begin at the end of the array (12371.), and the exponent provides a procedure to move the decimal point to the correct position. A positive value like 3 implies moving the decimal point rightwards three places (padding with zeroes if required), resulting in 12371000., whereas a negative value like -6 implies moving the decimal point leftwards six places places, yielding 0.012371.

Since I used Python classes, using these classes is straightforward - just create an object with a string representation like ArrayFloat("-1.23", precision=50).

One thing to note - I implemented a __str__ and a __repr__ method that generates a representation of the ArrayFloat by calling float(). This is just for the sake of convenience - the float method can be prone to rounding error and in many cases doesn't represent what the ArrayFloat actually contains, especially if the precision is set high. I didn't bother implementing a more precise __str__ method, but I would recommend using the equality operators to compare values instead of relying on the string representation.

For example, compare an arrayfloat *a* to zero with 'if a == ArrayFloat("0", precision=50)'.

b. For the implementation of addition, I first noted the property:

If a is positive and b is positive,
$$a + b = |a| + |b|$$

If a is negative and b is negative,
$$a + b = -(|a| + |b|)$$

As such, we can implement an addition procedure here. We extend the two arrays (or rather deepcopies of them to avoid mutation) such that the two arrays are of equal length, and perform a simple arithmetic addition with carrying, as one would do by hand. Then, the resulting array has its leading and trailing zeroes removed and the new value is returned.

Note that the precision limit applies for this calculation! I can save some computation by checking for the *magnitude ranges* of the two values. For example, [1, 2, 3], +, 3 has a magnitude range of 3 to 5, since its digits span the orders of magnitude 10^3 to 10^5. This range is compared to the range of the other ArrayFloat, and the overlap is computed - but any elements with a magnitude more than the precision limit lesser than the upper limit of the magnitude range is discarded.

As such, adding $[1,\ 2,\ 3],\ +,\ 10$ to $[1,\ 2,\ 3],\ +,\ -10$ with a precision limit of 5 would result in the digits in the second ArrayFloat being discarded, and $[1,\ 2,\ 3],\ +,\ 10$ returned.

For the case of adding a negative number to a positive number, it instead uses a different subtraction procedure. We note that for the operation $a - b$,

If $|a| > |b|$,

$$a - b = |a| - |b|$$

If $|a| < |b|$,

$$a - b = -(|b| - |a|)$$

With this, we calculate $a - b$ by comparing the absolute magnitudes, then doing a subtraction of the smaller magnitude number from the larger with a simple procedure (again as if it were done by hand), then assigning the signage as required.

c. Here, we first create a multiplication table of the ArrayFloat products of the digits form 0 to 9 by hand, so that we can refer to these without cheating! I wasn't sure if we were permitted to use inbuild functionality for one digit multiplication, so I wrote out the multiplication table.

From here, it's pretty simple to create a sum of the products of the digits - we need to create a Cartesian product using itertools.product between the two arrays, but also computing the place value of each digit in each ArrayFloat using the procedure in part a). When the product is computed between those digits, the resulting exponent is the sum of the exponents of the two digits. It is then added to an array to be summed, and the addition function is used from there.

d. We note that the iterative step in Newton's method is given by

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

Here, we compute the derivative of $f(x) = \frac{1}{x^2} - 2$:

$$f'(x) = -2x^{-3}$$

Then, we can reduce $\frac{f(x_i)}{f'(x_i)}$:

$$\frac{f(x_i)}{f'(x_i)} = \frac{\frac{1}{x^2} - 2}{-2x^{-3}} = x^3 - \frac{x}{2}$$

Then, $x_{i+1} = x_i - \left(x^3 - \frac{x}{2}\right)$

We have (I'm not sure how many digits that is)

0.70710678118654752440084436210484903928483593768847403658833986899536

The method has a tolerance that can be set as the stopping condition. We stop if

```
new_x_func(x).abs() < tol
```

That is, if the magnitude of the $\left(x^3 - \frac{x}{2}\right)$ term is less than the tolerance. For a convergent system, we expect the deviation from the true value at every iteration to decrease, and that term constitutes the 'adjustment' to the previous term that we have to make. If this adjustment reaches a level of tolerance, then we can be certain that no later timestep has the capacity to impact the correctness of digits obtained from previous iterations.

We also can be pretty sure of our correctness if we set the precision absurdly high - like 100.

e. I've implemented the polynomial as a function, along with a k parameter that offsets the calculation as required. I then iterate starting at a reasonable value like -50, plugging in the values until I get a bunch of zeros, or terminating calculation if I get a root that isn't a zero.

   I find a value for k of 12, and at this point, we yield 0 for all the roots.

Part 2

a. Let's state the baseline SEIR model first:

$$\frac{dS}{dt} = -\beta IS$$
$$\frac{dE}{dt} = \beta IS - \eta E$$
$$\frac{dI}{dt} = \eta E - \alpha I$$
$$\frac{dR}{dt} = \alpha I$$

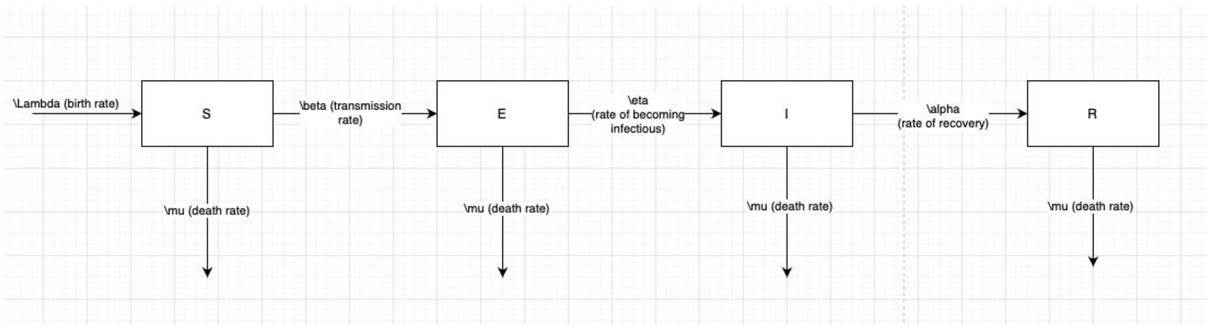
Let's reformulate this with the assumption that individuals are born susceptible (no maternally derived immunity), and that the death rate is constant (ie, the same for all compartments). Let's also assume that the disease has no impact on birth rate, and that all people in all compartments are eligible to reproduce. We can then note the following:

- The total population at a given point in time is given by $N(t) = S(t) + E(t) + I(t) + R(t)$
- The entry point into the system is only to compartment S
- The exit point from the system is from all compartments

We can now write the following:

$$\begin{aligned}\frac{dS}{dt} &= \Lambda(S + E + I + R) - \beta IS - \mu S \\ &= \Lambda(S + E + I + R) - (\beta I + \mu)S \\ \frac{dE}{dt} &= \beta IS - \eta E - \mu E \\ &= \beta IS - (\eta + \mu)E \\ \frac{dI}{dt} &= \eta E - \alpha I - \mu I \\ &= \eta E - (\alpha + \mu)I \\ \frac{dR}{dt} &= \alpha I - \mu R\end{aligned}$$

This is our reformulated system. We can draw a diagram that shows our compartments and the parameters for their state transitions.



Given that we haven't described any interactions between the dynamics of the disease and the death rate, we should expect the total population to only depend on the birth rate and the death rate.

We can express this mathematically - if the total population is given by N, then we can use the multivariate chain rule to write the following:

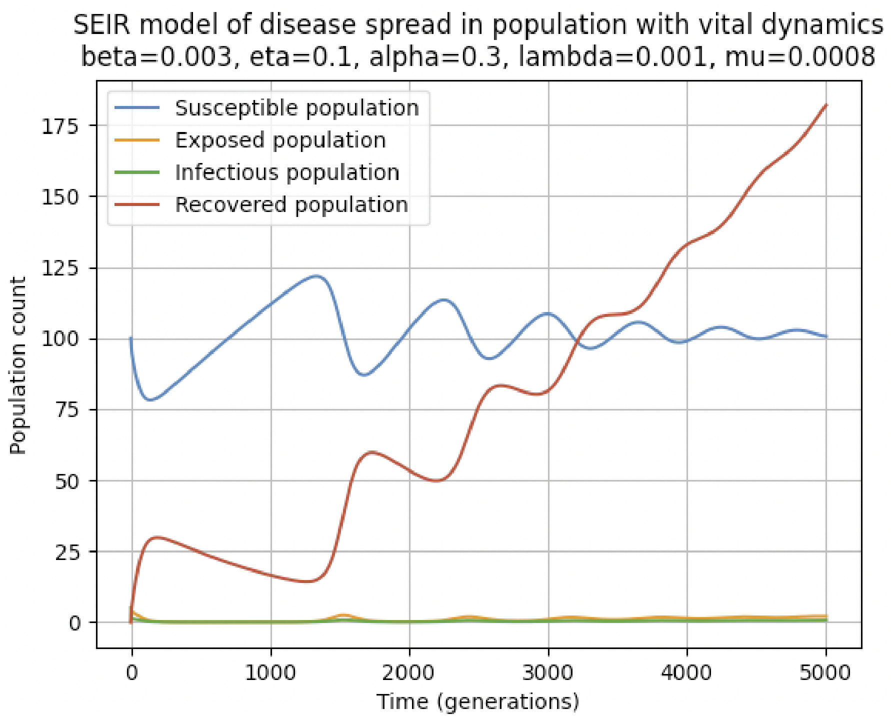
$$\begin{aligned}
N(S, E, I, R) &= S(t) + E(t) + I(t) + R(t) \\
\Rightarrow \frac{dN}{dt} &= \frac{\partial N}{\partial S}\frac{dS}{dt} + \frac{\partial N}{\partial E}\frac{dE}{dt} + \frac{\partial N}{\partial I}\frac{dI}{dt} + \frac{\partial N}{\partial R}\frac{dR}{dt} \\
&= \frac{dS}{dt} + \frac{dE}{dt} + \frac{dI}{dt} + \frac{dR}{dt} \\
&= (\Lambda(S + E + I + R) - \beta IS - \mu S) + (\beta IS - \eta E - \mu E) + (\eta E - \alpha I - \mu I) \\
&\quad + (\alpha I - \mu R) \\
&= \Lambda N - \mu S - \mu E - \mu I - \mu R \\
&= (\Lambda - \mu)N
\end{aligned}$$

This really should confirm our intuition - we've treated the entire system as a black box, disregarded internal interactions, and given our constant death rate across all containers, we expect the death rate of the system to also be proportional to *N.*

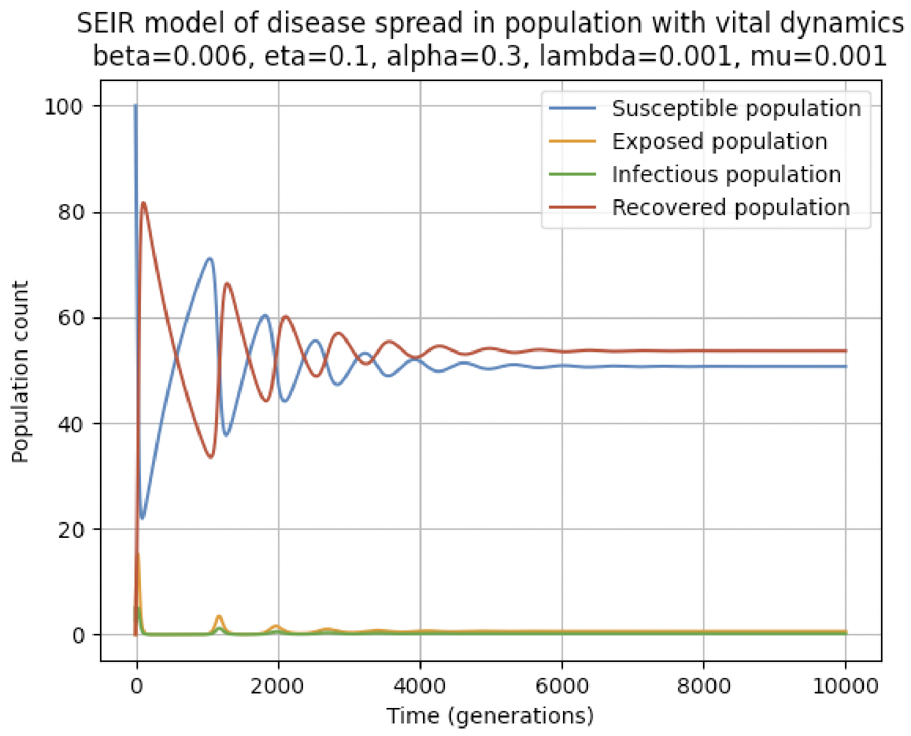
As such, if we write the growth rate of the system as $R = \Lambda - \mu$, then we would still expect the system to behave according to the closed form solution which is exponential:

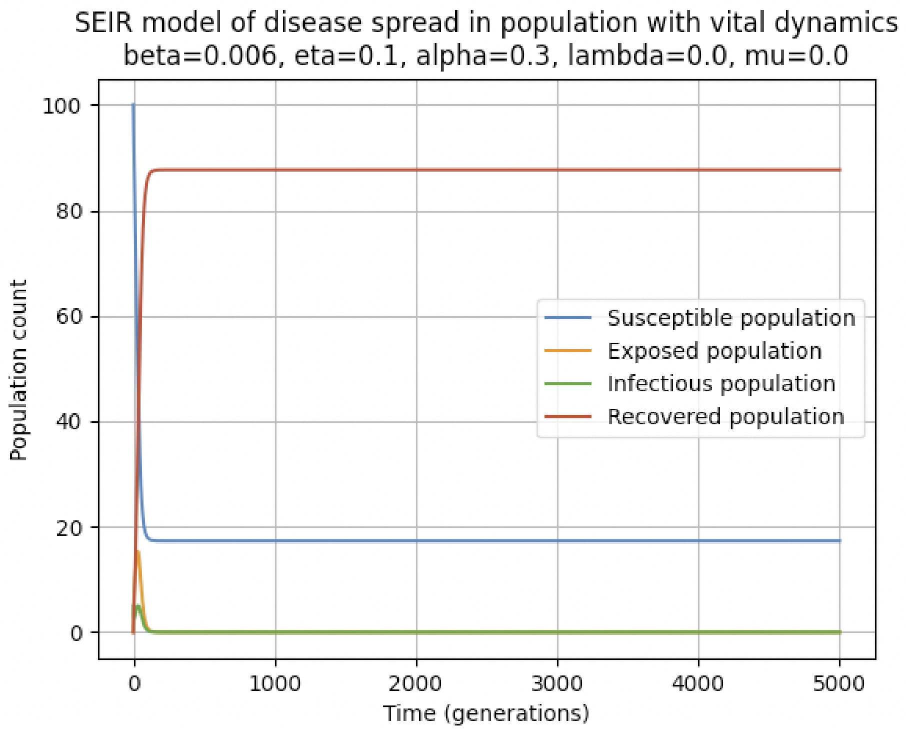
$$N(t) = N_0 e^{(\Lambda - \mu)t}$$

Let's take a look at the qualitative change in the system over time. An example of how one system might evolve is as follows:



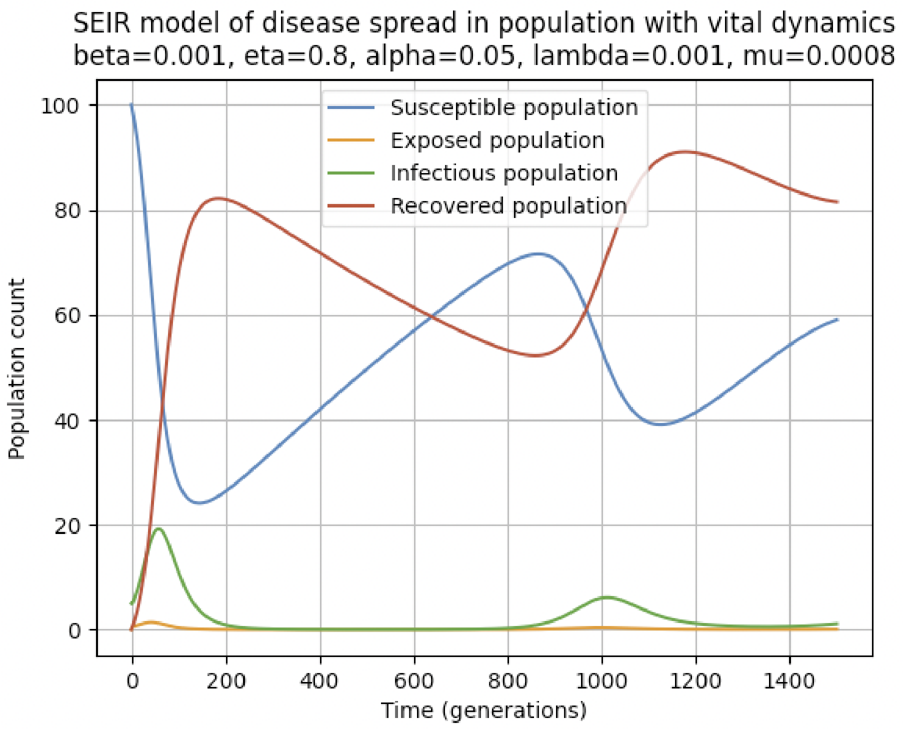
A key thing to notice here is that if we set $\Lambda = \mu$, the overall population of our system does not change over time, but we do have a qualitative change to the internal configuration of the system depending on what the values of Λ and μ are. For example, these two systems have very different evolutions over the long run:





Our overall population does not change over time, but the behaviour of the system is drastically different! This is understandable, as the birth and death rates in this case would constitute a re-assignment from each of the compartments back to the susceptible compartment.

The key difference between the version with vital dynamics is the possibility of seeing oscillatory behaviour as seen above:



The basic SEIR model has no mechanism for recovered populations to be reinfected. Being infected is a 'one-and-done' thing, and as such, there is no possibility for more complex long term evolutions like we see here. Either the disease takes hold, or it doesn't, and a long term equilibrium is reached.

Qualitatively, this manifests as us only ever seeing one 'hump', and that is understandable as the flow of population amongst the compartments in that type of configuration is one-directional.

Another way to think of this is the base SEIR model has our term for $\frac{dS}{dt}$ as follows:

$$\frac{dS}{dt} = -\beta IS$$

Given that all the variables on the right are positive, it follows that $\frac{dS}{dt}$ must be negative. In other words, there is no mechanism for the change in the susceptible population to inflect back into the positive (and result in the population of S growing). Given that the other compartments directly or indirectly rely only on their previous compartments as inflow, if there is evolution in this system, it is irreversible.

An introduction of population 'recirculation' changes this, and allows for long term oscillatory behaviour.

b. Let's restate our model from part a)

$$\frac{dS}{dt} = \Lambda N - (\beta I + \mu)S$$
$$\frac{dE}{dt} = \beta IS - (\eta + \mu)E$$
$$\frac{dI}{dt} = \eta E - (\alpha + \mu)I$$
$$\frac{dR}{dt} = \alpha I - \mu R$$

We are required to split compartment I into compartments I and A. Let's formulate this. Our transition from S to E is now dependent on both the populations of S and A.

$$\frac{dS}{dt} = \Lambda N - (\beta I + qA + \mu)S$$

Our inflow to E now is equivalent to the outflow of S (apart from the death term). Note that this is still population's outflow is still only modulated by η, since that is still the term that provides our per-capita rate of transition over time! In other words, even though we have outflows to compartment I and A, since they are modulated by coefficients that sum to 1, the total outflow is still only dependent on η. We can 'split' the flow when we are working out the next transition.

$$\frac{dE}{dt} = S(\beta I + qA) - (\eta + \mu)E$$

We can now describe the inflows to our I and A compartments, making sure to add the probability term to the non-death outflow of E.

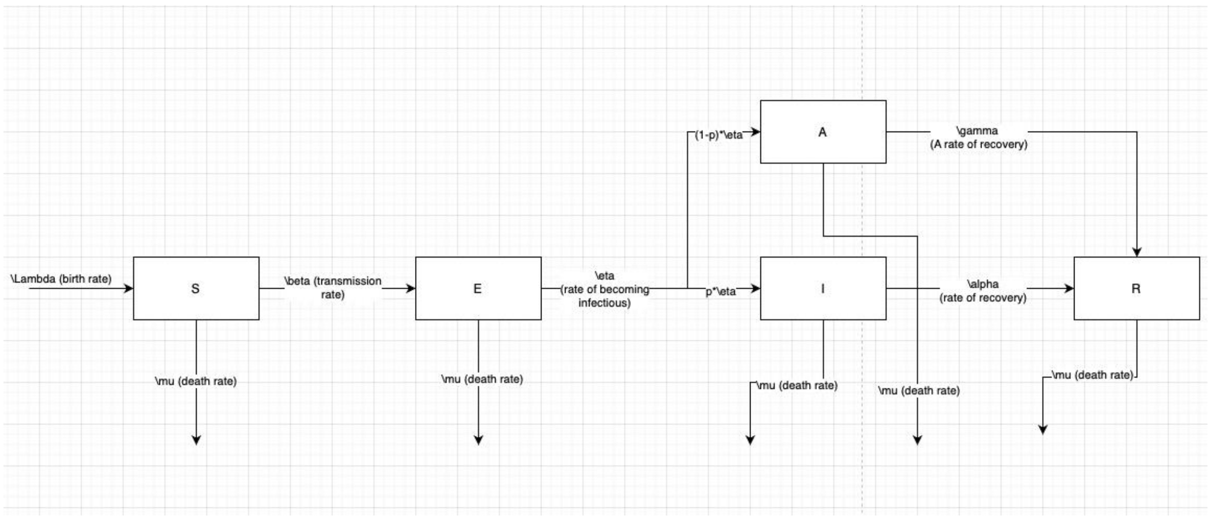
$$\frac{dI}{dt} = p\eta E - (\alpha + \mu)I$$
$$\frac{dA}{dt} = (1 - p)\eta E - (\gamma + \mu)A$$

Finally, we can use the outflows of A and I to describe R.

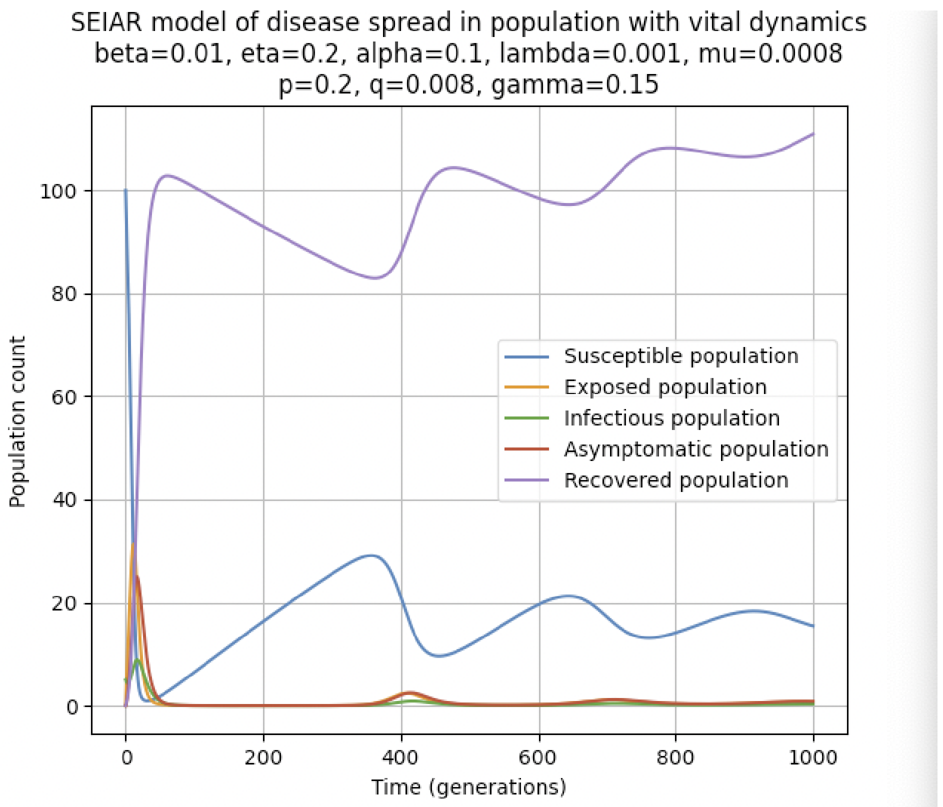
$$\frac{dR}{dt} = \alpha I + \gamma A - \mu R$$

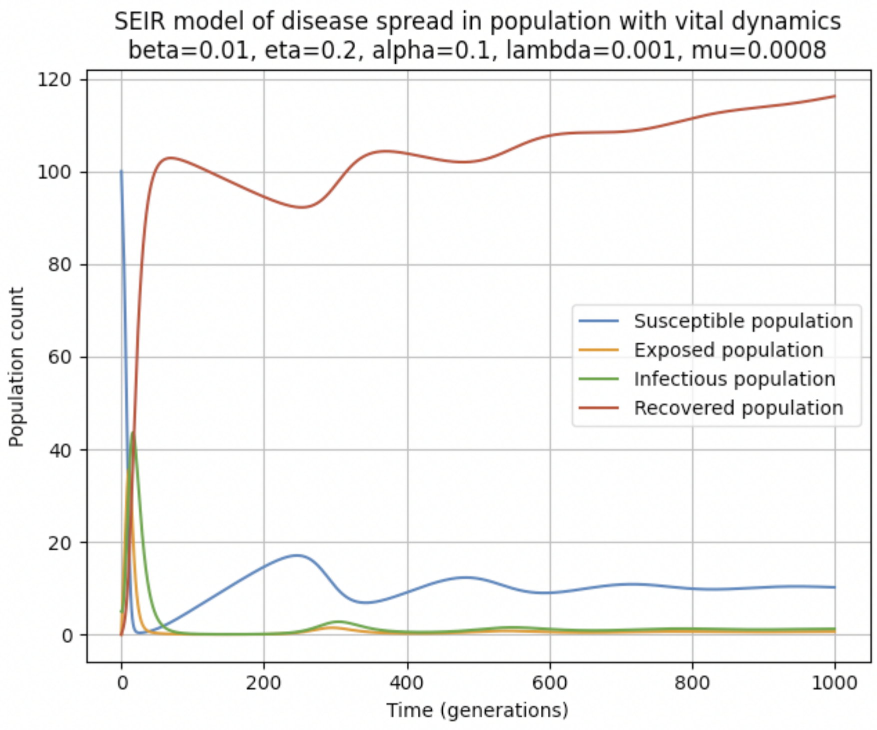
Putting this together, we have our set of differential equations describing the system:

$$\frac{dS}{dt} = \Lambda N - (\beta I + qA + \mu)S$$
$$\frac{dE}{dt} = S(\beta I + qA) - (\eta + \mu)E$$
$$\frac{dI}{dt} = p\eta E - (\alpha + \mu)I$$
$$\frac{dA}{dt} = (1 - p)\eta E - (\gamma + \mu)A$$
$$\frac{dR}{dt} = \alpha I + \gamma A - \mu R$$



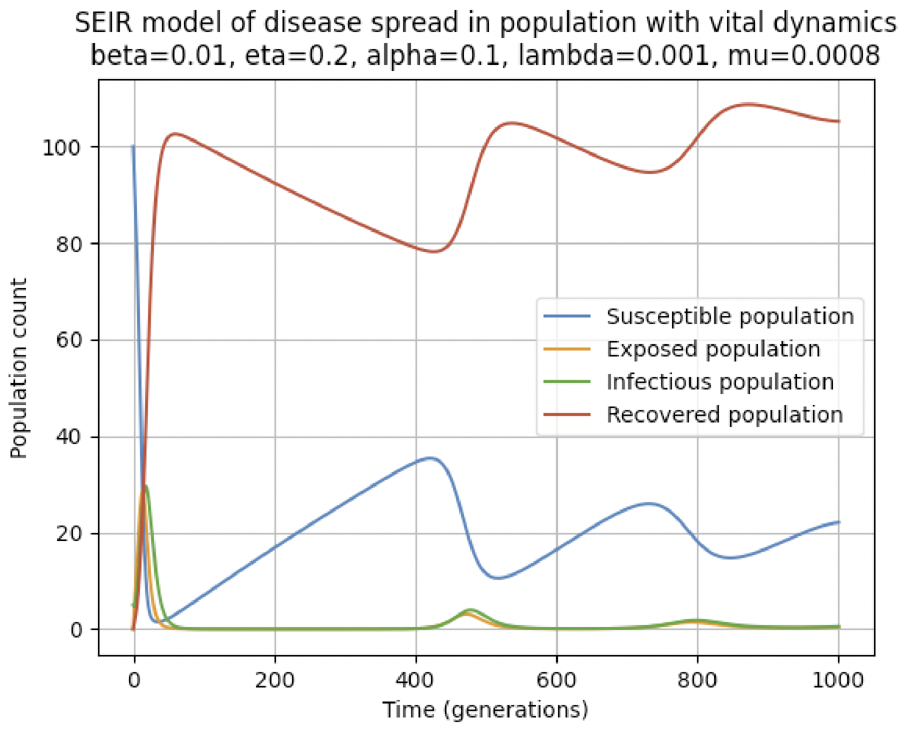
Let's take a look at the dynamics of the system:



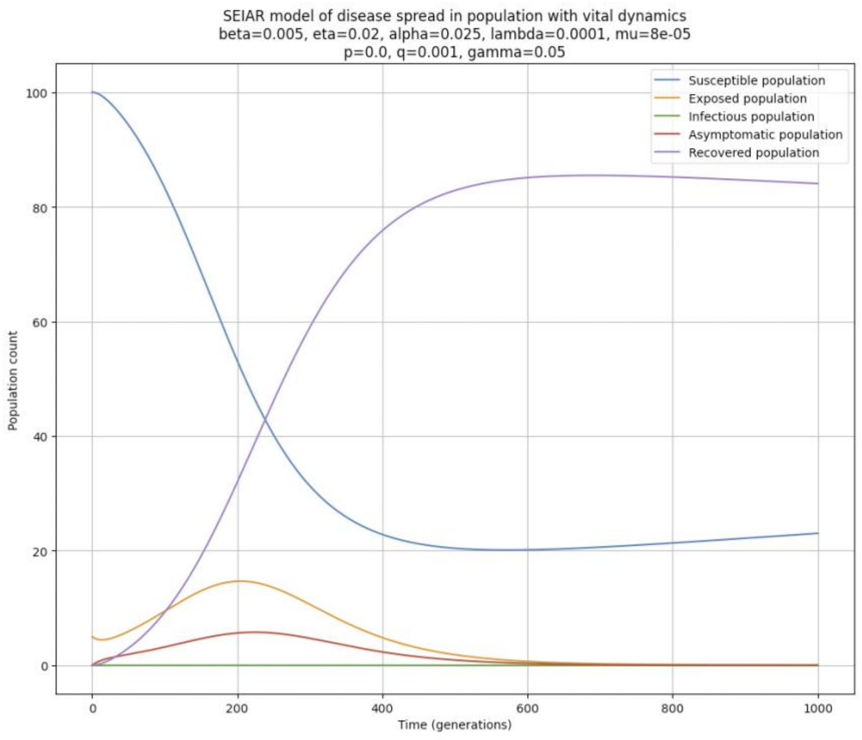


The first image is that of a SEIAR system, whilst the second is that of an SEIR system (or alternatively, an SEIAR system where p is 1). There isn't too much qualitative difference at this timescale - the SEIR system tends towards an equilibrium where more people have been infected and recovered, and the susceptible population converges towards a lower value, but that's about it.

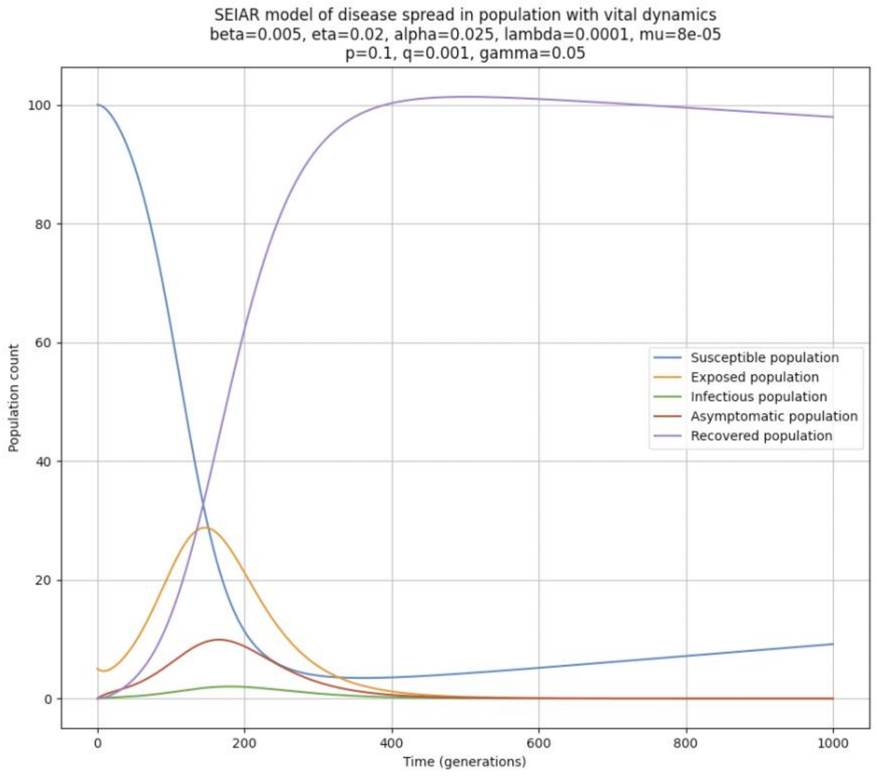
The differences are even less apparent when we compare the SEIAR system with a SEIR system with *transmission parameters equal to that of the asymptomatic parameters in the SEIAR system*, or equivalently, setting *p* to 0. The behaviour is almost exactly the same!



Let's take a look at another situation - simulating some 'super spreaders'. We'll make our transmission parameters for symptomatic transmission five times higher than asymptomatic transmission, with half the rate of recovery, but say that the probability of becoming a super spreader is only 10%. First, our spread if we have no super spreaders:

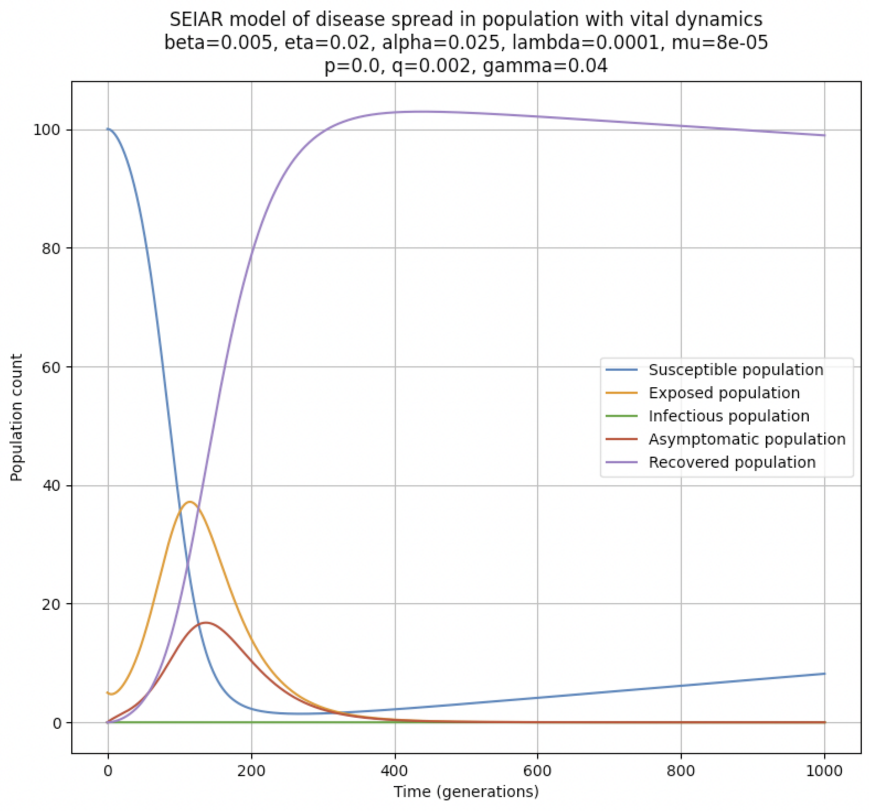


And now introducing super spreaders:



Our peaks are noticeably sharper, and we end up with a greater portion of the population having contracted the disease! Even though only 10% of the population are super spreaders, the proportion of our population who have contracted the disease increases significantly, and our peaks are much more compressed with higher peak amplitude and shorter tails.

Still - it's completely possible to reproduce this behaviour even without superspreaders if we pick transmission criteria somewhere in the middle:



Here we have no super spreaders, but our disease is twice as infectious as in the previous no super spreaders case. Our behaviour is quite similar to the case with a less infectious disease but with super spreaders.

The bottom line is that introducing a compartment *A* for asymptomatic transmission hasn't impacted the evolution of our system by a great deal - it's roughly equivalent to decreasing the overall transmissibility of the disease a bit.

Of course, this set of behaviours is true because

> a) - our system is continuous, which means that the 'probability' term we introduced doesn't introduce any true stochastic/probabilistic aspects to our model, but rather gets abstracted away into smooth but less realistic calculations with fractional humans. In reality, a single super spreader event and a bit of bad luck can kickstart an outbreak.
>
> b) we set the transmissibility of asymptomatic illness to be lower than that of symptomatic illness. The real challenge with asymptomatic transmission is the opposite - people are much more likely to go undetected with the disease and thus spread it. To simulate this behaviour accurately, we would have to introduce additional parameters, such as a 'mobility coefficient', or something similar.

Part 3

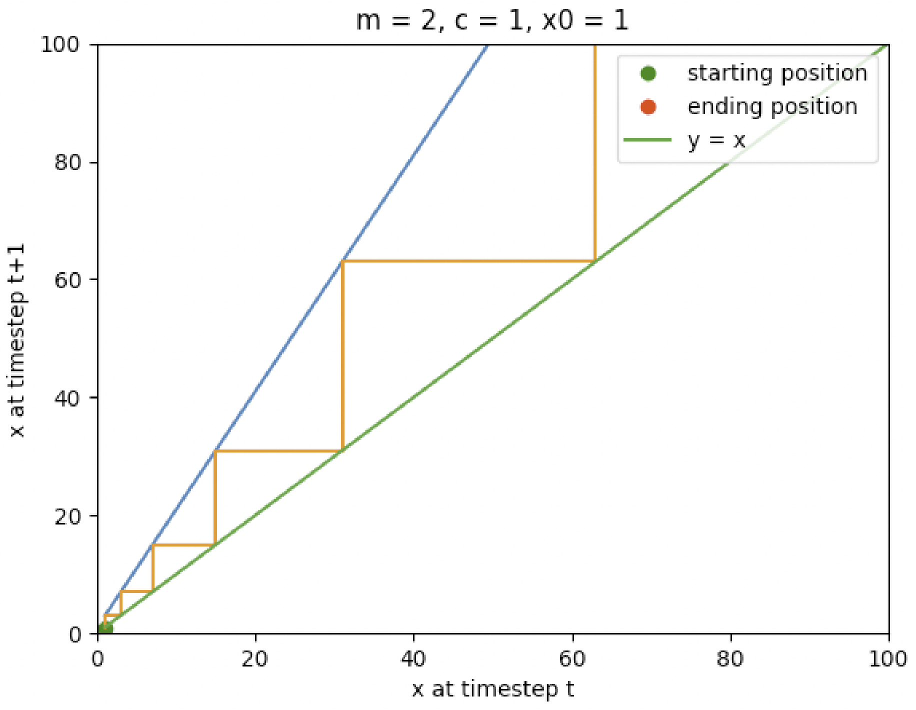
a. We can find a steady state and the conditions that it requires by setting $x_{t+1} = x_t$.

$$x^\star = mx^\star + c$$
$$\Rightarrow x^\star = \frac{c}{(1-m)}$$

We need $m \neq 1$ in order for a steady state to exist. Otherwise, every other configuration of the system for values of m and c will have exactly one steady state.

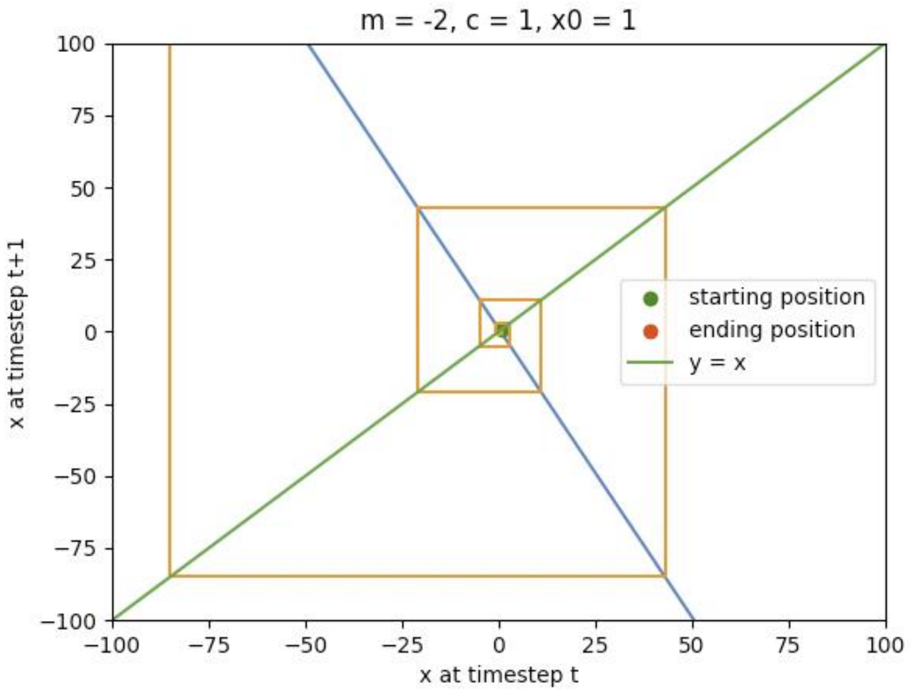
Of course, whether this steady state is reachable by iterating from an initial condition that isn't $x^\star$ is another question entirely.

b. Here, we have the m = 2 case, with c = 1 and $x_0 = 1$.



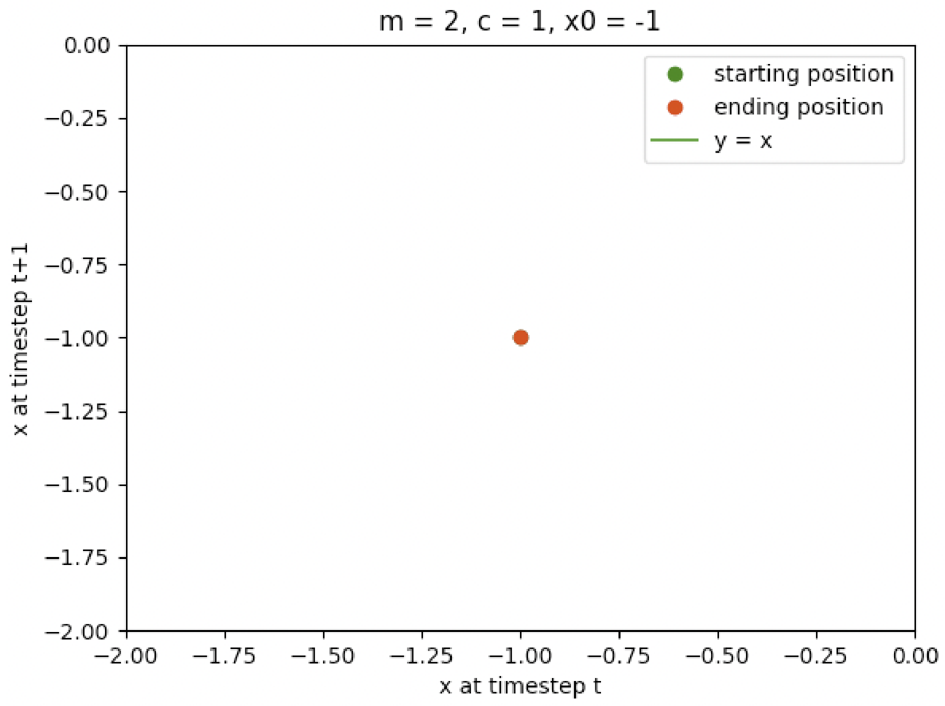
We experience unbounded exponential growth - and this is not surprising. Our steady state in this situation is given by a negative number (-1), and since we start at at a positive x value and our difference equation increases our x_0 at every timestep, we find ourselves diverging to infinity quite quickly.

Interestingly, if we set $m = -2$, we still have exponential divergence, but this time it alternates between a positive and negative value:

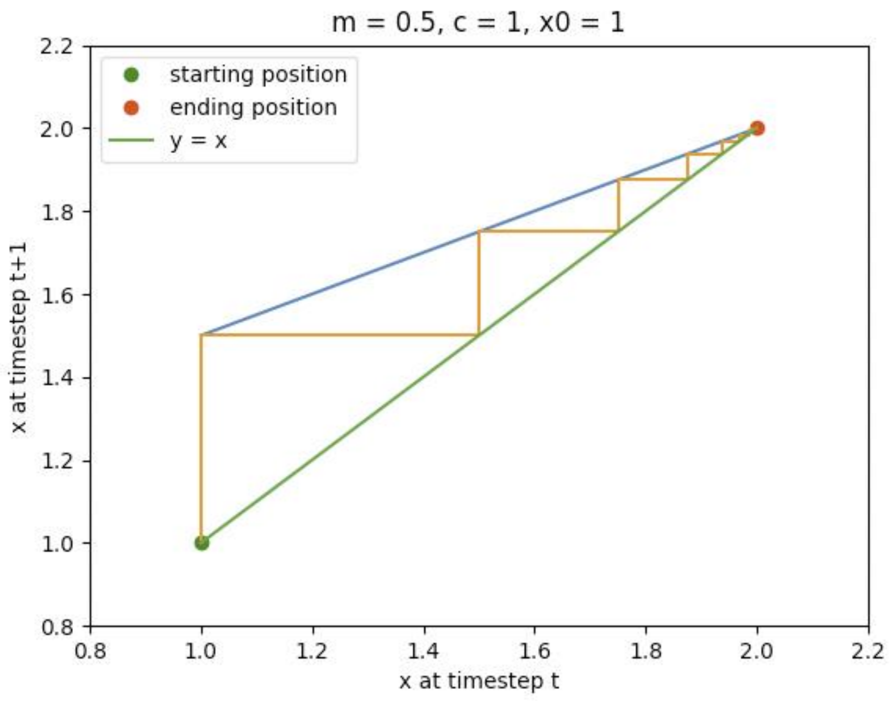


This is because a constant factor of m applied at every iteration ensures that we have enough 'swing' to overpower the previous iteration's displacement from the origin.

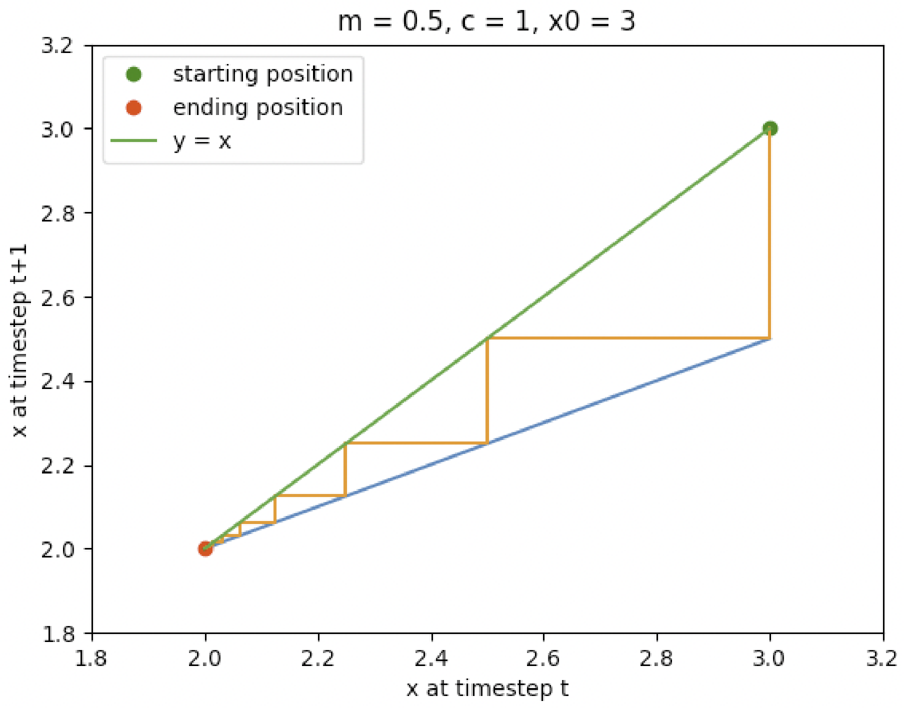
On the other hand, keeping the same values and setting our initial position to our steady state of $x = -1$ yields no movement as expected.



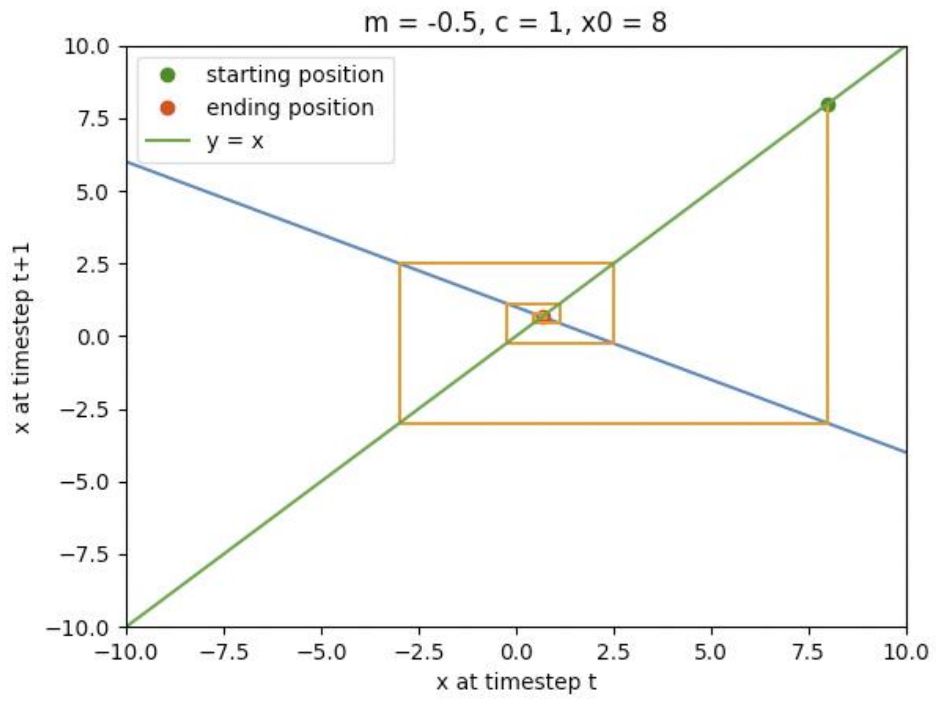
In the $m = 0.5$ case, we have the following cobweb diagram:



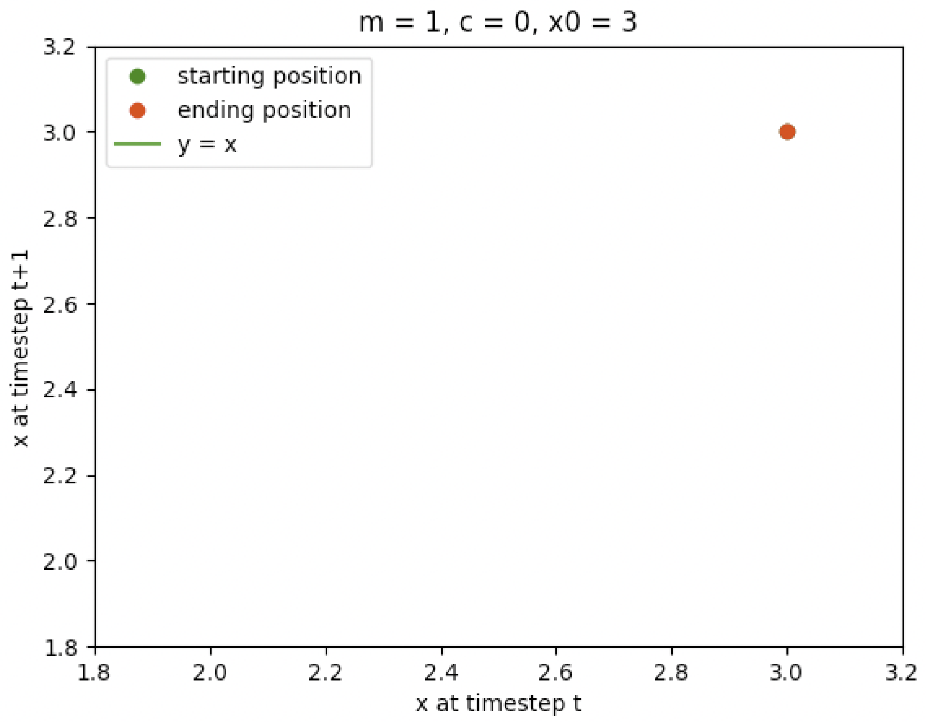
Here, we have a steady state implied by part a) of 2, and we find ourselves tending towards this point, eventually converging there. This convergence happens regardless of our initial position - here is the case where our initial position is greater than the steady state position.



If we set $m = -0.5$, we still have convergence, but this time with oscillatory behaviour:



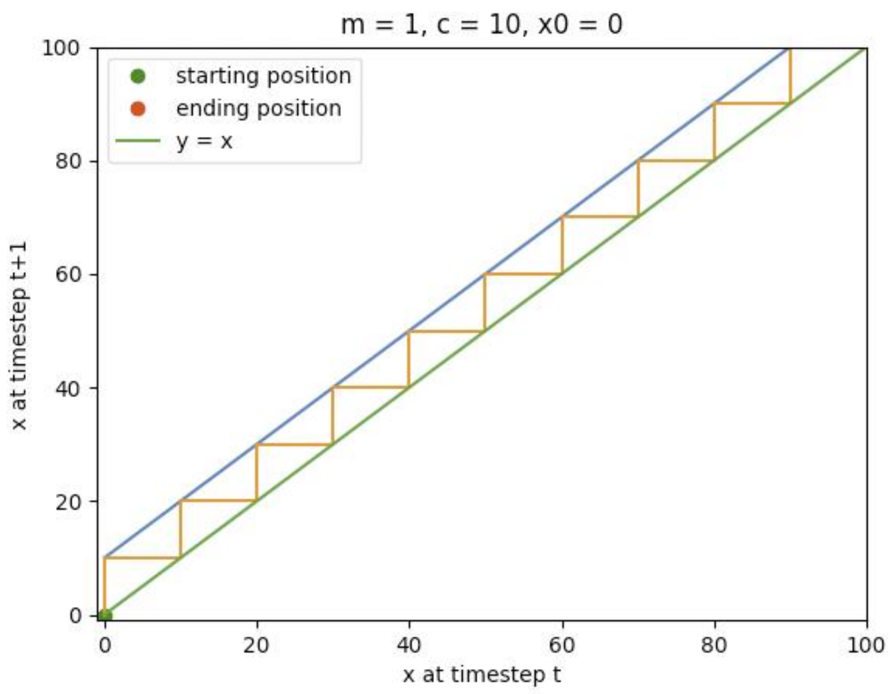
In the case $m = 1$ and $c = 0$, we note that the analysis in part a) implies that we should expect to have no steady state. Yet our system behaves like it does:



Of course it should - a system where $m = 1$ and $c = 0$ implies that our difference equation is $x_{t+1} = x_t$, which understandibly yields no movement. This demonstrates that cases arising from analysis like that in part a) can occasionally cause weird mathematical 'bugs', and the system at these singularities should be analysed on a case-by-case basis.

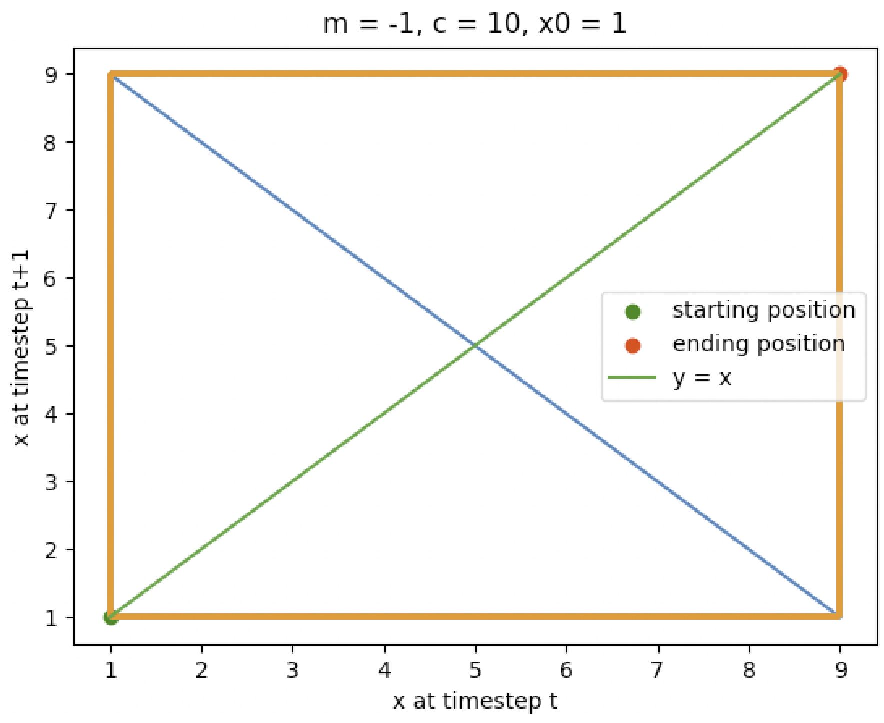
A more rigorous approach might be to analyse some limit as m and c go to zero, and why we end up with a sensical steady state (implying the $c \to 0$ term dominates the fraction) rather than a $0 \div 0$ singularity - but that's not for me to tackle.

Setting c to a nonzero value is more straightforward:



No steady state, no convergence, ever. Once again, this is understandable, as the difference equation boils down to adding a constant every iteration.

In the $m = -1$ case, we have oscillations between two values:



This is a similar reasoning as the $m < -1$ case with the divergent exponential oscillations, except that each iteration introduces just enough of a correction to bring us exactly back to the first point, and vice versa. As a result, we get this perpetual oscillation between two values.

c. Let's use a Taylor polynomial of $f(x^\star)$. Assuming we are near the steady state, we can approximate

$$x_{t+1} = f(x_t) \approx f(x^\star) + f'(x^\star)(x_t - x^\star) + \cdots$$

We can use the property of $f(x^\star) = x^\star$. In addition, let's truncate any terms after this point:

$$x_{t+1} \approx x^\star + f'(x^\star)(x_t - x^\star)$$

Dividing through by $x^\star$ yields us

$$\frac{x_{t+1}}{x^\star} \approx 1 + f'(x^\star)\left(\frac{x_t}{x^\star} - 1\right)$$

Now let's introduce the substitution $X_t = \frac{x_t}{x^\star}$. This means our steady state is now at $X^\star = 1$, and in addition we have

$$X_{t+1} \approx 1 + f'(x^\star)(X_t - 1)$$

Making another substitution of $\delta_t = X_t - 1$, we have a linear system where the steady state is 0.

$$\delta_{t+1} + 1 \approx 1 + f'(x^\star)\delta_t$$
$$\Rightarrow \delta_{t+1} \approx f'(x^\star)\delta_t$$

The closed form solution of this system is given by

$$\delta_{t+1} \approx f'(x^\star)^{t+1}\delta_0$$

Undoing the substitutions, we have

$$\begin{aligned} X_{t+1} - 1 &\approx f'(x^\star)^{t+1}(X_0 - 1) \\ \frac{x_{t+1}}{x^\star} - 1 &\approx f'(x^\star)^{t+1}\left(\frac{x_0}{x^\star} - 1\right) \\ x_{t+1} - x^\star &\approx f'(x^\star)^{t+1}(x_0 - x^\star) \\ \boldsymbol{x_{t+1}} &\approx \boldsymbol{f'(x^\star)^{t+1}(x_0 - x^\star) + x^\star} \end{aligned}$$

This is an expression that describes the behaviour of our x terms as we iterate in the vicinity of the steady state. If we want local stability - ie, for the system to converge to $x^\star$ in this vicinity, we require this whole expression to reduce to $x^\star$.

As such, we must set

$$f'(x^\star)^{t+1}(x_0 - x^\star) \approx 0$$

The conditions under which this relationship is true gives us information about our steady states.

The second term just implies that if we begin at our steady state, then we remain at our steady state - this is the trivial case where we have no evolution of the system, but it is a steady state nonetheless. But the good thing is that this case results in a steady state regardless of whether the other term results in a steady state or not.

The other steady state condition is the non-trivial steady state - which happens when

$$f'(x^\star)^{t+1} \approx 0$$

The key observation to notice here is that the term $f'(x^\star)$ will only tend towards 0 under repeated exponentiation if $|f'(x^\star)| < 1$. If $|f'(x^\star)| \geq 1$, then we can see that we will have no convergence under repeated exponentiation - we will either have oscillation from term to term if $f'(x^\star) \leq -1$, or we will have a direct divergence to infinity if $f'(x^\star) \geq 1$.

The good part is that these findings confirm the exploration of the parameters that we performed in part b), where we had $f(x_t) = mx_t + c$.

In that situation, $f'(x_t) = m = f'(x^\star)$, and we were able to confirm that we had guaranteed non-trivial convergence from all points in the domain when $|m| < 1$, and divergence for other values of m, except for the trivial case when $x_0 = x^\star$.

As such, we can write for this non-linear dynamical system that the convergence conditions are as follows:

- Trivial convergence when $x_0 = x^\star$
- Non-trivial convergence when $|f'(x^\star)| < 1$