Assignment 2

Miguel A. Gomez B.

August 26, 2020

(50 points) Many problems in science and mathematics involve iterations. In dynamics, the process that is repeated is the application of a function. To iterate a function means to evaluate the function over and over, using the output of the previous application as the input for the next. Matematically, this is the process of repeatedly composing the function with itself. Given $x_0 \in \mathbb{R}$ we define the orbit of x_0 under F to be the sequence of points $x_0, x_1 = F(x_0), \ldots, x_n = F^n(x_0)$[1]. The point x_0 is called the *seed* of the orbit.

With this in mind, let us play with the following function:

$$F(x) = \begin{cases} 2x & 0 \leq x < \frac{1}{2} \\ 2x - 1 & \frac{1}{2} \leq x < 1 \end{cases}$$

The code of this point can be looked at *RunMeInPython.py*.

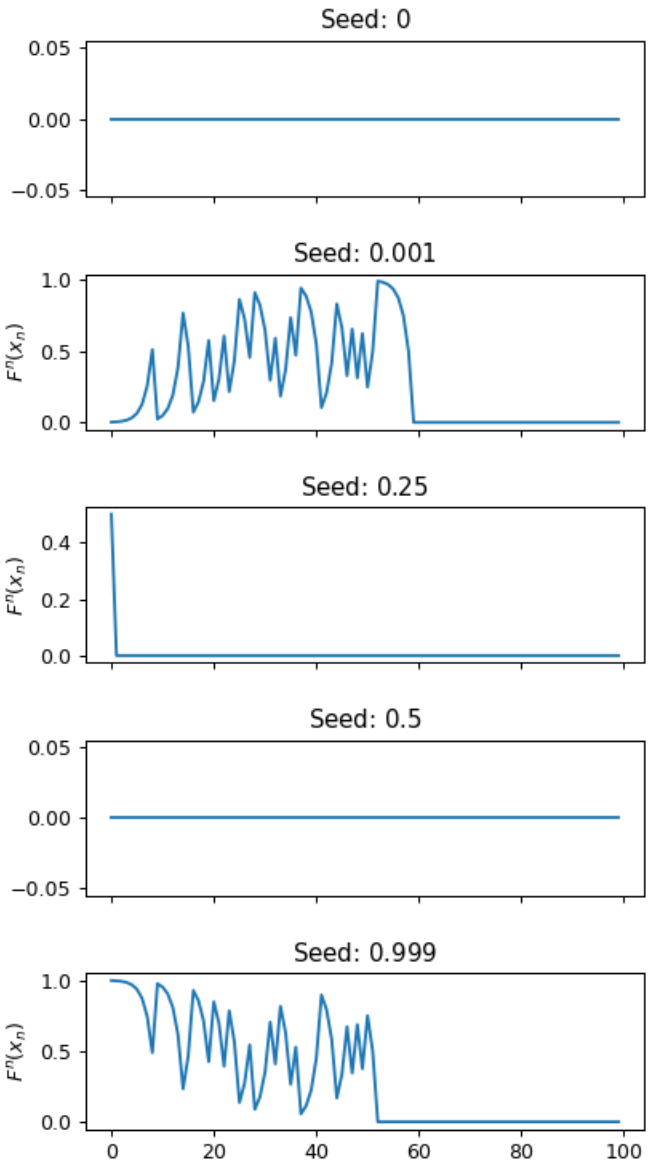
1 Choose 5 different initial seeds that should be in the interval $[1, 0)$ and for each one compute the first 100 points on the corresponding orbit. Record the results by listing the initial seed together with what happened to the orbit.

step	seed 1	seed 2	seed 3	seed 4	seed 5
0	0.000000	0.001000	0.250000	0.500000	0.999000
1	0.000000	0.002000	0.500000	0.000000	0.998000
2	0.000000	0.004000	0.000000	0.000000	0.996000
3	0.000000	0.008000	0.000000	0.000000	0.992000
4	0.000000	0.016000	0.000000	0.000000	0.984000
5	0.000000	0.032000	0.000000	0.000000	0.968000
6	0.000000	0.064000	0.000000	0.000000	0.936000
7	0.000000	0.128000	0.000000	0.000000	0.872000
8	0.000000	0.256000	0.000000	0.000000	0.744000
9	0.000000	0.512000	0.000000	0.000000	0.488000
10	0.000000	0.024000	0.000000	0.000000	0.976000
11	0.000000	0.048000	0.000000	0.000000	0.952000
12	0.000000	0.096000	0.000000	0.000000	0.904000
13	0.000000	0.192000	0.000000	0.000000	0.808000
14	0.000000	0.384000	0.000000	0.000000	0.616000
15	0.000000	0.768000	0.000000	0.000000	0.232000
16	0.000000	0.536000	0.000000	0.000000	0.464000
17	0.000000	0.072000	0.000000	0.000000	0.928000
18	0.000000	0.144000	0.000000	0.000000	0.856000
19	0.000000	0.288000	0.000000	0.000000	0.712000
20	0.000000	0.576000	0.000000	0.000000	0.424000
21	0.000000	0.152000	0.000000	0.000000	0.848000
22	0.000000	0.304000	0.000000	0.000000	0.696000
23	0.000000	0.608000	0.000000	0.000000	0.392000
24	0.000000	0.216000	0.000000	0.000000	0.784000
25	0.000000	0.432000	0.000000	0.000000	0.568000
...	...	...	...	...	...
100	0.000000	0.000000	0.000000	0.000000	0.000000

the complete results can be found on the file *results*.

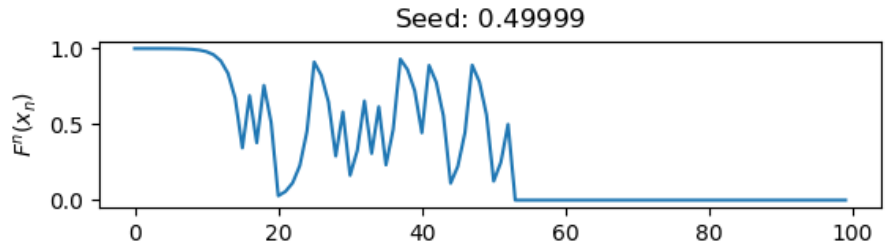
[1]Note that $F^n(x)$ does not mean raise $F(x)$ to the nth power (an operation we will never use). It is the nth iterate of F evaluated at x.

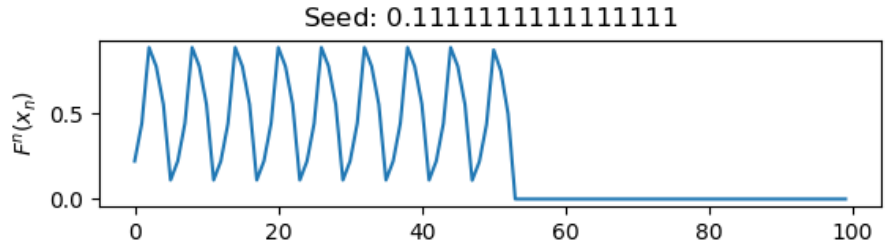
(a) *(20 points)* Plot the results of the two cases in a plane x_i vs x_{i+1}.



(b) *(10 points)*Do they have a visible pattern? Do all (or almost all) orbits behave in the same way?.

As it was expected with the exact values of 0 and 0.5 nothing interesting happens, the orbit moves from the seed to zero, but, with values closer to 0 and 1 (which are limits of the intervals on which the function is defined), the orbit shifts in a visible pattern from increase to decrease to finally go to 0. But actually different patterns arise with different values on the seed.





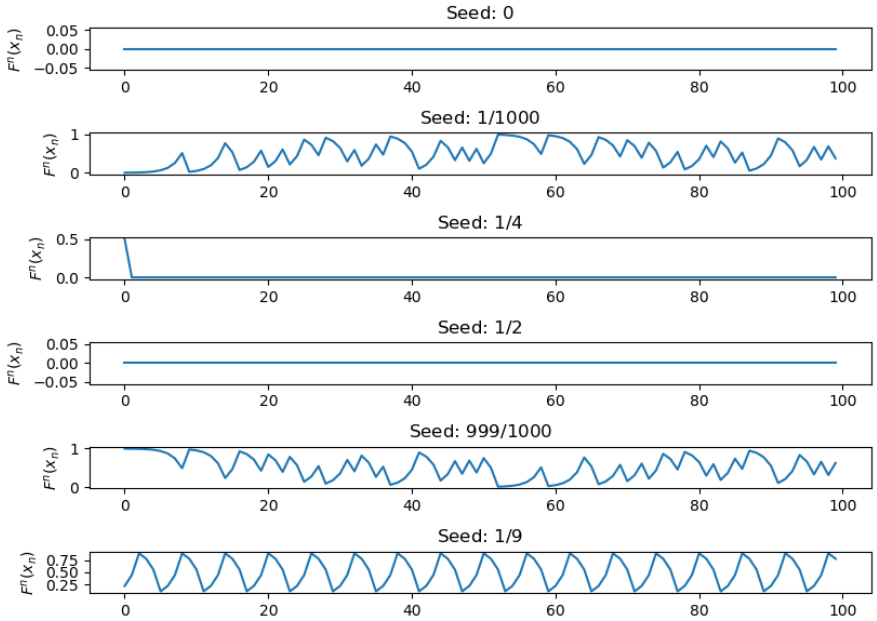
So for different seed values we have different behavior on the orbits.

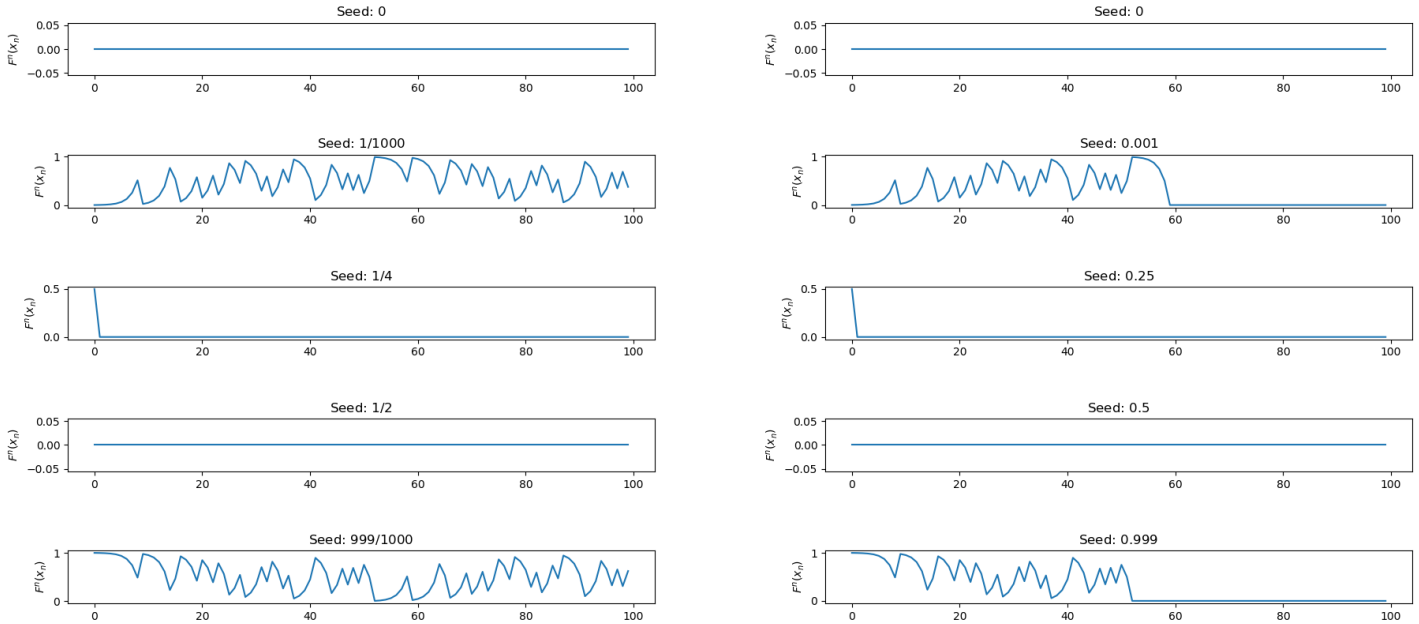
(c) *(20 points)* For the same two discussed cases, instead of treating the seeds a floating point numbers, use *SymPy* to do the exact rational arithmetic. Do the same for the seed $x_0 = \frac{1}{9}$. What do you find for these three cases? Does the computer lie?

step	seed1	seed2	seed3	seed4	seed5	seed6
0	0.000000	0.001000	0.250000	0.500000	0.999000	0.111111
1	0.000000	0.002000	0.500000	0.000000	0.998000	0.222222
2	0.000000	0.004000	0.000000	0.000000	0.996000	0.444444
3	0.000000	0.008000	0.000000	0.000000	0.992000	0.888889
4	0.000000	0.016000	0.000000	0.000000	0.984000	0.777778
5	0.000000	0.032000	0.000000	0.000000	0.968000	0.555556
6	0.000000	0.064000	0.000000	0.000000	0.936000	0.111111
7	0.000000	0.128000	0.000000	0.000000	0.872000	0.222222
8	0.000000	0.256000	0.000000	0.000000	0.744000	0.444444
9	0.000000	0.512000	0.000000	0.000000	0.488000	0.888889
10	0.000000	0.024000	0.000000	0.000000	0.976000	0.777778
11	0.000000	0.048000	0.000000	0.000000	0.952000	0.555556
12	0.000000	0.096000	0.000000	0.000000	0.904000	0.111111
13	0.000000	0.192000	0.000000	0.000000	0.808000	0.222222
14	0.000000	0.384000	0.000000	0.000000	0.616000	0.444444
15	0.000000	0.768000	0.000000	0.000000	0.232000	0.888889
16	0.000000	0.536000	0.000000	0.000000	0.464000	0.777778
17	0.000000	0.072000	0.000000	0.000000	0.928000	0.555556
18	0.000000	0.144000	0.000000	0.000000	0.856000	0.111111
19	0.000000	0.288000	0.000000	0.000000	0.712000	0.222222
20	0.000000	0.576000	0.000000	0.000000	0.424000	0.444444
21	0.000000	0.152000	0.000000	0.000000	0.848000	0.888889
22	0.000000	0.304000	0.000000	0.000000	0.696000	0.777778
23	0.000000	0.608000	0.000000	0.000000	0.392000	0.555556
24	0.000000	0.216000	0.000000	0.000000	0.784000	0.111111
25	0.000000	0.432000	0.000000	0.000000	0.568000	0.222222
26	0.000000	0.864000	0.000000	0.000000	0.136000	0.444444
27	0.000000	0.728000	0.000000	0.000000	0.272000	0.888889
28	0.000000	0.456000	0.000000	0.000000	0.544000	0.777778
29	0.000000	0.912000	0.000000	0.000000	0.088000	0.555556
30	0.000000	0.824000	0.000000	0.000000	0.176000	0.111111
31	0.000000	0.648000	0.000000	0.000000	0.352000	0.222222
32	0.000000	0.296000	0.000000	0.000000	0.704000	0.444444
33	0.000000	0.592000	0.000000	0.000000	0.408000	0.888889
34	0.000000	0.184000	0.000000	0.000000	0.816000	0.777778
35	0.000000	0.368000	0.000000	0.000000	0.632000	0.555556
36	0.000000	0.736000	0.000000	0.000000	0.264000	0.111111
37	0.000000	0.472000	0.000000	0.000000	0.528000	0.222222
38	0.000000	0.944000	0.000000	0.000000	0.056000	0.444444
39	0.000000	0.888000	0.000000	0.000000	0.112000	0.888889
40	0.000000	0.776000	0.000000	0.000000	0.224000	0.777778
41	0.000000	0.552000	0.000000	0.000000	0.448000	0.555556
42	0.000000	0.104000	0.000000	0.000000	0.896000	0.111111
43	0.000000	0.208000	0.000000	0.000000	0.792000	0.222222
44	0.000000	0.416000	0.000000	0.000000	0.584000	0.444444
45	0.000000	0.832000	0.000000	0.000000	0.168000	0.888889
46	0.000000	0.664000	0.000000	0.000000	0.336000	0.777778
...	...	...	...	...	...	...
100	0.000000	0.376000	0.000000	0.000000	0.624000	0.777778

The complete results can be found on the file *weird_results*, the code file used to generate this values is *orbits_sympy.py*.

The data shows a diferent behavior, also notice that *seed6* corresponds to the seed $\frac{1}{9}$. lets look at the graph and analyze the results.

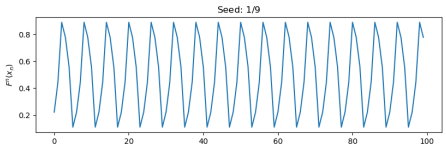


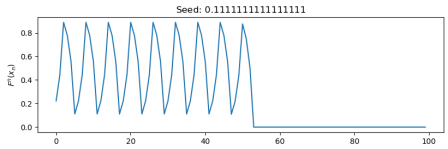


With the seeds 0 and 0.5, there is no difference but on the seeds $\frac{1}{1000} = 0.001$ and $\frac{999}{1000} = 0.999$, visually we can see that there is no difference up to some point, The data shows an slightly difference at some point of the computation:

step	Seed2 (numeric)	Seed5 (numeric)	Seed2 (symbolic)	Seed5 (symbolic)	Difference Seed2	Difference Seed5
0	0.001	0.999	0.001	0.999	0	0
1	0.002	0.998	0.002	0.998	0	0
...	...	...	...	...	...	...
39	0.888	0.112	0.888	0.112	0	0
40	0.776	0.224	0.776	0.223999	0	-9.999999999E-07
41	0.552	0.448	0.552	0.447998	0	-1.9999999999E-06
42	0.104	0.896	0.104	0.895996	0	-3.99999999989298E-06
...	...	...	...	...	...	...
74	0.784	0.216	0	0	-0.784	-0.216
...	...	...	...	...	...	...
100	0.376	0.624	0	0	-0.376	-0.624

The symbolic solution it is a better approximation, the same happens with the seed $\frac{1}{9}$:





step	seed (numeric)	seed (symbolic)	difference
0	0.111111	0.111111	0
1	0.222222	0.222222	0
34	0.777778	0.777778	0
35	0.555555	0.555556	9.99999999917733E-07
36	0.111111	0.111111	0
37	0.222221	0.222222	1.000000000001E-06
38	0.444443	0.444444	1.00000000002876E-06
39	0.888885	0.888889	4.000000000004E-06
40	0.777771	0.777778	6.99999999997925E-06
41	0.555542	0.555556	1.39999999999585E-05
42	0.111084	0.111111	2.70000000000131E-05
43	0.222168	0.222222	5.40000000000263E-05
44	0.444336	0.444444	0.000108
45	0.888672	0.888889	0.000217
46	0.777344	0.777778	0.000434
47	0.554688	0.555556	0.000868
48	0.109375	0.111111	0.001736
49	0.21875	0.222222	0.003472
50	0.4375	0.444444	0.006944
51	0.875	0.888889	0.013889
52	0.75	0.777778	0.027778
53	0.5	0.555556	0.055556
54	0	0.111111	0.111111
100	0	0.777778	0.777778

We can now explain why this happens, and it is related to the approximation that the numeric solution does, notice that $\frac{1}{9} \approx 0.11111...$ but the computer has limited resources, so by convention some approximation to this number is performed when the numeric solution is executed. By contrast, the symbolic computation does not seem to use the same logic, we can conclude that the behavior is in reality given by the way the computer deals with the computation, the technical name for this approximation is related to the value of the EPS constant or machine epsilon, this constant determines the upper bound on the relative error due to a float point number arithmetic. On my machine this value corresponds to $2.220446049250313e-16$ for the float data type, at some point the operation in the program exceeded this constant propagating rounding errors that got bigger at later iterations.