Chapter 11

Empirical Optimization with Divergent Fixed Point Algorithm

When All Else Fails

Why would anyone be interested in an algorithm that never converges to the solution you are looking for? This version of the fixed-point iteration, when approaching a zero or an optimum, emits a strong signal and allows you to detect a small interval likely to contain the solution: the zero or global optimum in question. It may approach the optimum quite well, but subsequent iterations do not lead to convergence: the algorithm eventually moves away from the optimum, or oscillates around the optimum without ever reaching it.

The fixed-point iteration [Wiki] is the mother of all optimization and root-finding algorithms. In particular, all gradient-based optimization techniques [Wiki] are a particular version of this generic method. In this chapter, I use it in a very challenging setting. The target function may not be differentiable or may have a very large number of local minima and maxima. All the standard techniques fail to detect the global optima. In this case, even the fixed-point method diverges. However, somehow, it can tell you the location of a global optimum with a rather decent precision. Once an approximation is obtained, the method can be applied again, this time focusing around a narrow interval containing the solution to achieve higher precision. Also, this method is a lot faster than brute force such as grid search.

I first illustrate the method on a specific problem. Then, generating synthetic data that emulates and generalizes the setting of the initial problem, I illustrate how the method performs on different functions or data sets. The purpose is to show how synthetic data can be used to test and benchmark algorithms, or to understand when they work, and when they don't. This, combined with the intuitive aspects my fixed-point iteration, illustrates a particular facet of explainable AI. Finally, I use a smoothing technique to visualize the highly chaotic functions involved here. It highlights the features of the functions that we are interested in, while removing the massive noise that makes these functions almost impossible to visualize in any meaningful way.

11.1 Introduction

While the technique discussed here is a last resort solution when all else fails, it is actually more powerful than it seems at first glance. First, it also works in standard cases with "nice" functions. However, there are better methods when the function behaves nicely, taking advantage of the differentiability of the function in question, such as the Newton algorithm [Wiki] (itself a fixed-point iteration). It can be generalized to higher dimensions, though I focus on univariate functions here.

Perhaps the attractive features are the fact that it is simple and intuitive, and quickly leads to a solution despite the absence of convergence. However, it is an empirical method and may require working with different parameter sets to actually find a solution. Still, it can be turned into a black-box solution by automatically testing different parameter configurations. In that respect, I compare it to the empirical elbow rule to detect the number of clusters in unsupervised clustering problems. I discuss the elbow rule and its automation in section 16.6. Another fixed-point algorithm leading to explainable AI in the context of linear regression, is discussed in section 7.1.

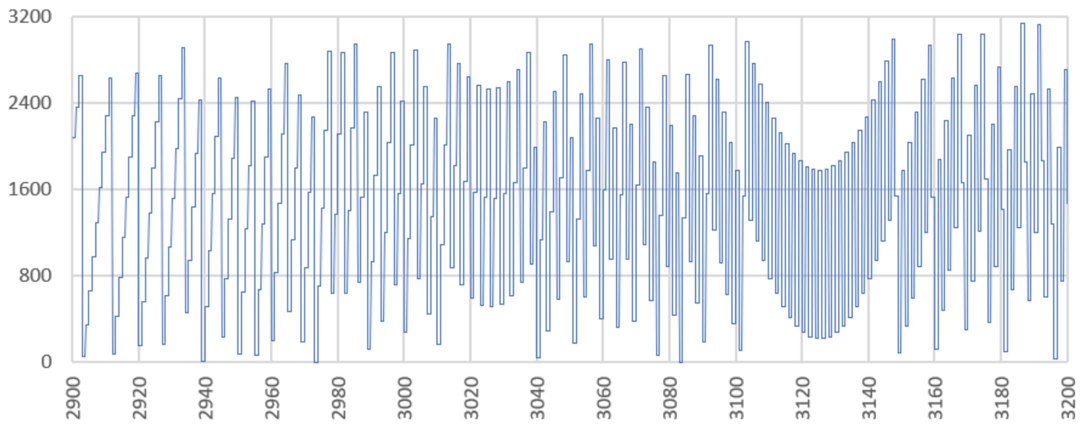
Figure 11.1: Function $f(b)$ as a better alternative to $g(b)$ in Figure 11.2. Root at $b = 3083$.

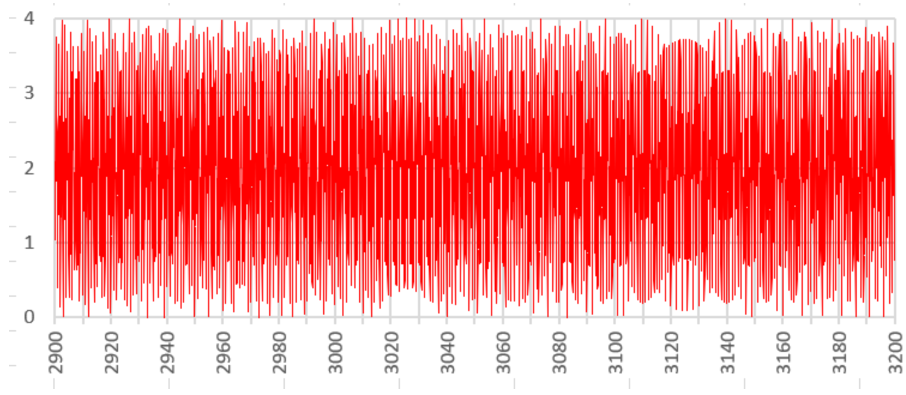
Figure 11.2: Function $g(b) = 2 - \cos(2\pi b) - \cos(2\pi a/b)$, with $a = 3083 \times 7919$.

11.1.1 The problem, with illustration

The method can solve two types of problems: finding the zeros of a function, or its optima (maxima or minima). Optima correspond to a zero of the derivative, so finding them is a root-finding problem. In my example, the functions typically have multiple global minima that we want to detect. The initial problem is to find a factor b of a large integer number a. Here a is fixed, and the function is denoted as $f(b)$ with $f(b) = 0$ if and only if b is such a factor (integer number) different from 1 and a. If b is not a factor, then $f(b) > 0$. Initially, the interest was to factor a number that is a product of two large primes, as this has implications in cryptography. However, the technique led to a much larger class of applications where it has much more value than for factoring integers.

The first step was to change the problem setting: extending the function $f(b)$ which accepts an integer argument b, into a continuous function where b is a real number. Here $f(b) = a \bmod b$. So to get a continuous extension, one has to define the modulo operator for arguments b that are not integer numbers. Finally, it led to

$$f(b) = a - \lfloor b + \epsilon \rfloor \left\lfloor \epsilon + \frac{a}{\lfloor b + \epsilon \rfloor} \right\rfloor, \quad b > 0. \tag{11.1}$$

The brackets represent the integer part function, and $\epsilon = 0$. However, in the code, $\epsilon = 10^{-8}$ to avoid problems caused by numerical precision. Formula (11.1) defines the base function, denoted as `fmod` in the Python code. It is a piecewise constant function, pictured in Figure 11.1 between $b = 2900$ and $b = 3200$, using $a = 3083 \times 7919$. Thus the interval $[2900, 3200]$ contains the root $b = 3083$. There are no other roots besides 3083 and 7919 since these two integers are prime numbers. Also note that $0 \leq f(b) < b$. Optimization methods based on the gradient are guaranteed not to work here.

Note that instead of $f(b)$, I could have used $g(b) = 2 - \cos(2\pi b) - \cos(2\pi a/b)$. This function is also positive and equal to zero only if b is an integer number that divides the integer a. In addition, g is bounded with $0 \leq g(b) \leq 4$, and infinitely differentiable. Yet it has a very large number of local minima and maxima: the frequency of oscillations is insanely high. For all purposes, g is just as chaotic of f, indeed worse than f, and no standard optimization algorithm could handle it. The function g is pictured in Figure 11.2.

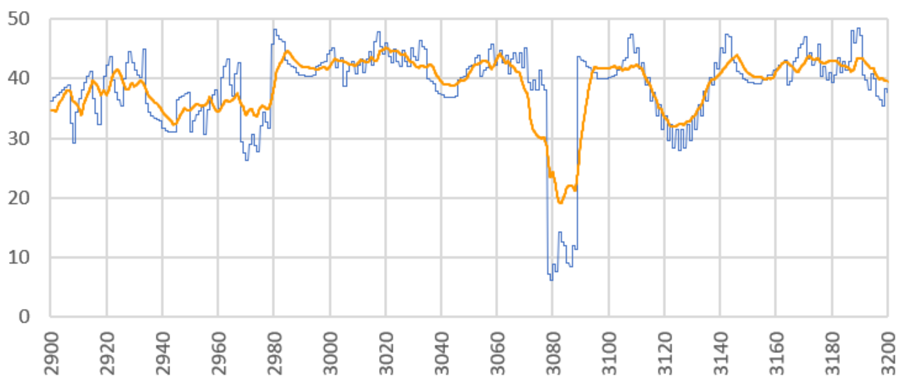

Figure 11.3: Transformed function f_3, amplifying the root at $b = 3083$.

11.2 Non-converging fixed-point algorithm

In this section, I start with the function $f(b)$ defined by Formula (11.1). I investigate a larger class of functions in section 11.3. The trick is to massively amplify the zero, creating ripple effects that the fixed point iteration can leverage to narrow down on the solution. In the end, instead of looking at convergence (absent here), you look at the strength of a signal ρ_n at iterations $n = 1, 2$ and so on. The value of ρ_n is typically close to 1, but high values (above 2) indicate that something unusual is happening. Typically such high values are created when stepping over a root.

11.2.1 Trick leading to intuitive solution

Let us assume that f is positive and that its minimum value is zero. If f takes on negative values, replace $f(b)$ by the absolute value $|f(b)|$ or by the square $f^2(b)$. This works both in root-finding and optimization problems. In optimization, f is the derivative of the target function. The base function f (or its absolute value or square) is denoted as f_0.

The goal is to apply successive transformations to make the function f_0 more amenable to root detection. The first transformation consists of taking the logarithm, thus creating a vertical bottomless abyss in the graph of the function, around any root. In practice, it is implemented as follows: the new function is simply

$$f_1(b) = \max\Big[\log(f_0(b)), \delta\Big],$$

where δ is a negative number, large enough in absolute value. In the Python code, δ is represented by `logeps`, and set to -10. This parameter controls the depth of the abyss, that now has a bottom. It helps discriminate between a value very close to zero, and a value exactly equal to zero. For instance, if $f_0(b) = 0.01$, then $f_1(b) = -4.61$, while if $f_0(b) = 0$, then $f_1(b) = -10$.

The second step consists in enlarging the abyss. Its walls will change from vertical to inclined. The width of the abyss, and the slope of its walls, is controlled by the parameter `window` in the Python code, and referred to here as w. In a nutshell, this transformation is a moving average applied to f_1. The resulting function is

$$f_2(b) = \frac{1}{2w+1}\sum_{k=-w}^{w} f_1(b + kh).$$

The increment h is set to 1 as it makes sense in my particular problem. I did not test other values.

The third step is a linear transformation, turning f_2 into f_3, with $f_3(b) = p + qf_2(b)$. The parameters p and q are respectively denoted as `offset` and `slope` in the Python code. The functions f_0, f_1 and f_3 are respectively denoted as `fmod`, `fmod2` and `fresidue` in the Python code. The blue curve in Figure 11.3 is the f_3 transform associated to the f_0 function pictured in Figure 11.1, with $p = -100$ and $q = 20$. Subsequent transformations discussed in section 11.4 are done only for embellishment, and are irrelevant to the fixed point algorithm.

11.2.2 Root detection: method and parameters

The fixed point algorithm starts with an initial value b_0, and then proceeds iteratively as follows:

$$b_{n+1} = b_n + \mu f(b_n). \tag{11.2}$$

If the sequence (b_n) converges, the function f is continuous and $\mu \neq 0$, then b_n must converge to some b^* such that $f(b^*) = 0$, thus b_n converges to a root of f. You can allow μ to depend on n, and in one example I successfully used $\mu = 1/\sqrt{b_n}$. The function f used here is actually the function f_3 defined in section 11.2.1 and pictured in blue in Figure 11.3. This function is not even continuous, and the fixed point iteration does not converge. Other functions are investigated in section 11.3. Here, the purpose is to explain how the fixed point iteration can help find a root despite the lack of convergence.

Many of the parameters in the Python code are described in section 11.2.1, including `offset`, `slope` and `logeps`. The parameter `eps` corresponds to ϵ in section 11.1.1. The main parameters driving the fixed-point iteration are:

- `mu`: corresponding to μ in Formula (11.2). A large value results in bigger jumps in the fixed point iteration, allowing you to find a root faster, but with an increased risk of missing all roots when μ becomes too large.
- `window`: corresponding to w in section 11.2.1. A large w increases your chances of finding a root. The price to pay is reduced precision. If b^* is a root and the fixed point iteration succeeds in locating it, it will not find b^*, but instead, it will tell you that there might be a root in the interval $[b^* - w, b^* + w]$, without knowing what the actual b^* is. Thus a large w is useful to get a rough approximation of where a root is located.

I illustrate these features, as well as how fast this algorithm is compared to brute force, on a real example in section 11.2.3. The algorithm does not converge with the type of functions discussed here: the successive iterates b_n become larger and larger as n increases, or they may oscillate without ever converging. Of course if the function is smooth enough and with the right parameters, it will converge. But we are not interested in that case.

Since there is no convergence in my examples, how can the algorithm detect a root? Now I explain how it works. First, define $\Delta_n = b_n - b_{n-1}$. Then let $\rho_n = \Delta_n/\Delta_{n-1}$. The number ρ_n is called the **signal** at iteration n. Usually, $\rho_n \approx 1$. If ρ_n is unusually low or high, we say that the signal is strong. In my examples, a value $\rho_n > 2$ usually means that there is a root close to b_{n-1}. Details are discussed in section 11.2.3.

Finally, the functions investigated here have many values close to zero. The brute force method consisting of testing a very large number of values may not even work. In most cases, finding b such that $f_0(b) = 0.001$ does not mean that there is a b^* close to b such that $f_0(b^*) = 0$. For the same reason, the efficient but naive bisection method [Wiki] will also fail. Note that my method is empirical: a strong signal does not always correspond to a root, and the absence of strong signal does not mean that there is no root. Testing with different parameter sets helps.

11.2.3 Case study: factoring a product of two large primes

The goal here is to find at least one of the two roots of the function $f_0(b)$ pictured in Figure 11.1, and defined by (11.1). Using the transformations described in section 11.2.1, I will actually work with the third transform $f_3(b)$ in the fixed point iteration [Formula (11.2)], starting with $b_0 = 2000$. The two roots are the two prime factors of $a = 7919 \times 3083$, that is $b^* = 3083$ and $b^* = 7919$. So finding one root makes it straightforward to find the other one.

Finding the roots of f_0 is the same as finding the roots of $g(b) = 2 - \cos(2\pi b) - \cos(2\pi a/b)$ pictured in Figure 11.2. The parameters values are those in the Python code in section 11.3.3. The results are displayed in Table 11.1 and Figure 11.4. The root $b^* = 3083$ is detected at iteration $n = 31$. Since $\rho_{31} = 4.72$ is exceptionally high, there is a strong possibility that the algorithm stepped over a root in the previous iteration ($n = 30$). Indeed, $b_{30} = 3086.18$ is close to the root $b^* = 3083$. What the algorithm tells is that there is probably a root in the interval $[b_{30} - w, b_{30} + w]$ where w is the size of the window (the parameter `window` in the Python code) set to 5 here.

At this point, after testing $b = 3081, 3082$ and 3083 to find the root, one can stop the algorithm. It does not converge anyway: b_n is getting bigger and bigger, faster and faster, as shown in Table 11.1. The second root is simply $b^* = 7919 = a/3083$. Surprisingly though, at iteration $n = 127$, there is another spike: $\rho_{127} = 2.00$. Again, $b_{126} = 7915.67$ is close to the second root $b^* = 7919$. There is another, weaker spike at $n = 71$, with $\rho_{71} = 1.88$. This one is a false positive.

11.3 Generalization with synthetic random functions

Perhaps the synthetic example most closely resembling $f_0(b) = a \bmod b$ discussed in section 11.2.2 is $f_0(b) = \lfloor bU_b \rfloor$ where the U_b's are independent random deviates on $[0, 1]$. The major difference is the independence

n	b_n	Δ_n	ρ_n	n	b_n	Δ_n	ρ_n	n	b_n	Δ_n	ρ_n
1	2033.70	33.70	–	46	3740.11	48.99	0.79	91	5940.08	58.20	0.89
2	2070.73	37.02	0.91	47	3787.29	47.18	1.04	92	5996.05	55.97	1.04
3	2105.69	34.96	1.06	48	3833.88	46.58	1.01	93	6044.88	48.83	1.15
4	2143.15	37.47	0.93	49	3880.82	46.95	0.99	94	6094.87	50.00	0.98
5	2177.58	34.43	1.09	50	3929.45	48.62	0.97	95	6158.31	63.43	0.79
6	2211.57	33.98	1.01	51	3974.72	45.27	1.07	96	6200.89	42.58	1.49
7	2234.25	22.68	1.50	52	4021.53	46.81	0.97	97	6257.29	56.40	0.75
8	2263.00	28.75	0.79	53	4066.34	44.81	1.04	98	6308.72	51.43	1.10
9	2300.72	37.72	0.76	54	4109.76	43.42	1.03	99	6368.62	59.90	0.86
10	2332.77	32.05	1.18	55	4146.40	36.64	1.18	100	6426.70	58.08	1.03
11	2372.61	39.84	0.80	56	4196.55	50.15	0.73	101	6482.41	55.70	1.04
12	2396.52	23.91	1.67	57	4240.43	43.88	1.14	102	6538.13	55.72	1.00
13	2435.08	38.56	0.62	58	4283.13	42.70	1.03	103	6593.53	55.40	1.01
14	2469.07	33.99	1.13	59	4339.39	56.26	0.76	104	6651.41	57.89	0.96
15	2503.82	34.75	0.98	60	4379.64	40.25	1.40	105	6705.05	53.64	1.08
16	2536.48	32.65	1.06	61	4427.22	47.58	0.85	106	6760.28	55.23	0.97
17	2572.00	35.53	0.92	62	4475.73	48.51	0.98	107	6816.19	55.91	0.99
18	2609.43	37.42	0.95	63	4525.94	50.22	0.97	108	6873.41	57.22	0.98
19	2650.28	40.86	0.92	64	4575.57	49.63	1.01	109	6931.39	57.98	0.99
20	2692.95	42.67	0.96	65	4627.85	52.28	0.95	110	6980.78	49.39	1.17
21	2731.07	38.12	1.12	66	4674.64	46.79	1.12	111	7039.95	59.18	0.83
22	2771.50	40.43	0.94	67	4721.67	47.04	0.99	112	7108.67	68.72	0.86
23	2802.76	31.26	1.29	68	4771.85	50.17	0.94	113	7168.63	59.96	1.15
24	2844.27	41.51	0.75	69	4813.86	42.01	1.19	114	7220.86	52.23	1.15
25	2889.18	44.90	0.92	70	4871.29	57.43	0.73	115	7274.43	53.57	0.98
26	2924.68	35.50	1.26	71	4901.82	30.53	1.88	116	7336.46	62.02	0.86
27	2961.03	36.35	0.98	72	4957.42	55.60	0.55	117	7390.45	53.99	1.15
28	3001.25	40.22	0.90	73	4996.69	39.27	1.42	118	7449.19	58.74	0.92
29	3046.38	45.13	0.89	74	5061.62	64.93	0.60	119	7510.73	61.55	0.95
30	3086.18	39.80	1.13	75	5101.22	39.59	1.64	120	7565.90	55.17	1.12
31	3094.62	8.44	4.72	76	5149.52	48.30	0.82	121	7628.20	62.30	0.89
32	3135.72	41.10	0.21	77	5206.31	56.79	0.85	122	7688.69	60.49	1.03
33	3172.01	36.29	1.13	78	5255.88	49.57	1.15	123	7750.83	62.15	0.97
34	3216.43	44.42	0.82	79	5306.17	50.30	0.99	124	7802.41	51.58	1.20
35	3259.84	43.41	1.02	80	5365.49	59.31	0.85	125	7855.95	53.54	0.96
36	3303.38	43.54	1.00	81	5416.54	51.06	1.16	126	7915.67	59.72	0.90
37	3343.08	39.70	1.10	82	5472.24	55.69	0.92	127	7945.56	29.89	2.00
38	3384.42	41.34	0.96	83	5524.53	52.30	1.06	128	8008.42	62.85	0.48
39	3422.90	38.48	1.07	84	5577.40	52.87	0.99	129	8072.94	64.52	0.97
40	3468.43	45.53	0.85	85	5629.32	51.92	1.02	130	8131.86	58.93	1.09
41	3510.77	42.34	1.08	86	5672.84	43.52	1.19	131	8194.84	62.98	0.94
42	3565.04	54.26	0.78	87	5724.36	51.52	0.84	132	8258.05	63.21	1.00
43	3603.60	38.57	1.41	88	5777.10	52.74	0.98	133	8312.50	54.45	1.16
44	3652.45	48.85	0.79	89	5829.89	52.79	1.00				
45	3691.12	38.66	1.26	90	5881.87	51.98	1.02				

Table 11.1: High ρ_n at iterations $n = 31$ and $n = 127$ points to roots 3083 and 7919

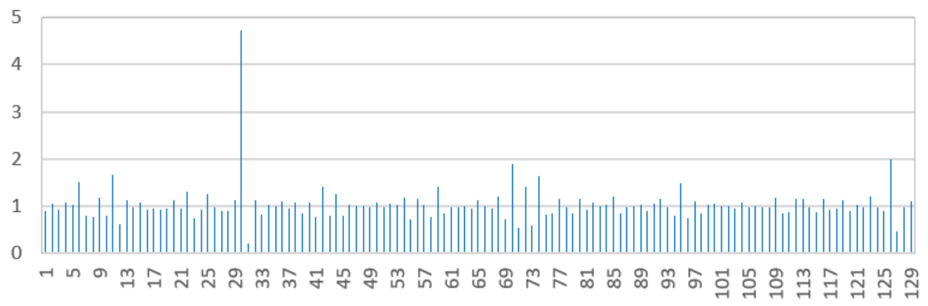
Figure 11.4: Signal strength ρ_n, first 130 fixed-point iterations; $n = 31$ leads to a root.

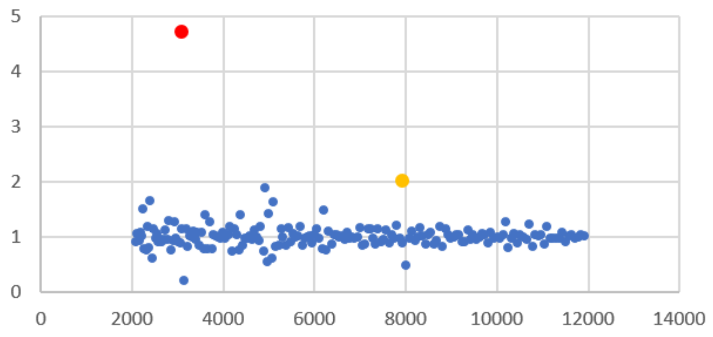
Figure 11.5: (b_n, ρ_n) plot. Yellow and orange dots linked to roots.

assumption. For $a \bmod b$, the residues somewhat behave as if they were independent (assuming the b's are integers and a is fixed) but they are not really independent. To use $f_0(b) = \lfloor bU_b \rfloor$, set `mode='Random'` in the Python code.

11.3.1 Example

Unlike in the example based on a product of two primes, now the random function $f_0(b)$ has infinitely many roots. They are more and more spaced out as b increases. In this example, with the seed of the random generator set to `seed=105`, the first few roots are

5646, 15156, 59004, 122345, 689987, 1021037, 1186047, 2829138.

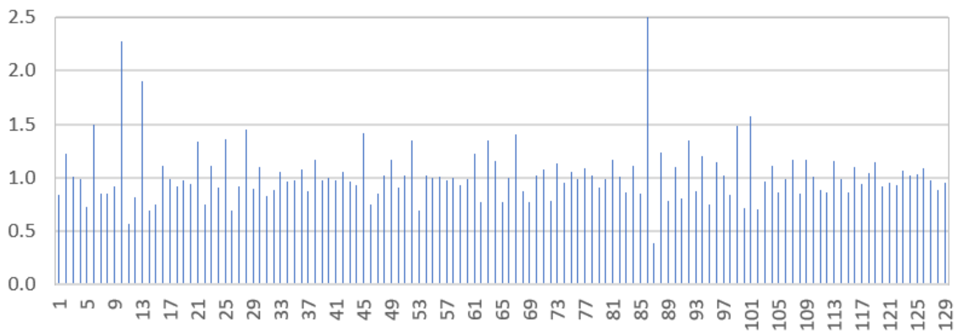
Figure 11.6: Signal strength ρ_n, first 130 fixed-point iterations; $n = 87$ leads to a root.

The statistical distribution of the roots and the number of expected roots in any given interval is studied in section 11.3.2. I used the same python code with same parameters as in section 11.2.3, also starting with $b_0 = 2000$. This time, at iteration $n = 87$, we have $\rho_n = 2.52$ which is extremely high. Again, this points to a root at iteration $n-1$, with $b_{86} = 5650.48$. The actual root is $b^* = 5646$. It could be anywhere between $b_{86} - w$ and $b_{86} + w$, with w set to 5 in this case, via `window=5`. So it suffices to test $b = 5645$ and $b = 5646$ to find the root. The values of ρ_n for successive iterations of the fixed point algorithm, between $n = 1$ and $n = 130$, are displayed in Figure 11.6.

With the current parameters, the algorithm misses the next root $b^* = 15156$. Decreasing μ or increasing w (or a combination of both that minimizes the number of required iterations) may lead to that root, most likely in more than 100 iterations. Note that high values of ρ_n are also found at $n = 11$ and $n = 14$, respectively $\rho_{11} = 2.28$ and $\rho_{14} = 1.90$. These are false positives and can easily be ruled out. It is unusual to reach a root in as few as 20 iterations for this kind of problem, especially when starting far away from the root.

Figure 11.7 features the random function discussed in this section, with a visible absolute minimum around $b = 5646$ both for the blue and orange curves, but invisible (although present) for the initial function f_0 in red. Here $b \in [5500, 5800]$. The blue curve is the transformed version of f_0, referred to as f_3, while the orange curve is the result after additional smoothing discussed in section 11.4. The vertical axis for the blue and orange curves is on the left. For the red curve, it is on the right.

11.3.2 Connection to the Poisson-binomial distribution

Depending on the sequence of random deviates (U_b), the new function f_0 may have zero, one, or more than one integer root in any specific interval $[s, t]$. The expected number of integer roots is a random variable $N(s, t)$ with a Poisson-binomial distribution [Wiki]. Assuming s and t are integers, the expectation, variance, and probability of no root are respectively

$$\mathrm{E}[N(s,t)] = \sum_{k=s}^{t} \frac{1}{k}, \quad \mathrm{Var}[N(s,t)] = \sum_{k=s}^{t} \frac{1}{k}\Big(1 - \frac{1}{k}\Big), \quad P[N(s,t) = 0] = \prod_{k=s}^{t} \Big(1 - \frac{1}{k}\Big).$$

When s, t are very large and $t/s \to \lambda$, the Poisson-binomial distribution is well approximated by a Poisson distribution of expectation $\log \lambda$. A proof of this advanced result (not even taught in graduate classes) can be found in my book on stochastic processes [47], in section 2.3.1. This result generalizes the well-known convergence of Binomial to Poisson distribution under some assumptions. It is a particular case of Le Cam's inequality [Wiki]; see also [95].

Applied to the problem in section 11.2.2 with $f_0(b) = a \bmod b$, with $b \in [2900, 3200]$, the chance to find at least one root in that interval is approximately $1 - \exp(-\log \lambda) = 1 - 1/\lambda$. Since $\lambda = t/s$ with $s = 2900, t = 3200$, the chance in question is about 9.4%. Note that $t < \sqrt{a}$. In general this condition is required for the result to be valid. The absence of root in the interval $[2, \sqrt{a}]$ would mean that a is a prime number.

11.3.2.1 Location of next root: guesstimate

Let T_s be the location of the first root larger than s. I am interested in the ratio $R_s = T_s/s > 1$. From the previous discussion, the distribution of successive roots approximately follows a non-homegeneous Poisson process when s is large. It is easy to prove that R_s has an infinite expectation. However, $\log R_s$ has a finite expectation. Let's compute it, using the Poisson approximation. We have

$$P(\log R_s > \tau) = P[T_s > s \exp(\tau)] = \exp(-\tau).$$

Thus,

$$\mathrm{E}[\log R_s] = \int_0^\infty \exp(-\tau) d\tau = 1.$$

So, given a root $b^* > 2000$, one would expect the next one to be of the order $e \cdot b^*$, with $e = 2.718\ldots$. This is consistent with the successive roots displayed at the beginning in section 11.3.1.

11.3.2.2 Integer sequences with high density of primes

The Poisson-binomial distribution, including its Poisson approximation, can also be used in this context. The purpose is to find fast-growing integer sequences with a very high density of primes, see here.

The probability for a large integer a to be prime is about $1/\log a$, a consequence of the prime number theorem [Wiki]. Let $a_1, a_2, \ldots$ be a strictly increasing sequence of positive integers, and N denote the number of primes among $a_n, a_{n+1}, \ldots, a_{n+m}$ for some large n. Assuming the sequence is independently and congruentially equidistributed, then N has a Poisson-binomial distribution of parameters $p_n, \ldots, p_{n+m}$, with $p_k = 1/\log a_k$. It is unimportant to know the exact definition of congruential equidistribution. Roughly speaking, it means that the joint empirical distribution of residues across the a_k's, is asymptotically undistiguishable from that of a sequence of random integers. Thus a sequence where 60% of the terms are odd integers, do not qualify (that proportion should be 50%).

This result is used to assess whether a given sequence of integers is unusually rich, or poor, in primes. If it contains far more large primes than the expected value $p_n + \cdots + p_{n+m}$, then we are dealing with a very interesting, hard-to-find sequence, useful both in cryptographic applications and for its own sake. One can build confidence intervals for the number of such primes, based on the Poisson-binomial distribution under the assumption of independence and congruential equidistribution. A famous example of such a sequence (rich in prime numbers) is $a_k = k^2 - k + 41$ [Wiki]. If $n, m \to \infty$ and $p_n^2 + \cdots + p_{n+m}^2 \to 0$, then the distribution of N is well approximated by a Poisson distribution, thanks to Le Cam's theorem.

11.3.3 Python code: finding the optimum

This Python code performs the transformations from f_0 to f_3 as described in section 11.2.1 and run the fixed point algorithm on f_3 to find a minimum of f_3, and thus a root of f_0. The parameters are described in section 11.2.2. The case `mode='Prime'` corresponds to the example discussed in section 11.2.3, while the case `mode='Random'` corresponds to the random function studied in section 11.3.1. The results are saved in a tab-separated text file `rmodb.txt`. The full version with curve smoothing, tabulation and saving the values of the various transforms, is in section 11.4.

A potential improvement is to handle the situation when b_n decreases to the point of becoming negative. Also, the values of $f_0(b)$ and $f_3(b)$ are not stored in some array or hash table, but instead computed on the fly. Some may be computed multiple times, and the code is not efficient in that regard. This is particularly true if the parameter `window` is large.

```python
# realmod.py | MLTechniques.com | vincentg@MLTechniques.com
# Find b such that fresidue(b) = 0, via fixed-point iteration
# Here, fresidue(b) = a mob b (a is a fixed integer; b is a real number)

import math
import random

a = 7919*3083      # product of two prime numbers (purpose: find factor b = 3083)
logeps = -10       # approximation to log(0) = - infinity
eps = 0.00000001   # used b/c Python sometimes fails to compute INT(x) correctly
offset = -100      # offset of linear transform, after log transform
slope = 20         # slope of linear transform, after log transform
mu = 1             # large mu --> large steps between successive b in fixed-point
b0  = 2000         # initial b in fixed-point iterration
window = 5         # size of window search
mode  = 'Prime'    # Options: 'Prime' or 'Random'

def fresidue(b):
   # function f_3
   sum=0
   sumw=0
   for w in range(-window,window+1):
      sumw = sumw+1
      sum += fmod2(b+w)
   ry=offset + slope*sum/sumw
   return(ry)

def fmod2(b):
   # function f_1
   ry=fmod(b)
   if ry==0:
      ry=logeps
   else:
      ry=math.log(ry)
   return(ry)

def fmod(b):
   # function f_0
   if mode=='Prime':
      ry=a-int(b+eps)*int(eps+a/int(b+eps))
   elif mode=='Random':
      ry=res[int(b+eps)]
   return(ry)

if mode=='Random':
   # pre-compute f_0(b) for all integers b
   seed = 105
   random.seed(seed)
   res={}
   for b in range(1,40000):
      res[b]=int(b*random.random());
      if res[b]==0 and b >= b0:
         print("zero if b =", b)

# fixed-point iteration
OUT=open("rmodb.txt","w")
b = b0
for n in range(1,190):
   old_b = b
   b = b + mu*fresidue(b)
   delta = b - old_b
   line=str(n)+"\t"+str(b)+"\t"+str(delta)+"\n"
   OUT.write(line)
OUT.close()
```

11.4 Smoothing highly chaotic curves

All the functions discussed so far are piecewise constant. The purpose here is to beautify these functions to make them look smooth and continuous, while preserving the key feature: the roots, corresponding to massive dips in the graph of these functions. This is best illustrated in Figure 11.7, where the original function f_0 in red is impossible to interpret, while the blue curve f_3 clearly features the minimum. The orange curve is the final result obtained after smoothing the blue curve, using the three transformations `fresidue2`, `fresidue3` and `fresidue4` in the Python code in section 11.4.1, in that order.

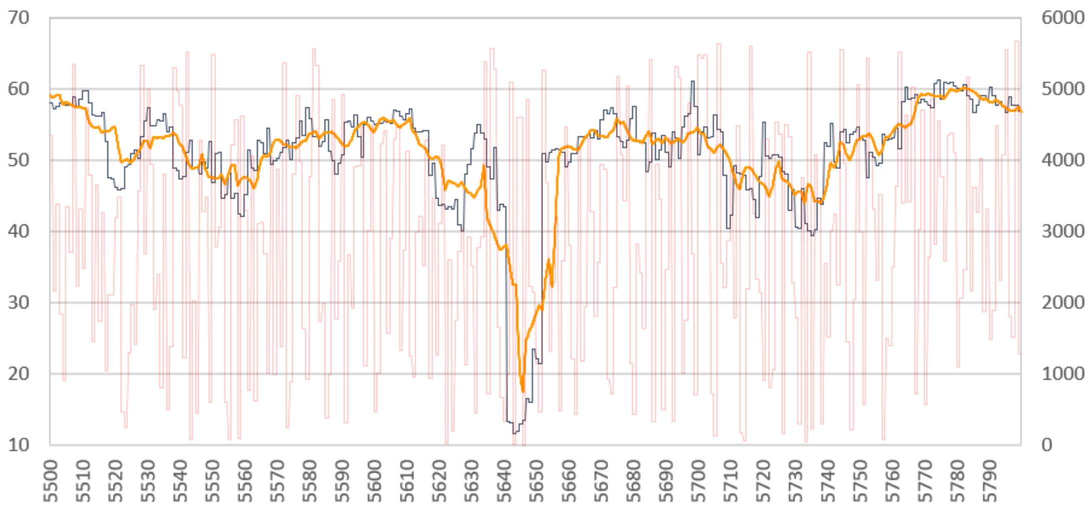

Figure 11.7: Random function from section 11.3.1, with root at $b = 5646$.

11.4.1 Python code: smoothing

The code is also available on my GitHub repository, here. In addition to smoothing, it also produces a tab-separated text file `rmod.txt`. This files contains the tabulated values of the function f_0 and its successive transforms, for b in an range specified by the user.

```
# realmod_full.py | MLTechniques.com | vincentg@MLTechniques.com
# Find b such that fresidue(b) = 0, via fixed-point iteration
# Here, fresidue(b) = a mob b (a is a fixed integer; b is a real number)

import math
import random

a = 7919*3083      # product of two prime numbers (purpose: find factor b = 3083)
logeps = -10       # approximation to log(0) = - infinity
eps = 0.00000001  # used b/c Python sometimes fails to compute INT(x) correctly
```

```
offset = -100     # offset of linear transform, after log transform
slope = 20        # slope of linear transform, after log transform
mu = 1            # large mu --> large steps between successive b in fixed-point
b0  = 2000        # initial b in fixed-point iterration
window = 5        # size of window search
mode  = 'Random'   # Options: 'Prime' or 'Random'

# -- transformation needed for fixed-point iteration
def fresidue(b):
   # function f_3
   sum=0
   sumw=0
   for w in range(-window,window+1):
      sumw = sumw+1
      sum += fmod2(b+w)
   ry=offset + slope*sum/sumw
   return(ry)

def fmod2(b):
   # function f_1
   ry=fmod(b)
   if ry==0:
      ry=logeps
   else:
      ry=math.log(ry)
   return(ry)

def fmod(b):
   # function f_0
   if mode=='Prime':
      ry=a-int(b+eps)*int(eps+a/int(b+eps))
   elif mode=='Random':
      ry=res[int(b+eps)]
   return(ry)

#-- smooth the curve f_3
def fresidue4(b):
   left = fresidue3(b)
   right = fresidue3(b+1)
   weight = b - int(eps+b)
   ry = (1-weight)*left + weight*right
   return(ry)

def fresidue3(b):
   f1 = fresidue2(b-5)
   f2 = fresidue2(b-6)
   f3 = fresidue2(b+4)
   ry = (f1+f2+f3)/3
   return(ry)

def fresidue2(b):
   flag1=0
   flag2=0
   ry = fresidue(b)
   ry2 = ry
   if ry2 > fresidue(b+5):
      ry2 = ry2 - 0.20*(ry2-fresidue(b+5))
      flag1 = 1
   if ry2 > fresidue(b+4):
      ry2 = ry2 - 0.20*(ry2-fresidue(b+4))
      flag1=1
   if ry2 > fresidue(b+3):
      ry2 = ry2 - 0.20*(ry2-fresidue(b+3))
      flag1=1
   if ry2 > fresidue(b+2):
      ry2 = ry2 - 0.50*(ry2-fresidue(b+2))
      flag1=1
   if ry2 > fresidue(b+1):
      ry2 = ry2 - 0.50*(ry2-fresidue(b+1))
      flag1=1
   ry3 = ry;
   if ry3 < fresidue(b+5):
      ry3 = ry3 - 0.30*(ry3-fresidue(b+5))
      flag2 = 1
   if ry3 < fresidue(b+4):
      ry3 = ry3 - 0.30*(ry3-fresidue(b+4))
      flag2 = 1
   if ry3 < fresidue(b+3):
      ry3 = ry3 - 0.30*(ry3-fresidue(b+3))
      flag2 = 1
   if ry3 < fresidue(b+2):
      ry3 = ry3 - 0.30*(ry3-fresidue(b+2))
      flag2 = 1
   if ry3 < fresidue(b+1):
      ry3 = ry3 - 0.50*(ry3-fresidue(b+1))
      flag2 = 1
   if flag1==1 and flag2==0:
      ry = ry2
   if flag1==0 and flag2==1:
      ry = ry3
      if flag1==1 and flag2==1:
         gap2 = abs(ry2-ry)
         gap3 = abs(ry3-ry)
         if gap3 > gap2:
            ry = ry3
         else:
            ry = ry2
   return(ry)

#-- preprocessing if mode=='Random'
if mode=='Random':
   # pre-compute f_0(b) for all integers b
   seed = 105
   random.seed(seed)
   res={}
   for b in range(1,40000):
      res[b]=int(b*random.random());
      if res[b]==0 and b >= b0:
         print("zero if b =", b)

#-- fixed-point iteration
OUT=open("rmodb.txt","w")
b = b0
for n in range(1,390):
   old_b = b
   b = b + mu*fresidue(b)
   delta = b - old_b
   line=str(n)+"\t"+str(b)+"\t"+str(delta)+"\n"
   OUT.write(line)
OUT.close()

#-- save tabulated function f (transforms and smoothed versions)
import numpy as np
OUT=open("rmod.txt","w")
for b in np.arange(5500, 5800, 0.1):
  r0 = fmod(b)
  r1 = fmod2(b)
  r2 = fresidue(b)
  r3 = fresidue2(b)
  r4 = fresidue3(b)
  r5 = fresidue4(b)
  line=str(b)+"\t"+str(r0)+"\t"+str(r1)+"\t"+str(r2)+"\t"+str(r3)+"\t"
  line=line+str(r4)+"\t"+str(r5)+"\n"
  OUT.write(line)
OUT.close()
```

11.5 Connection to synthetic data: random functions

First, there is a strong similarity between the various transforms used to magnify the roots and beautify the curves, and the concept of transformers in Seq2seq neural networks [Wiki]. But the true connection to synthetic data is about how the simulations can be used to generate random functions mimicking the deterministic ones that we deal with in real life. This also applies to the random Rademacher functions discussed in section 15.3.5, mimicking deterministic multiplicative functions, also related to prime numbers as in this section.

The functions discussed here have a variable number of roots, a certain behavior (piecewise constant, roots are spaced out) and a certain potential range of values depending on the argument. These values, for a specific argument, are typically equally likely to show up. There is also some relative independence between (say) $f(b)$ and $f(b+1)$. These are the parameters of the real-life functions $f(b) = a \bmod b$ where a is a large fixed integer number with few divisors. Remember that the initial goal was to factor a product a two very large primes – a very hard problem of considerable interest in cryptography. Here the product in question is the number a.

To study the divergent fixed-point algorithm, and to benchmark and improve it in this context, I used synthetic random functions that mimic the properties of the original functions. Each random function is uniquely determined by the `seed` parameter. Thus, we have access to an infinite collection of functions, both for the real ones (determined by a), and the random ones. Assessing whether the synthetic functions are a good fit or not, as a proxy to represent the real-life functions, is the central "synthetic data" problem. The functions can be categorized in different types, depending on the number of roots and other parameter values. The problem is to simulate functions that can be used as substitutes, in each category.

To achieve this goal, some random functions must be ruled out from some categories, as being too different from the target functions that they try to emulate. Comparing two functions (a random one with a set of real-life functions within a specific category) is done by comparing their core parameters (number of roots and so on). In particular, random functions with no root below some threshold, are rejected. The mechanism that produces the suitable random functions is called a generative model. It is discussed in its most general form, in in section 1.4.