Kordel France
Lab 3
Class 605.621 Section 83
24 August 2021

Lab 4 Analysis Document

FranceLab3 is a signal processing program that detects specified patterns within a binary signal. The detection algorithm is analyzed against increasingly complex signals and increasingly complex detections. *FranceLab3* is a software module written in Python 3.7 that facilitates such algorithms[1].

Signal Detection Algorithm

The *detectSignal* algorithm is a signal processing algorithm that recognizes patterns within a binary signal. My intent was to model it as a state machine such that the algorithm is in different "states" as it iterates through the signal looking for pattern *X* or pattern *Y*. Pseudocode for the algorithm is provided below. Please note that it is slightly modified from the actual algorithm in *FranceLab3* in order to provide interpretable pseudocode. The algorithm in the project was modified with a driver function in order to effectively communicate cost runs to the user.

```
    1) /* This function detects the specified X and Y patterns in a given transmission signal. It also sets
       up the state machine, checks for noise and keeps track of operational cost.
       /*

function detectPatternsInSignal ( xPattern, yPattern, sSignal ) returns [ bool, bool, bool, int ] {
        xFound, yFound = false                  // X, Y patterns found
        xIndex, yIndex = 0                      // current index of detected X, Y patterns
        xMaxIndex, yMaxIndex = 0                // length of X, Y patterns
        xTracked, yTracked = [ ]                // a list of values detected that belong to X, Y
        xScanning, yScanning = false            // elements of X, Y detected, we are scanning
        noiseFlag = false                       // flag indicating excessive noise in signal
        noiseCounter = 0                        // counter used to determine excessive noise
        costCounter = 0                         // metric to analyze cost of detection
        STATE = 0                               // represents the state of detection

        for ( int i = 0; i < sSignal.count; i++ ) {
                // check for excessive noise
                if (sSignal [ i ] == 0) && (sSignal[ i – 1] == 0) {
                        noiseCounter += 1
                } else {
                        noiseCounter = 0
                } endif

                if ( noiseCounter > xMaxIndex ) || ( noiseCounter > yMaxIndex ) {
```

[1] A table is included in Appendix A that shows the operational cost along with asymptotic regression equations for all detection scenarios side by side.

```
    noiseFlag = True
} endif

/* this is a Rule object that is used to determine the future state of the state machine given
values in its current state; initialize rule here.
*/
rule = Rule( sSignal[ i ],
    xPattern[ xIndex ],
    yPattern [yIndex],
     xScanning,
     yScanning,
    xTracked,
    yTracked,
    STATE )

rule.decideNextState( )              // call "decide" to determine next state

STATE = rule[ 0 ]                    // the next state resulting from evaluated rule
xIndex = rule[ 1 ]                   // the current index of X being scanned for
yIndex = rule[ 2 ]                   // the current index of Y being scanned for
xTracked = rule [ 3 ]                // detected values of the X pattern
yTracked  = rule [ 4 ]               // detected values of the Y pattern

/* if we aren't looking at noise, increment the cost counter; in theory, a signal processing
algorithm will use more energy to process non-noise over noise, so this aims to model
by incrementing the cost upon receipt of "signal chatter" only. STATE = 0 is noise.
*/
if ( STATE != 0 ) {
    costCount += 1
} endif

/* Is the pattern (subarray) we've detected equivalent to the specified X or Y signal
patterns? In other words, should we still be scanning for the X and Y patterns?
*/
xScanning = checkForTermination( xTracked, xPattern )
yScanning = checkForTermination( yTracked, yPattern )

/* if we are not still scanning for X, Y, or both, then they must be found within the signal
so set the flags and, if both patterns are found, terminate and return.
if ( ! xScanning ) && ( ! yScanning ) {
    xFound = True
```

```
                yFound = True
                return True, True, noiseFlag, costCount
            } elif ( ! xScanning ) {
                xFound = True
            } elif ( ! yScanning ) {
                yFound = True
            } endif

    return xFound, yFound, noiseFlag, costCount
} endfunction
```

2) */* This function compares two arrays (two signals, two patterns) and determines if they are equivalent.*
*/**

```
function checkForTermination( refSignal, compSignal ) returns bool {
    /* if the number of values in the reference signal are not equivalent to the number of values
    in the comparison signal, obviously the signals aren't equivalent so terminate.
    */
    if (refSignal.count != baseSignal.count ):
        return True
    } endif

    /* otherwise, check each value of comparison signal to value of equivalent index in reference
    signal to ensure equivalency/
    */
    for ( int i=0; i<compSignal.count; i++) {
        if ( refSignal[ i ] != compSignal[ i ]:
            return True
        } endif
    } endloop

    return False
} endfunction
```

3) */* This is a specification for a Rule object. The Rule object takes in the current state parameters of a state machine and determines its next state. We don't detail all of its functionality here, but outline the object specification along with its primary function.*
*/**

```
Class Rule ( self, S, X, Y, xScanning, yScanning, xTracked, yTracked, STATE ) {
    initialize ( self, S, X, Y, xScanning, yScanning, xTracked, yTracked, STATE ) {
        self.X = X
        self.Y = Y
        self.xScanning = xScanning
        self.yScanning = yScanning
        self.xTracked = xTracked
        self.yTracked = yTracked
        self.STATE = STATE
    } endinit

    function decideNextState ( self ) returns [ int, int, int, int [], int[] ] {
        /* The logic to determine the next state is complex and cumbersome, so we do not
        detail it here. However, this function takes into argument the current state with its
        parameters and returns the following:
            int STATE -> the new state of the state machine
            int xIndex -> the next index of the X pattern to be scanned for
            int yIndex -> the next index of the Y pattern to be scanned for
            int [ ] xTracked -> an array of detected sequential values from the X pattern
            int [ ] yTracked -> an array of detected sequential values from the Y pattern
        */
        return STATE, xIndex, yIndex, xTracked, yTracked
    } endfunction
} endclass
```

Asymptotic Behavior of Signal Detection Algorithm

It was a bit difficult to find a reliable metric for the signal detection algorithm. For example, *Quicksort* has two operations that make it particularly easy to analyze its runtime for – comparisons and exchanges. This algorithm is effectively a linear search algorithm so its costs are similar. The most reliable metric I could find for determining algorithmic cost was largely based on the size of the *X* and *Y* patterns being detected. Below we will discuss the asymptotic behavior of the *detectPatternsInSignal* recognition algorithm specified above with a particular focus on the size of the known signals detected by the algorithm as a key contributor to its asymptotic cost.

To clarify some terms, the following establishes the definition of the constants *c, k, m,* and *n* as they are used throughout this document.

n: the length of the transmission signal *S*

m: the length of the *X* and *Y* signals to detect *k:* the number of states in the state machine

c: some small constant, as is typically defined

1. To begin, the *detectPatternsInSignal* algorithm initializes some global parameters that handle performance and track the rules for the state machine. These have cost ***O (c)*** and do not significantly contribute to the system, but we consider them anyway.

   Cost: $O (c)$

2. One of the requirements of the lab was to address whether or not a signal contained excess noise in its transmission. The *detectPatternsInSignal* algorithm then traverses through the entire *S* signal (the transmission signal) to detect if there is excess noise. With a few constant instructions nested inside the loop, this runs in ***O (cn)*** time.

   Cost: $O (cn) \Rightarrow O (n)$

3. The algorithm then sets up another identical loop through the entire *S* signal. This loop will construct and update the state machine with each observed value in *S*. While this loop runs in *O(n)* time, there is an instruction set inside that needs to be considered. The first execution within the loop is to construct the current rule of the state machine and determine the next state. This all runs in *O(kc)* time where $k < 20$ is a finite number of states. Upon completion of the rule evaluation and determination of the current state, we increment the cost (if applicable) and perform two calls to the *checkForTermination* function. The *checkForTermination* performs in *O(m)* time where $m <= n$ is the length of the pattern signal *X* or *Y* to recognize. At a worst case, we can consider $m = n$ such that *checkForTermination* performs in *O(n)* time. Finally, we have a few statements at the end of the loop that check for whether or not the *X* and *Y* patterns have been found that run in *O(c)* time. So to consolidate all of the costs within this loop, we have the following:

   Best Case Cost: $O (n * (kc + m + m +c)) \Rightarrow O (n * (2m)) \Rightarrow O (nm)$
   Average Case Cost: $O (n * (kc + m + m +c)) \Rightarrow O (n * (2m)) \Rightarrow O (nm)$
   Worst Case Cost: $O (n * (kc + n + n +c)) \Rightarrow O (n * (2n)) \Rightarrow O (n^2)$

4. We can consolidate steps 1-3 to receive the total expected cost for the best, worst, and average case scenarios. Summing the costs from steps 1-3, we receive the following:

   Best Case Cost: $T (n) = O (c) + O (n) + O (nm)$
   $\Rightarrow O (n + nm)$

   Average Case Cost: $T (n) = O (c) + O (n) + O (nm)$
   $\Rightarrow O (n + nm)$

Worst Case Cost: $T(n) = O(c) + O(n) + O(n^2)$
$\Rightarrow O(n + n^2)$

One might notice that the runtimes above for the detection algorithm are grater than that of regular linear search which runs in $O(n^2)$ time. After all, we are essentially doing regular linear search on an array for a contained subarray. The additional complexity acquired by *detectPatternsInSignal* lies within the screening for excessive noise in the loop that proceeds the state machine loop. There is opportunity here for some optimization that allows for a solution that converges toward the complexity of linear search which will be discussed in a later section.

One might wonder why linear search is used to perform the pattern recognition in the signal. After all, linear search is known for its inefficiency against other more robust methods such as binary search. The methodology behind selecting linear search lies within two primary reasons. The first reason is that the two signal patterns we are looking for *X* and *Y* are linear subsets and in the event the binary search pivot dissects either of the two patterns, the search algorithm could potentially fail to detect the pattern. *X* and *Y* also may be entangled, in which unentangling the two patterns proves very complex in a more otherwise robust search method. The second reason for leveraging linear search is that it gives the program a more practical application. Linear search allows the signal to be interpreted in real time as it is transmitted so that the entire signal doesn't have to be received before detection of *X* and *Y* can begin. In the industry of defense and security, one can easily see why this is advantageous.

Cost Analysis for All Quicksort Algorithms

Space Complexity

All algorithms resemble the same best, worst, and average case time complexities. Consider the pseudocode that defines the *detectPatternsInSignal* algorithm in the first section. There are several global variables that initialize the state machine and track performance. There are less than 10 of these variables so we can consider them having space complexity $O(c)$. The first for loop that checks for excess noise occupies n space but that space is relieved upon its completion. The second for-loop is what defines the bound on the cost. The second loop performs in $O(n)$, but inside the loop we maintain two subarrays (one each for *X* and *Y*) of values that indicate which values of *X* and *Y* have been detected. Since each of these two signals has m length at a best/average case and n length at a worst case, each array has space complexity in proportion to their length. Aside from this, we can consider each of the k states in the state machine to occupy $O(c)$ space, and we account for some other error checking logic within the second loop that maintains $O(c)$ space as well. For the best case space complexity, we consider the scenario where only one signal is desired to be found. Therefore, we have the following that indicates total space complexity:

Best Case Space: $O(c + n + n + m + kc + c) \Rightarrow O(2n + m)$

Average Case Space: $O(c + n + n + m + m + kc + c) \Rightarrow O(2n + 2m)$

Worst Case Space: $O(c + n + n + n + n + kc + c) \Rightarrow O(4n)$

It can be seen that with n >> m, the space complexities for best, worst, and average case runtimes all converge to $O(n)$.

Time Complexity

Please refer to the table below for acquired runtime costs for each different data configuration under the *detectPatternsInSignal* algorithm in the *FranceLab3* module. The table shows the cost and SNR for each algorithm on each of the three datasets for different sizes of *m* and *n.* In order to more accurately assess just how fast the number of operations scaled with respect to changes in *n*, I fit exponential regression curves of the form *y = ax^b* to each algorithm over each dataset. The last two columns in the table give these equations to help validate the asymptotic trajectory of each algorithm. Please reference Table 1 below for the following sections.

Detecting Only X or Only Y

We first analyze the cost of detecting only one signal (either *X* or *Y,* but not both). Refer to lines 1-40 in Table 1 for reference to these cost runs. If *X* and *Y* are of equal size *m,* then we wouldn't expect to see much variance in their costs – theoretically the costs of detecting one should be equivalent to detecting the other. One will notice that the regression equations in this column roughly show $y = x^1 = x \Rightarrow n$ for almost every run. This validates the cost and scalability of linear search. High correlations with the regression function also confirm this. The cost grows with direct proportionality to the sizes of the transmission signal *S* and the signals to be detected *X* and *Y.* Figure 1 shows a consolidated graph of each cost run for detecting only *X* in *S* for values of *n* <= 3000. Figure 2 shows the converse scenario with a consolidated graph of each cost run detecting only *Y* in *S.* Visually, we don't see much deviation between the two which confirms uniformity in our algorithm and shows that the algorithm knows how to transition between detecting both signals versus detecting only one signal. One may also note the general decline in *SNR* as the gap between *n* and *m* widens.

Since we defined the upper bound runtime of our algorithm at a worst case to be $O(n + n^2)$, it's worth noting that the algorithmic cost stays below this for each trace run in the table. We can do some quick math to verify that the value in the *cost* column is less than the squared value in the *n* column.

The more interesting scenario arises when both *X* and *Y* need to be detected in *S.*

#	data_type	n	m	cost	SNR	noise trajectory equation	cost trajectory equation
1	x detected	50	3-pattern□	90	0.0800	y = 0.9996 (x) ^ 1.0919 w/ correlation 100.0 %	y = 1.0077 (x) ^ 0.2095 w/ correlation 99.6 %
2	x detected	100	3-pattern□	186	0.0400	y = 0.9996 (x) ^ 1.0919 w/ correlation 100.0 %	y = 1.0077 (x) ^ 0.2095 w/ correlation 99.6 %
3	x detected	500	3-pattern□	881	0.0080	y = 0.9996 (x) ^ 1.0919 w/ correlation 100.0 %	y = 1.0077 (x) ^ 0.2095 w/ correlation 99.6 %
4	x detected	1000	3-pattern□	1894	0.0040	y = 0.9996 (x) ^ 1.0919 w/ correlation 100.0 %	y = 1.0077 (x) ^ 0.2095 w/ correlation 99.6 %
5	x detected	1500	3-pattern□	2990	0.0027	y = 0.9996 (x) ^ 1.0919 w/ correlation 100.0 %	y = 1.0077 (x) ^ 0.2095 w/ correlation 99.6 %
6	x detected	50	5-pattern□	73	0.1200	y = 1.0083 (x) ^ 1.0851 w/ correlation 100.0 %	y = 0.9876 (x) ^ 0.245 w/ correlation 99.2 %
7	x detected	100	5-pattern□	184	0.0600	y = 1.0083 (x) ^ 1.0851 w/ correlation 100.0 %	y = 0.9876 (x) ^ 0.245 w/ correlation 99.2 %
8	x detected	500	5-pattern□	929	0.0120	y = 1.0083 (x) ^ 1.0851 w/ correlation 100.0 %	y = 0.9876 (x) ^ 0.245 w/ correlation 99.2 %
9	x detected	1000	5-pattern□	1685	0.0060	y = 1.0083 (x) ^ 1.0851 w/ correlation 100.0 %	y = 0.9876 (x) ^ 0.245 w/ correlation 99.2 %
10	x detected	1500	5-pattern□	2931	0.0040	y = 1.0083 (x) ^ 1.0851 w/ correlation 100.0 %	y = 0.9876 (x) ^ 0.245 w/ correlation 99.2 %
11	x detected	50	10-pattern□	65	0.2200	y = 0.9977 (x) ^ 1.082 w/ correlation 100.0 %	y = 1.003 (x) ^ 0.8839 w/ correlation 100.0 %
12	x detected	100	10-pattern□	163	0.1100	y = 0.9977 (x) ^ 1.082 w/ correlation 100.0 %	y = 1.003 (x) ^ 0.8839 w/ correlation 100.0 %
13	x detected	500	10-pattern□	811	0.0220	y = 0.9977 (x) ^ 1.082 w/ correlation 100.0 %	y = 1.003 (x) ^ 0.8839 w/ correlation 100.0 %
14	x detected	1000	10-pattern□	1795	0.0110	y = 0.9977 (x) ^ 1.082 w/ correlation 100.0 %	y = 1.003 (x) ^ 0.8839 w/ correlation 100.0 %
15	x detected	1500	10-pattern□	2471	0.0073	y = 0.9977 (x) ^ 1.082 w/ correlation 100.0 %	y = 1.003 (x) ^ 0.8839 w/ correlation 100.0 %
16	x detected	50	20-pattern□	78	0.4200	y = 0.9891 (x) ^ 1.0871 w/ correlation 100.0 %	y = 0.793 (x) ^ 0.6324 w/ correlation 73.7 %
17	x detected	100	20-pattern□	146	0.2100	y = 0.9891 (x) ^ 1.0871 w/ correlation 100.0 %	y = 0.793 (x) ^ 0.6324 w/ correlation 73.7 %
18	x detected	500	20-pattern□	762	0.0420	y = 0.9891 (x) ^ 1.0871 w/ correlation 100.0 %	y = 0.793 (x) ^ 0.6324 w/ correlation 73.7 %
19	x detected	1000	20-pattern□	1992	0.0210	y = 0.9891 (x) ^ 1.0871 w/ correlation 100.0 %	y = 0.793 (x) ^ 0.6324 w/ correlation 73.7 %
20	x detected	1500	20-pattern□	2774	0.0140	y = 0.9891 (x) ^ 1.0871 w/ correlation 100.0 %	y = 0.793 (x) ^ 0.6324 w/ correlation 73.7 %
21	y detected	50	3-pattern□	93	0.0800	y = 0.9809 (x) ^ 1.0716 w/ correlation 99.9 %	y = 1.0077 (x) ^ 0.2095 w/ correlation 99.6 %
22	y detected	100	3-pattern□	187	0.0400	y = 0.9809 (x) ^ 1.0716 w/ correlation 99.9 %	y = 1.0077 (x) ^ 0.2095 w/ correlation 99.6 %
23	y detected	500	3-pattern□	632	0.0080	y = 0.9809 (x) ^ 1.0716 w/ correlation 99.9 %	y = 1.0077 (x) ^ 0.2095 w/ correlation 99.6 %
24	y detected	1000	3-pattern□	1912	0.0040	y = 0.9809 (x) ^ 1.0716 w/ correlation 99.9 %	y = 1.0077 (x) ^ 0.2095 w/ correlation 99.6 %
25	y detected	1500	3-pattern□	2935	0.0027	y = 0.9809 (x) ^ 1.0716 w/ correlation 99.9 %	y = 1.0077 (x) ^ 0.2095 w/ correlation 99.6 %
26	y detected	50	5-pattern□	96	0.1200	y = 0.9944 (x) ^ 1.0917 w/ correlation 100.0 %	y = 1.01 (x) ^ 0.2708 w/ correlation 99.6 %
27	y detected	100	5-pattern□	181	0.0600	y = 0.9944 (x) ^ 1.0917 w/ correlation 100.0 %	y = 1.01 (x) ^ 0.2708 w/ correlation 99.6 %
28	y detected	500	5-pattern□	831	0.0120	y = 0.9944 (x) ^ 1.0917 w/ correlation 100.0 %	y = 1.01 (x) ^ 0.2708 w/ correlation 99.6 %
29	y detected	1000	5-pattern□	1970	0.0060	y = 0.9944 (x) ^ 1.0917 w/ correlation 100.0 %	y = 1.01 (x) ^ 0.2708 w/ correlation 99.6 %
30	y detected	1500	5-pattern□	2645	0.0040	y = 0.9944 (x) ^ 1.0917 w/ correlation 100.0 %	y = 1.01 (x) ^ 0.2708 w/ correlation 99.6 %
31	y detected	50	10-pattern□	75	0.2200	y = 1.0057 (x) ^ 1.0834 w/ correlation 100.0 %	y = 1.2031 (x) ^ 0.5603 w/ correlation 77.2 %
32	y detected	100	10-pattern□	143	0.1100	y = 1.0057 (x) ^ 1.0834 w/ correlation 100.0 %	y = 1.2031 (x) ^ 0.5603 w/ correlation 77.2 %
33	y detected	500	10-pattern□	894	0.0220	y = 1.0057 (x) ^ 1.0834 w/ correlation 100.0 %	y = 1.2031 (x) ^ 0.5603 w/ correlation 77.2 %
34	y detected	1000	10-pattern□	1700	0.0110	y = 1.0057 (x) ^ 1.0834 w/ correlation 100.0 %	y = 1.2031 (x) ^ 0.5603 w/ correlation 77.2 %
35	y detected	1500	10-pattern□	2349	0.0073	y = 1.0057 (x) ^ 1.0834 w/ correlation 100.0 %	y = 1.2031 (x) ^ 0.5603 w/ correlation 77.2 %
36	y detected	50	20-pattern□	70	0.4200	y = 0.9973 (x) ^ 1.0908 w/ correlation 100.0 %	y = 1.017 (x) ^ 0.4602 w/ correlation 99.6 %
37	y detected	100	20-pattern□	151	0.2100	y = 0.9973 (x) ^ 1.0908 w/ correlation 100.0 %	y = 1.017 (x) ^ 0.4602 w/ correlation 99.6 %
38	y detected	500	20-pattern□	853	0.0420	y = 0.9973 (x) ^ 1.0908 w/ correlation 100.0 %	y = 1.017 (x) ^ 0.4602 w/ correlation 99.6 %
39	y detected	1000	20-pattern□	1914	0.0210	y = 0.9973 (x) ^ 1.0908 w/ correlation 100.0 %	y = 1.017 (x) ^ 0.4602 w/ correlation 99.6 %
40	y detected	1500	20-pattern□	2694	0.0140	y = 0.9973 (x) ^ 1.0908 w/ correlation 100.0 %	y = 1.017 (x) ^ 0.4602 w/ correlation 99.6 %
41	x and y detected	50	3-pattern□	68	0.1200	y = 0.9975 (x) ^ 1.096 w/ correlation 100.0 %	y = 0.9847 (x) ^ 0.3155 w/ correlation 99.3 %
42	x and y detected	100	3-pattern□	194	0.0600	y = 0.9975 (x) ^ 1.096 w/ correlation 100.0 %	y = 0.9847 (x) ^ 0.3155 w/ correlation 99.3 %
43	x and y detected	500	3-pattern□	883	0.0120	y = 0.9975 (x) ^ 1.096 w/ correlation 100.0 %	y = 0.9847 (x) ^ 0.3155 w/ correlation 99.3 %
44	x and y detected	1000	3-pattern□	1981	0.0060	y = 0.9975 (x) ^ 1.096 w/ correlation 100.0 %	y = 0.9847 (x) ^ 0.3155 w/ correlation 99.3 %
45	x and y detected	1500	3-pattern□	2306	0.0040	y = 0.9975 (x) ^ 1.096 w/ correlation 100.0 %	y = 0.9847 (x) ^ 0.3155 w/ correlation 99.3 %
46	x and y detected	50	5-pattern□	87	0.2000	y = 1.0182 (x) ^ 1.0756 w/ correlation 99.9 %	y = 1.0128 (x) ^ 0.348 w/ correlation 99.6 %
47	x and y detected	100	5-pattern□	185	0.1000	y = 1.0182 (x) ^ 1.0756 w/ correlation 99.9 %	y = 1.0128 (x) ^ 0.348 w/ correlation 99.6 %
48	x and y detected	500	5-pattern□	975	0.0200	y = 1.0182 (x) ^ 1.0756 w/ correlation 99.9 %	y = 1.0128 (x) ^ 0.348 w/ correlation 99.6 %
49	x and y detected	1000	5-pattern□	1460	0.0100	y = 1.0182 (x) ^ 1.0756 w/ correlation 99.9 %	y = 1.0128 (x) ^ 0.348 w/ correlation 99.6 %
50	x and y detected	1500	5-pattern□	2380	0.0067	y = 1.0182 (x) ^ 1.0756 w/ correlation 99.9 %	y = 1.0128 (x) ^ 0.348 w/ correlation 99.6 %
51	x and y detected	50	10-pattern□	82	0.4000	y = 1.0005 (x) ^ 1.0889 w/ correlation 100.0 %	y = 0.8934 (x) ^ 0.8041 w/ correlation 94.4 %
52	x and y detected	100	10-pattern□	137	0.2000	y = 1.0005 (x) ^ 1.0889 w/ correlation 100.0 %	y = 0.8934 (x) ^ 0.8041 w/ correlation 94.4 %
53	x and y detected	500	10-pattern□	874	0.0400	y = 1.0005 (x) ^ 1.0889 w/ correlation 100.0 %	y = 0.8934 (x) ^ 0.8041 w/ correlation 94.4 %
54	x and y detected	1000	10-pattern□	1840	0.0200	y = 1.0005 (x) ^ 1.0889 w/ correlation 100.0 %	y = 0.8934 (x) ^ 0.8041 w/ correlation 94.4 %
55	x and y detected	1500	10-pattern□	2643	0.0133	y = 1.0005 (x) ^ 1.0889 w/ correlation 100.0 %	y = 0.8934 (x) ^ 0.8041 w/ correlation 94.4 %
56	x and y detected	50	20-pattern□	78	0.8000	y = 0.9898 (x) ^ 1.0835 w/ correlation 100.0 %	y = 0.9977 (x) ^ 0.9381 w/ correlation 100.0 %
57	x and y detected	100	20-pattern□	178	0.4000	y = 0.9898 (x) ^ 1.0835 w/ correlation 100.0 %	y = 0.9977 (x) ^ 0.9381 w/ correlation 100.0 %
58	x and y detected	500	20-pattern□	751	0.0800	y = 0.9898 (x) ^ 1.0835 w/ correlation 100.0 %	y = 0.9977 (x) ^ 0.9381 w/ correlation 100.0 %
59	x and y detected	1000	20-pattern□	1931	0.0400	y = 0.9898 (x) ^ 1.0835 w/ correlation 100.0 %	y = 0.9977 (x) ^ 0.9381 w/ correlation 100.0 %
60	x and y detected	1500	20-pattern□	2096	0.0267	y = 0.9898 (x) ^ 1.0835 w/ correlation 100.0 %	y = 0.9977 (x) ^ 0.9381 w/ correlation 100.0 %

Table 1 - Actual costs with expected asymptotic costs fitted through exponential regression for different values of m and n.

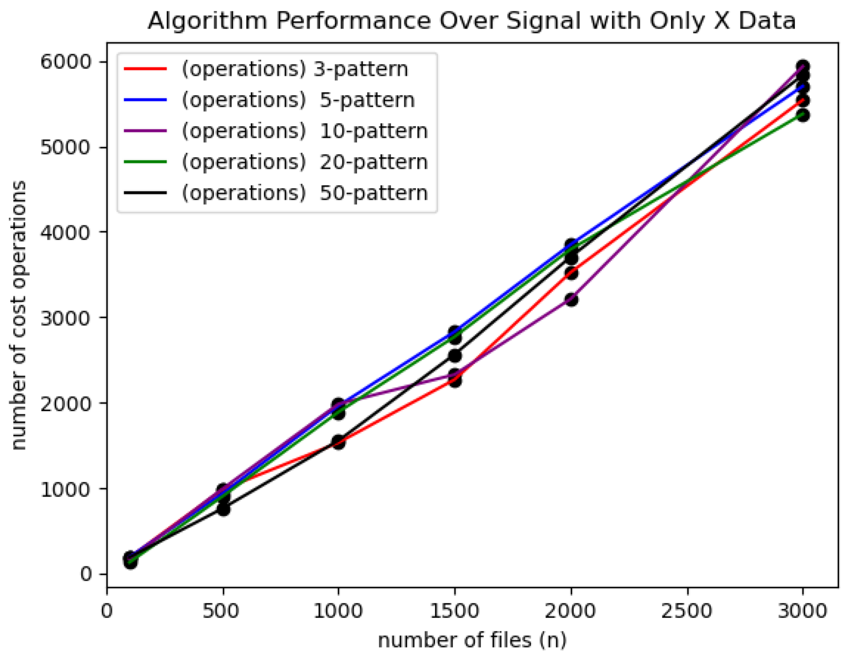

Figure 1 - A display of the asymptotic growth of the detection algorithm searching for only X over successively larger values of m and n.

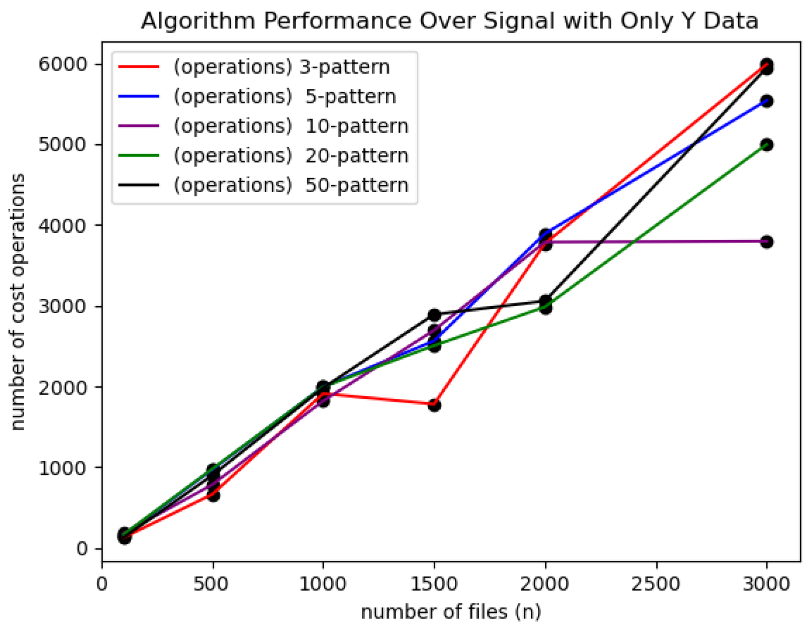

Figure 2 - A display of the asymptotic growth of the detection algorithm searching for only Y over successively larger values of m and n.

Detecting Both X and Y

We discover a lot more about how the algorithm performs in scenarios where we tell the algorithm to detect both *X* and *Y* in *S.* Please refer to lines 40-60 in the table as well as Figure 3 for details on this.

Looking strictly at the data in the table, we observe a similar scenario in regression coefficients as observed when detecting only one pattern—regression equations all approximate to $y = x^1 = x \rightarrow n$. Again, this confirms the linear search properties. Contrary to what we observed in the preceding section, we see more volatility in the cost as *n* gets larger, but not as *m* gets larger. This might seem counterintuitive at first, but makes sense upon further analysis. We expect a signal (subarray) with small *m* to have higher volatility in cost because there is a much larger range of indices the signal may reside in as *n* scales. We observe this with the red line in Figure 3 showing increased volatility of the 3-pattern signal as *n* grows. However, as *m* also gets larger, the number of possible indices that the subarray can occupy in the signal decreases, so the cost of acquiring the signal converges toward a smaller amount of possible values.

We can quantify all of this in a single statement by saying the detection algorithm has higher cost volatility for small *m* and large *n.* There is a direct correlation here between this relationship and the *SNR* that is generated for each run. The statistics now start to confirm common signal processing principles.

Figure 4 duplicates the results of Figure 3. We observe similar patterns over a different dataset which begins to confirm the generalization of the detection algorithm.

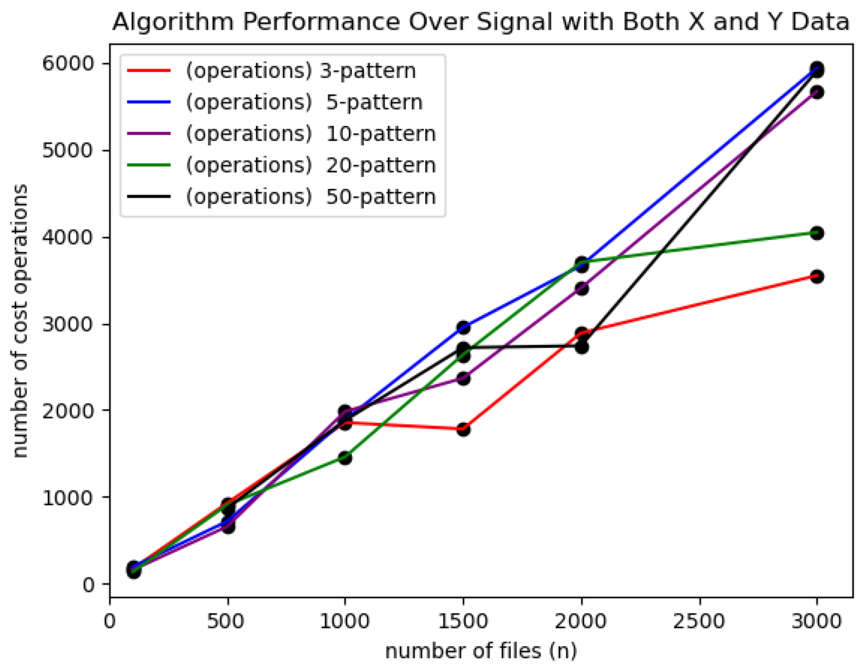

Figure 3 - A display of the asymptotic growth of the detection algorithm searching for both X and Y in S over successively larger values of m and n. Trial 1.

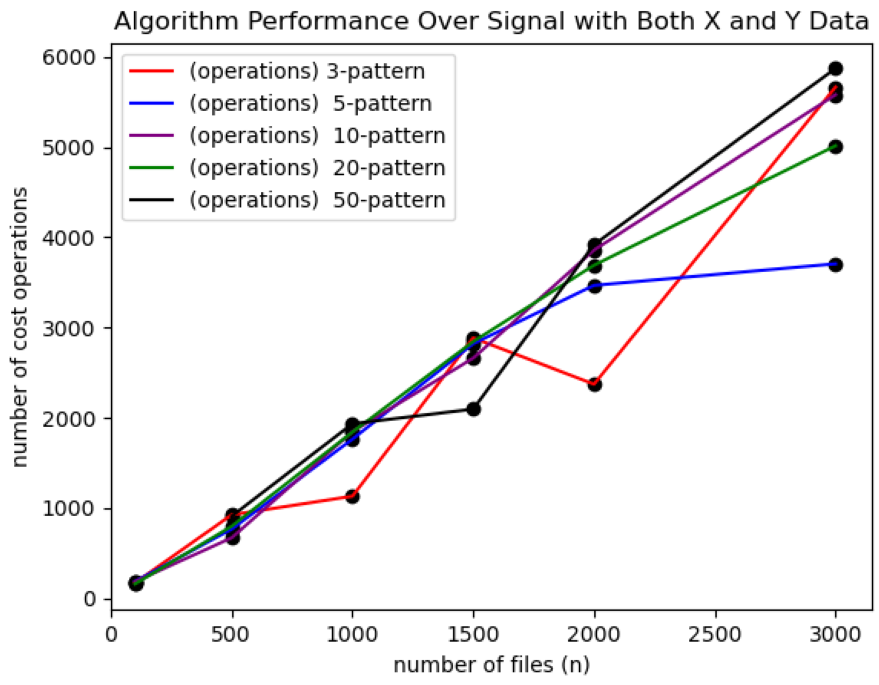

Figure 4 - A display of the asymptotic growth of the detection algorithm searching for both X and Y in S over successively larger values of m and n. Trial 2 to show reproducibility of the variance that scales with increases in n and m.

A table is included in *Appendix A* that shows the operational counts along with asymptotic regression equations for a larger dataset over a broader range of cost runs.

Optimizations for Later Versions

Right now, the detection algorithm runs iteratively. In a later version, I would rewrite the algorithm to operate recursively, specifically with tail recursion. This would simplify the code and more accurately implement a true state machine. This would be the first optimization I would make given more time.

Additionally, there is some opportunity to simplify the detection algorithm to run in $O(n^2)$ time instead of its current $O(n + n^2)$ runtime. To do this would be simple: instead of prescreening for excessive noise *before* initializing the state machine, screen for excessive noise at each state *inside* of the state machine. This adds *c* amount of cost to an n^2 loop and relieves another loop entirely driving the cost to $O(cn^2) \rightarrow O(n^2)$. This allows the algorithm to directly compare to linear search and models the Maximum Subarray problem a bit more accurately.

Kordel France
Lab 3
Class 605.621 Section 83
24 August 2021

Enhancements

The program provides 9 major enhancements (along with many other minor enhancements) on top of the original requirements. These enhancements are also elaborated upon in the *.README* document associated with the module

1) Time delays - processing is paused briefly throughout the program to allow the user time to read and interpret the output. This creates for a much better user experience.

2) Execution time is tracked and monitored for each sorting algorithm.

3) Each sorting run is graphed in a `*.csv*` file. Files are named {algorithm_name}-{date_type}-{n} count.csv such as *'3pattern_x and y _1000count.csv'* which specifies a cost run for the algorithm detecting both *X* and *Y* of size *m = 3* over a dataset of size *n* Each file is located in the *output_files* directory and contains the following properties:
   - Signal type
   - Signal length
   - Pattern type (detect only *X*, detect only *Y*, detect both *X* and *Y)*
   - number of operations performed in detecting patterns
   - ending index of the last detected pattern
   - an equation that plots the trajectory of the number of operations made by the algorithm over this data type as `n` scales along with the correlation coefficient between the equation and the observed data.
   - an equation that plots the trajectory of the average ending index made by the algorithm over this data type as `n` scales along with the correlation coefficient between the equation and the observed data.
   - the initial *S* transmission signal
   - the *X* pattern signal
   - the *Y* pattern signal
   - the execution time (in seconds) for the algorithm to completely sort the data
   - the signal-to-noise ratio (SNR) for each transmission *S*

4) A `Summary Table` is provided at the very end of the program that shows the performance of each algorithm over different data distributions. A *`FINAL_ANALYSIS.csv`* file is a direct copy of the *`Summary Table`*, but in a `*.csv*` file that shows performance over all trace runs.

5) Equations for trajectory curves were calculated to extrapolate the number of comparisons exchanges that would be theoretically needed for very large *n* as the algorithm scales. This was accomplished by calculating coefficients of power regression to define the trajectory path based on the data gathered for similar sorts. If one opens *Metric.py* where the regression algorithms are located, they will notice the regression curve is computed from scratch with no "packages" - the

regression equation is derived from low-level statistics functions. The `*numpy*` package is only used to cast an array of one type to a type easier to manipulate with these statistics functions; `*numpy*` plays no role in computation.

6) Correlation values are calculated (again from scratch and without any use of packages) to show how well the above regression curve fits the empirically gathered data in our analysis.

7) A status is communicated to the user as a percentage complete in the `*__main__.py*` file while the data is being processed and sorted. This allows for a more appealing user interface and lets the user have an idea of where the program is at in its execution steps.

8) Plots of all of the data runs for each algorithm are shown and allowed for easy comparison against other algorithms. This makes it simple for the user to spot analytical trends and spot which algorithms out-perform others on certain datasets.

9) The signal-to-noise ratio (SNR) for each transmission signal S is provided with each cost run as a real-world metric on signal processing cost.

Lessons Learned

Once again, my appreciation for state machines and Turing machines increased with this lab. I was able to expand my skillset in addressing real problems with these techniques. I appreciated how this lab had a practical application in signal processing and pattern recognition. Although the pattern detection happened over a simple array of binary values, the methods used to detect patterns within the array followed general signal processing practices. It is easy to take the detection algorithm built for this lab and scale it up to accommodate higher data dimensionality and additional detection complexity.

I heavily follow the machine learning and artificial intelligence state of the art and pattern recognition is a technique heavily used within these practices. However, pattern recognition is often advertised in these circles as advanced machine learning software packages and neural networks, but rarely gives credence to the fundamentals existing behind these packages. This lab allowed me to see how some of these "black box" recognition algorithms work at a fundamental level and how sophisticated solutions can be constructed through foundational mathematics and computer science. Not every pattern recognition software has to be a neural network. The transparency of a sophisticated state machine in signal processing allows one to remove the black box stereotype and actually make the algorithm explainable.

Conclusion

FranceLab3 addresses the requirements as specified in the *Lab 3* handout. Specifically, the program constructs a pattern recognition algorithm in the form of a state machine that is capable of discerning up to two different signals in a given larger transmission signal. *FranceLab3* autonomously generates a multitude of cost runs that diagnose the performance of the detection algorithm. Finally, the program consolidates the performance of all cost runs into summary tables and graphs so that the user can assess asymptotic performance of the algorithm in both time and space.

References

The following items were used as references for the construction of this project.

1) Cormen, T. H., & Leiserson, C. E. (2009). *Introduction to Algorithms, 3rd edition.*

2) Miller, B. N., & Ranum, D. L. (2014). *Problem solving with algorithms and data structures using Python (2nd ed.).* Decorah, IA: Brad Miller, David Ranum

3) *K-way merge algorithm.* (2021, April 09). Retrieved April 24-27, 2021, from https://en.wikipedia.org/wiki/K-way_merge_algorithm

4) *Artificial Intelligence: A Modern Approach. Third Edition*. Russel, Stuart J.; Norvig, Peter. 2015, Pearson India Education Services Pvt. Ltd. p 961-962.

5) *Deep Learning.* Goodfellow, Ian; Bengio, Yoshua; Courville, Aaron. 2016, Massachusetts Institute of Technology. p 147 -149, 525 – 527.

6) Garey, M. R., & Johnson, D. S. (2003). *Computers and Intractability: A Guide to the Theory of NP - Completeness.* W.H. Freeman and Co.

7) Kleinberg, J., & Tardos, É. (2014). *Algorithm Design.* Pearson India Education Services Pvt Ltd.

Kordel France
Lab 3
Class 605.621 Section 83
24 August 2021

Appendix A

#	data_type	n	m	cost	SNR	comparison trajectory equation	exchange trajectory equation
1	x detected	100	3-pattern	187	0.0400	y = 0.9974 (x) ^ 1.0662 w/ correlation 100.0 %	y = 1.001 (x) ^ 0.1857 w/ correlation 99.9 %
2	x detected	500	3-pattern	981	0.0080	y = 0.9974 (x) ^ 1.0662 w/ correlation 100.0 %	y = 1.001 (x) ^ 0.1857 w/ correlation 99.9 %
3	x detected	1000	3-pattern	1531	0.0040	y = 0.9974 (x) ^ 1.0662 w/ correlation 100.0 %	y = 1.001 (x) ^ 0.1857 w/ correlation 99.9 %
4	x detected	1500	3-pattern	2269	0.0027	y = 0.9974 (x) ^ 1.0662 w/ correlation 100.0 %	y = 1.001 (x) ^ 0.1857 w/ correlation 99.9 %
5	x detected	2000	3-pattern	3524	0.0020	y = 0.9974 (x) ^ 1.0662 w/ correlation 100.0 %	y = 1.001 (x) ^ 0.1857 w/ correlation 99.9 %
6	x detected	3000	3-pattern	5542	0.0013	y = 0.9974 (x) ^ 1.0662 w/ correlation 100.0 %	y = 1.001 (x) ^ 0.1857 w/ correlation 99.9 %
7	x detected	100	5-pattern	192	0.0600	y = 1.0001 (x) ^ 1.0865 w/ correlation 100.0 %	y = 1.0133 (x) ^ 0.2783 w/ correlation 96.1 %
8	x detected	500	5-pattern	935	0.0120	y = 1.0001 (x) ^ 1.0865 w/ correlation 100.0 %	y = 1.0133 (x) ^ 0.2783 w/ correlation 96.1 %
9	x detected	1000	5-pattern	1963	0.0060	y = 1.0001 (x) ^ 1.0865 w/ correlation 100.0 %	y = 1.0133 (x) ^ 0.2783 w/ correlation 96.1 %
10	x detected	1500	5-pattern	2830	0.0040	y = 1.0001 (x) ^ 1.0865 w/ correlation 100.0 %	y = 1.0133 (x) ^ 0.2783 w/ correlation 96.1 %
11	x detected	2000	5-pattern	3851	0.0030	y = 1.0001 (x) ^ 1.0865 w/ correlation 100.0 %	y = 1.0133 (x) ^ 0.2783 w/ correlation 96.1 %
12	x detected	3000	5-pattern	5704	0.0020	y = 1.0001 (x) ^ 1.0865 w/ correlation 100.0 %	y = 1.0133 (x) ^ 0.2783 w/ correlation 96.1 %
13	x detected	100	10-pattern	164	0.1100	y = 0.9997 (x) ^ 1.0613 w/ correlation 100.0 %	y = 0.9883 (x) ^ 0.2891 w/ correlation 97.1 %
14	x detected	500	10-pattern	989	0.0220	y = 0.9997 (x) ^ 1.0613 w/ correlation 100.0 %	y = 0.9883 (x) ^ 0.2891 w/ correlation 97.1 %
15	x detected	1000	10-pattern	1984	0.0110	y = 0.9997 (x) ^ 1.0613 w/ correlation 100.0 %	y = 0.9883 (x) ^ 0.2891 w/ correlation 97.1 %
16	x detected	1500	10-pattern	2328	0.0073	y = 0.9997 (x) ^ 1.0613 w/ correlation 100.0 %	y = 0.9883 (x) ^ 0.2891 w/ correlation 97.1 %
17	x detected	2000	10-pattern	3213	0.0055	y = 0.9997 (x) ^ 1.0613 w/ correlation 100.0 %	y = 0.9883 (x) ^ 0.2891 w/ correlation 97.1 %
18	x detected	3000	10-pattern	5936	0.0037	y = 0.9997 (x) ^ 1.0613 w/ correlation 100.0 %	y = 0.9883 (x) ^ 0.2891 w/ correlation 97.1 %
19	x detected	100	20-pattern	130	0.2100	y = 0.9999 (x) ^ 1.0841 w/ correlation 100.0 %	y = 0.9021 (x) ^ 0.5759 w/ correlation 68.0 %
20	x detected	500	20-pattern	901	0.0420	y = 0.9999 (x) ^ 1.0841 w/ correlation 100.0 %	y = 0.9021 (x) ^ 0.5759 w/ correlation 68.0 %
21	x detected	1000	20-pattern	1889	0.0210	y = 0.9999 (x) ^ 1.0841 w/ correlation 100.0 %	y = 0.9021 (x) ^ 0.5759 w/ correlation 68.0 %
22	x detected	1500	20-pattern	2771	0.0140	y = 0.9999 (x) ^ 1.0841 w/ correlation 100.0 %	y = 0.9021 (x) ^ 0.5759 w/ correlation 68.0 %
23	x detected	2000	20-pattern	3795	0.0105	y = 0.9999 (x) ^ 1.0841 w/ correlation 100.0 %	y = 0.9021 (x) ^ 0.5759 w/ correlation 68.0 %
24	x detected	3000	20-pattern	5379	0.0070	y = 0.9999 (x) ^ 1.0841 w/ correlation 100.0 %	y = 0.9021 (x) ^ 0.5759 w/ correlation 68.0 %
25	x detected	100	50-pattern	189	0.5100	y = 0.9989 (x) ^ 1.0775 w/ correlation 100.0 %	y = 0.8967 (x) ^ 0.5591 w/ correlation 64.8 %
26	x detected	500	50-pattern	760	0.1020	y = 0.9989 (x) ^ 1.0775 w/ correlation 100.0 %	y = 0.8967 (x) ^ 0.5591 w/ correlation 64.8 %
27	x detected	1000	50-pattern	1555	0.0510	y = 0.9989 (x) ^ 1.0775 w/ correlation 100.0 %	y = 0.8967 (x) ^ 0.5591 w/ correlation 64.8 %
28	x detected	1500	50-pattern	2564	0.0340	y = 0.9989 (x) ^ 1.0775 w/ correlation 100.0 %	y = 0.8967 (x) ^ 0.5591 w/ correlation 64.8 %
29	x detected	2000	50-pattern	3705	0.0255	y = 0.9989 (x) ^ 1.0775 w/ correlation 100.0 %	y = 0.8967 (x) ^ 0.5591 w/ correlation 64.8 %
30	x detected	3000	50-pattern	5834	0.0170	y = 0.9989 (x) ^ 1.0775 w/ correlation 100.0 %	y = 0.8967 (x) ^ 0.5591 w/ correlation 64.8 %
31	y detected	100	3-pattern	128	0.0400	y = 0.9914 (x) ^ 1.0557 w/ correlation 99.9 %	y = 1.001 (x) ^ 0.1857 w/ correlation 99.9 %
32	y detected	500	3-pattern	664	0.0080	y = 0.9914 (x) ^ 1.0557 w/ correlation 99.9 %	y = 1.001 (x) ^ 0.1857 w/ correlation 99.9 %
33	y detected	1000	3-pattern	1911	0.0040	y = 0.9914 (x) ^ 1.0557 w/ correlation 99.9 %	y = 1.001 (x) ^ 0.1857 w/ correlation 99.9 %
34	y detected	1500	3-pattern	1781	0.0027	y = 0.9914 (x) ^ 1.0557 w/ correlation 99.9 %	y = 1.001 (x) ^ 0.1857 w/ correlation 99.9 %
35	y detected	2000	3-pattern	3769	0.0020	y = 0.9914 (x) ^ 1.0557 w/ correlation 99.9 %	y = 1.001 (x) ^ 0.1857 w/ correlation 99.9 %
36	y detected	3000	3-pattern	5986	0.0013	y = 0.9914 (x) ^ 1.0557 w/ correlation 99.9 %	y = 1.001 (x) ^ 0.1857 w/ correlation 99.9 %
37	y detected	100	5-pattern	160	0.0600	y = 0.998 (x) ^ 1.081 w/ correlation 100.0 %	y = 1.0013 (x) ^ 0.24 w/ correlation 99.9 %
38	y detected	500	5-pattern	971	0.0120	y = 0.998 (x) ^ 1.081 w/ correlation 100.0 %	y = 1.0013 (x) ^ 0.24 w/ correlation 99.9 %
39	y detected	1000	5-pattern	1994	0.0060	y = 0.998 (x) ^ 1.081 w/ correlation 100.0 %	y = 1.0013 (x) ^ 0.24 w/ correlation 99.9 %
40	y detected	1500	5-pattern	2564	0.0040	y = 0.998 (x) ^ 1.081 w/ correlation 100.0 %	y = 1.0013 (x) ^ 0.24 w/ correlation 99.9 %
41	y detected	2000	5-pattern	3891	0.0030	y = 0.998 (x) ^ 1.081 w/ correlation 100.0 %	y = 1.0013 (x) ^ 0.24 w/ correlation 99.9 %
42	y detected	3000	5-pattern	5539	0.0020	y = 0.998 (x) ^ 1.081 w/ correlation 100.0 %	y = 1.0013 (x) ^ 0.24 w/ correlation 99.9 %
43	y detected	100	10-pattern	180	0.1100	y = 0.9994 (x) ^ 1.0822 w/ correlation 100.0 %	y = 0.9199 (x) ^ 0.6408 w/ correlation 78.7 %
44	y detected	500	10-pattern	784	0.0220	y = 0.9994 (x) ^ 1.0822 w/ correlation 100.0 %	y = 0.9199 (x) ^ 0.6408 w/ correlation 78.7 %
45	y detected	1000	10-pattern	1820	0.0110	y = 0.9994 (x) ^ 1.0822 w/ correlation 100.0 %	y = 0.9199 (x) ^ 0.6408 w/ correlation 78.7 %
46	y detected	1500	10-pattern	2694	0.0073	y = 0.9994 (x) ^ 1.0822 w/ correlation 100.0 %	y = 0.9199 (x) ^ 0.6408 w/ correlation 78.7 %
47	y detected	2000	10-pattern	3786	0.0055	y = 0.9994 (x) ^ 1.0822 w/ correlation 100.0 %	y = 0.9199 (x) ^ 0.6408 w/ correlation 78.7 %
48	y detected	3000	10-pattern	3798	0.0037	y = 0.9994 (x) ^ 1.0822 w/ correlation 100.0 %	y = 0.9199 (x) ^ 0.6408 w/ correlation 78.7 %
49	y detected	100	20-pattern	167	0.2100	y = 1.0025 (x) ^ 1.0607 w/ correlation 100.0 %	y = 1.0023 (x) ^ 0.4078 w/ correlation 99.9 %
50	y detected	500	20-pattern	979	0.0420	y = 1.0025 (x) ^ 1.0607 w/ correlation 100.0 %	y = 1.0023 (x) ^ 0.4078 w/ correlation 99.9 %

51	y detected	1000	20-pattern	1994	0.0210	y = 1.0025 (x) ^ 1.0607 w/ correlation 100.0 %	y = 1.0023 (x) ^ 0.4078 w/ correlation 99.9 %
52	y detected	1500	20-pattern	2505	0.0140	y = 1.0025 (x) ^ 1.0607 w/ correlation 100.0 %	y = 1.0023 (x) ^ 0.4078 w/ correlation 99.9 %
53	y detected	2000	20-pattern	2984	0.0105	y = 1.0025 (x) ^ 1.0607 w/ correlation 100.0 %	y = 1.0023 (x) ^ 0.4078 w/ correlation 99.9 %
54	y detected	3000	20-pattern	4995	0.0070	y = 1.0025 (x) ^ 1.0607 w/ correlation 100.0 %	y = 1.0023 (x) ^ 0.4078 w/ correlation 99.9 %
55	y detected	100	50-pattern	129	0.5100	y = 1.0049 (x) ^ 1.0714 w/ correlation 100.0 %	y = 0.9482 (x) ^ 0.7372 w/ correlation 91.7 %
56	y detected	500	50-pattern	899	0.1020	y = 1.0049 (x) ^ 1.0714 w/ correlation 100.0 %	y = 0.9482 (x) ^ 0.7372 w/ correlation 91.7 %
57	y detected	1000	50-pattern	1981	0.0510	y = 1.0049 (x) ^ 1.0714 w/ correlation 100.0 %	y = 0.9482 (x) ^ 0.7372 w/ correlation 91.7 %
58	y detected	1500	50-pattern	2890	0.0340	y = 1.0049 (x) ^ 1.0714 w/ correlation 100.0 %	y = 0.9482 (x) ^ 0.7372 w/ correlation 91.7 %
59	y detected	2000	50-pattern	3057	0.0255	y = 1.0049 (x) ^ 1.0714 w/ correlation 100.0 %	y = 0.9482 (x) ^ 0.7372 w/ correlation 91.7 %
60	y detected	3000	50-pattern	5942	0.0170	y = 1.0049 (x) ^ 1.0714 w/ correlation 100.0 %	y = 0.9482 (x) ^ 0.7372 w/ correlation 91.7 %
61	x and y detected	100	3-pattern	169	0.0600	y = 0.9964 (x) ^ 1.0369 w/ correlation 100.0 %	y = 1.0013 (x) ^ 0.24 w/ correlation 99.9 %
62	x and y detected	500	3-pattern	927	0.0120	y = 0.9964 (x) ^ 1.0369 w/ correlation 100.0 %	y = 1.0013 (x) ^ 0.24 w/ correlation 99.9 %
63	x and y detected	1000	3-pattern	1855	0.0060	y = 0.9964 (x) ^ 1.0369 w/ correlation 100.0 %	y = 1.0013 (x) ^ 0.24 w/ correlation 99.9 %
64	x and y detected	1500	3-pattern	1782	0.0040	y = 0.9964 (x) ^ 1.0369 w/ correlation 100.0 %	y = 1.0013 (x) ^ 0.24 w/ correlation 99.9 %
65	x and y detected	2000	3-pattern	2888	0.0030	y = 0.9964 (x) ^ 1.0369 w/ correlation 100.0 %	y = 1.0013 (x) ^ 0.24 w/ correlation 99.9 %
66	x and y detected	3000	3-pattern	3550	0.0020	y = 0.9964 (x) ^ 1.0369 w/ correlation 100.0 %	y = 1.0013 (x) ^ 0.24 w/ correlation 99.9 %
67	x and y detected	100	5-pattern	193	0.1000	y = 1.0019 (x) ^ 1.0857 w/ correlation 100.0 %	y = 1.0017 (x) ^ 0.3084 w/ correlation 99.9 %
68	x and y detected	500	5-pattern	716	0.0200	y = 1.0019 (x) ^ 1.0857 w/ correlation 100.0 %	y = 1.0017 (x) ^ 0.3084 w/ correlation 99.9 %
69	x and y detected	1000	5-pattern	1886	0.0100	y = 1.0019 (x) ^ 1.0857 w/ correlation 100.0 %	y = 1.0017 (x) ^ 0.3084 w/ correlation 99.9 %
70	x and y detected	1500	5-pattern	2954	0.0067	y = 1.0019 (x) ^ 1.0857 w/ correlation 100.0 %	y = 1.0017 (x) ^ 0.3084 w/ correlation 99.9 %
71	x and y detected	2000	5-pattern	3664	0.0050	y = 1.0019 (x) ^ 1.0857 w/ correlation 100.0 %	y = 1.0017 (x) ^ 0.3084 w/ correlation 99.9 %
72	x and y detected	3000	5-pattern	5941	0.0033	y = 1.0019 (x) ^ 1.0857 w/ correlation 100.0 %	y = 1.0017 (x) ^ 0.3084 w/ correlation 99.9 %
73	x and y detected	100	10-pattern	157	0.2000	y = 0.9989 (x) ^ 1.0665 w/ correlation 100.0 %	y = 0.9804 (x) ^ 0.4837 w/ correlation 97.1 %
74	x and y detected	500	10-pattern	656	0.0400	y = 0.9989 (x) ^ 1.0665 w/ correlation 100.0 %	y = 0.9804 (x) ^ 0.4837 w/ correlation 97.1 %
75	x and y detected	1000	10-pattern	1983	0.0200	y = 0.9989 (x) ^ 1.0665 w/ correlation 100.0 %	y = 0.9804 (x) ^ 0.4837 w/ correlation 97.1 %
76	x and y detected	1500	10-pattern	2367	0.0133	y = 0.9989 (x) ^ 1.0665 w/ correlation 100.0 %	y = 0.9804 (x) ^ 0.4837 w/ correlation 97.1 %
77	x and y detected	2000	10-pattern	3407	0.0100	y = 0.9989 (x) ^ 1.0665 w/ correlation 100.0 %	y = 0.9804 (x) ^ 0.4837 w/ correlation 97.1 %
78	x and y detected	3000	10-pattern	5673	0.0067	y = 0.9989 (x) ^ 1.0665 w/ correlation 100.0 %	y = 0.9804 (x) ^ 0.4837 w/ correlation 97.1 %
79	x and y detected	100	20-pattern	134	0.4000	y = 0.9994 (x) ^ 1.0793 w/ correlation 100.0 %	y = 1.0129 (x) ^ 0.8791 w/ correlation 99.6 %
80	x and y detected	500	20-pattern	905	0.0800	y = 0.9994 (x) ^ 1.0793 w/ correlation 100.0 %	y = 1.0129 (x) ^ 0.8791 w/ correlation 99.6 %
81	x and y detected	1000	20-pattern	1459	0.0400	y = 0.9994 (x) ^ 1.0793 w/ correlation 100.0 %	y = 1.0129 (x) ^ 0.8791 w/ correlation 99.6 %
82	x and y detected	1500	20-pattern	2638	0.0267	y = 0.9994 (x) ^ 1.0793 w/ correlation 100.0 %	y = 1.0129 (x) ^ 0.8791 w/ correlation 99.6 %
83	x and y detected	2000	20-pattern	3703	0.0200	y = 0.9994 (x) ^ 1.0793 w/ correlation 100.0 %	y = 1.0129 (x) ^ 0.8791 w/ correlation 99.6 %
84	x and y detected	3000	20-pattern	4048	0.0133	y = 0.9994 (x) ^ 1.0793 w/ correlation 100.0 %	y = 1.0129 (x) ^ 0.8791 w/ correlation 99.6 %
85	x and y detected	500	50-pattern	868	0.2000	y = 1.0057 (x) ^ 1.0599 w/ correlation 99.9 %	y = 1.0033 (x) ^ 0.9451 w/ correlation 100.0 %
86	x and y detected	1000	50-pattern	1881	0.1000	y = 1.0057 (x) ^ 1.0599 w/ correlation 99.9 %	y = 1.0033 (x) ^ 0.9451 w/ correlation 100.0 %
87	x and y detected	1500	50-pattern	2719	0.0667	y = 1.0057 (x) ^ 1.0599 w/ correlation 99.9 %	y = 1.0033 (x) ^ 0.9451 w/ correlation 100.0 %
88	x and y detected	2000	50-pattern	2741	0.0500	y = 1.0057 (x) ^ 1.0599 w/ correlation 99.9 %	y = 1.0033 (x) ^ 0.9451 w/ correlation 100.0 %
89	x and y detected	3000	50-pattern	5909	0.0333	y = 1.0057 (x) ^ 1.0599 w/ correlation 99.9 %	y = 1.0033 (x) ^ 0.9451 w/ correlation 100.0 %