Efficiency of some algorithms to compute fractal total variation

Hanwen Li, Tom Elbaz, Stéphane Junca

Efficiency of some algorithms to compute fractal total variation

Tom Elbaz [*], Hanwen Li [†], Stéphane Junca [‡]

March 11, 2023

1 Introduction

The word "fractal" was invented in 1974 by Benoît Mandelbrot who is a Franco-American mathematician. One of the best known fractals is the Koch flake. If we look for its etymology, "fractal" means irregular. Therefore, in mathematics, a curve or a surface having an irregular shape is called a fractal. So even if we zoom in on this shape, it may still have irregularities. We can also notice that the fractal aspect is present in everyday life: we can cite for example the coast of Saint-Jean Cap Ferrat or the mountains of the Mercantour. But these examples are far from being the only ones: we are surrounded by fractals! The goal of this project is to measure a "fractal total variation", rather a "fractal unevenness" of the graph of a function. We define the total p-variation for a function u : I $\to \mathbb{R}$ by:

$$p - TV[u] = \sup_{\sigma \in Sub(I)} \sum_{i=1}^{|\sigma|-1} \mid u(x_i) - u(x_{i-1})|^p$$

with $p \geq 1$ and where Sub designates the set of all the subdivisions of I , $|\sigma|$ is the cardinal of σ.

We recall that on a fixed interval I=[a,b], a subdivision $\sigma = \{x_0, x_1, ..., x_N\}$ of this interval is an ordered finite sequence such that $a \leq x_0 < x_1 < .... < x_N \leq b$. If the subdivision is regular, then the distance between each x_N stays the same.

Fractal unevenness is: $D_s u = (TV[u])^{\frac{1}{p}}$ whereabouts $p = \frac{1}{s}$ and s is the fractal dimension $(0 < s \leq 1)$.We can thus write $D_s u = (TV[u])^s$
Culturally, $D_s u$ is also a semi-norm used for stochastic processes and also for hyperbolic conservation laws [CJJ],[DJ]. The fractal total variation usually called p-variation $p > 1$ is used in probability [NorRa], in the financial applications [NorSa] and in the functional analysis [DN].
Recently, the fractional BV spaces have been introduced for partial differential equations

[*]Université côte d'Azur
[†]Université côte d'Azur, Polytech'Nice
[‡]Université côte d'Azur, Inria & CNRS, LJAD

[BG],[CJJ]. In this subject, there are more and more applications and researchs using fractional BV spaces for hyperbolic conservation laws [GGJJ],[JR],[M].
The previous results are essentialy theoritical. In this project, we want to compute the fractal total variation efficiently. We begin by introducing the case where p=1 which corresponds to the non-fractal case.
The total 1-variation of a function is related to the derivative of the function. Indeed, on an interval [a,b], We have 1-TV[u] $= \int_a^b |\ u'(x)|\ dx$ only if $u'(x) \in L^1$.

For $p = 1$ we note that the total variation is equal to the unevenness. This quantity is particularly well known to hikers or mountaineers... Indeed, this is used to calculate the total difference in level (ascent + descent).

In general, when $p > 1$, the problem is much more complex. Indeed, we can cite the case of the sequence: $u = (u_0, u_1, u_2, u_3) = (1, 4, 3, 7)$. We have for the case $p = 2$ for example, we calculate $|u_3 - u_2|^2 + |u_2 - u_1|^2 + |u_1 - u_0|^2 = 26$, $|u_3 - u_2|^2 + |u_2 - u_0|^2 = 20$, $|u_3 - u_1|^2 + |u_1 - u_0|^2 = 18$ and $|u_3 - u_0|^2 = 36$. In this case we keep only u_0 and u_3. This is why we cite the following theorem from the article [BN] which brings us back to the discrete case.

Theorem 1 *If f is a piecewise monotone function i.e. f is monotone on every subinterval of the subdivision with* $I = [a, b] = \cup_{i=0}^{n} I_i$ *and* $I_i = [a_{i-1}, a_i]$*, then for the piecewise constant function u such that* $u(x) = f(a_i)$ *on* $[a_{i-1/2}, a_{i+1/2}[$*, we have p-TV(f) = p-TV(u)*

To start, we will therefore focus in a sequence of points $(u_0, u_1, \ldots.., u_N)$ containing N+1 points. As we are in the discrete case, a subdivision of $\{0, 1, \ldots, N\}$ is a finite ordered subset and we denote $\sigma = \{\sigma_0, \sigma_1, \ldots, \sigma_m\}$ with $\sigma_0, \ldots, \sigma_N \in \mathbb{N}$, $0 \leq \sigma_k \leq m$, $0 \leq k \leq m \leq N$, and $|\sigma| = m + 1$ is the number of points of the subdivision. We can write the p-variation of the sequence as: $\sup_{\sigma \in Sub(I)} \sum_{i=1}^{|\sigma|-1} |u(\sigma_i) - u(\sigma_{i-1}|^p$ where the supremum is computed on all subdivisions.

The objective is to set up algorithms to calculate the p-variation. We then want to estimate this p-variation on each subdivision and maximize it: we will then take the maximum on all subdivisions. One of missions is to find the most suitable algorithm to answer to the problem posed. We are going to compare 4 algorithms and study their execution time. Finally, it will be necessary to compare the efficiency of these algorithms. We will use the following algorithms: the Exhaustive algorithm, the Eraser algorithm, the Add One Point algorithm and the Merge algorithm which will be introduced later. We will also introduce another algorithm: this is the Merge Eraser Initialization algorithm. The goal will therefore be reduce this complexity by storing some variables in Memory.

2 Exhaustive Algorithm

We start by trying to code an algorithm allowing to calculate all the subdivisions and to estimate the maximum on all the subdivisions of the sum of the variations of u to the power p (the total p-variation). We first have the following intermediate program:

```
def powerset(u):
  A=[[]]
  for x in u:
    for y in A[:]:
      z=y[:]
      z.append(x)
      A.append(z)
  return A
```

In the "powerset" program, we read all the subsets of the sequence of points considered (there are in total 2^{n+1} where n+1 is the number of points). If we take for example a sequence u such that $u = (u_0, u_1)$, then powerset will return $[[], [u_0], [u_1], [u_0, u_1]]$.

The code to implement the exhaustive algorithm is as follows :

```
#time_start=time.clock()
##########
def exhaustive(u,p):
    A1=[]
    A2=[]
    A3=[]
    prétraitement
    for x in powerset(indice(u)):
        if (len(x)>1):
            a = []
            for y in x:
                a.append(u[y])
                A1.append(a)
    for t in A1:
        A2.append(listofdelta(t,p))
    for z in A2:
        A3.append(sum(z))
    return max(A3)
##########
#time_end=time.clock()
#time_sum=time_end-time_start
#print(time_sum)
```

By performing several tests to estimate the computation time, we confirm that this algorithm is expensive, that is to say that its complexity is too great. Indeed, we see that if the sequence has more than twenty terms, it becomes very long for the machine to give a result. CPU time is the computer clock. The time measurement has some fluctuations due to the background running of the laptop. It is therefore a time which is badly measured and which is influenced by the other tasks that the computer is in the process of solving.

As we perform the tests on each subdivision, the complexity is of the order of $O(2^n)$ i.e. the number of parts of a set with n elements. This means that if we add a term to a considered sequence and launch the calculation, the execution time will be multiplied by 2.

This is why we carry out a few pre-processing in order to be able to reduce the calculation time and reduce the complexity. We begin by carrying out the following preprocessing:

We test the monotony of the considered sequence: if $(u_i - u_{i-1})(u_{i+1} - u_i) > 0$, then the sequence is monotone.
We will use the following inequality for preprocessing:

$$a^p + b^p < (a+b)^p, \qquad 0 < a,\ 0 < b, \quad 1 < p. \tag{1}$$

Proof of the inequality (1): The strict convexity is the tool [JR].

$$\begin{aligned} a^p + b^p &= \left(a\left(\frac{a+b}{a+b}\right)\right)^p + \left(b\left(\frac{a+b}{a+b}\right)\right)^p \\ &= \left(\frac{a}{a+b}(a+b)\right)^p + \left(\frac{b}{a+b}(a+b)\right)^p \\ &< \frac{a}{a+b}(a+b)^p + \frac{b}{a+b}(a+b)^p \\ &= (a+b)^p. \end{aligned}$$

Which ends the proof.

We can apply the convex inequality (1) on this sum to estimate the p-variation:

$$\sum_{i=1}^{n} |\ u(x_i) - u(x_{i-1})|^p = |u_1 - u_0|^p + |u_2 - u_1|^p + \ldots \\ \ldots + |u_i - u_{i-1}|^p + |u_{i+1} - u_i|^p + \ldots + |u_N - u_{N-1}|^p.$$

If $u_{i-1} < u_i < u_{i+1}$ then we set $a = u_i - u_{i-1} > 0$, $b = u_{i+1} - u_i > 0$ and, by virtue of the convex inequality (1) we have $(u_i - u_{i-1})^p + (u_{i+1} - u_i)^p \leq (u_i - u_{i-1} + u_{i+1} - u_i)^p$ that is to say $(u_i - u_{i-1})^p + (u_{i+1} - u_i)^p \leq (u_{i+1} - u_{i-1})^p$ and we can then delete u_i to get a subsequence with the bigger variation than the initial sequence. This first preprocessing already makes it possible to reduce the number of terms of the sequence. We notice that the algorithm to compute the p-variation then runs much faster.

```
def prétraitement(u):
    for i in range(1,len(u)-1):
        if (u[i]-u[i-1])*(u[i+1]-u[i])>=0:
            del u[i]
    return u
```

Note: if the sequence is strictly monotonic, then only the first and last terms remain.
generalized convexity inequality: $a_1^p + a_2^p + ... + a_N^p < (a_1 + a_2 + ... + a_N)^p$ if $a_1, a_2, ..., a_N > 0$ and $p > 1$.
We want to know if there is uniqueness or not of the minimal optimal subdivision. We remind that a subdivision is an ordered sequence such that $x_0 < x_1 < ..... < x_N$. The program that returns the different subdivisions given a sequence u is:

```
def subdivisions(u):
    S=[[]]
    for x in indice(u):
        for y in S[:]:
            z=y[:]
            z.append(x)
            S.append(z)
    return S
```

For a sequence $u = (u_0, u_1)$, the program return the list of the subdivisions [[0], [1], [0,1]].

We then define the program below, which given a subdivision of a sequence, allows us to calculate the values of the sequence associated with this subdivision.

```
def valeursuitesub(u,sigma):
    A=[]
    for i in sigma:
          A.append(u[i])
    return A
```

If we take for example a sequence $u = (u_0, u_1, u_2, u_3)$ and a list [0,3], the program return $[u_0, u_3]$.

We will now present a second algorithm for testing if the sum of the variations of u to the power p on a given subdivision (of size at least equal to 2) is equal to the value obtained in the exhaustive program. If this is the case, then the associated subdivision is displayed.

It is therefore in this program that we test whether the minimal optimal subdivision is unique. For the case where p=1, we can keep all the terms of the sequence for the optimal minimal subdivision. Indeed, we add positive terms (absolute values) in order to maximize the sum you have to take them all. Here is the code implementing the algorithm :

```
def exhaustive2(u,p):
    prétraitement
    pTu=exhaustive(u,p)
    print(pTu)
    A2=[]
    A1=subdivisions(u)
    for sigma in A1:
        if len(sigma)>=2:
            if pTu==sum(listofdelta(valeursuitesub(u,sigma),p)):
                A2.append(sigma)
    return A2
```

We notice that if we have a plateau defined by n points, we notice that it is necessary to remove n-1 points to have a subdivision of minimal size. In this specific case, we therefore have n minimal optimal subdivisions. This is why we are testing the program on multiple suites. You have to generate sequences randomly with the random menu and observe the results. If we obtain 2 subdivisions of minimum size associated with a given sequence, then uniqueness is excluded. After doing multiple tests to look for uniqueness (more than

10000 tests), we can see that there is no more than a minimal optimal subdivision.
We then conjecture that there is uniqueness of the minimal optimal subdivision.

3 Eraser Algorithm

This algorithm was introduced in [DJ]. The Eraser algorithm is built like this : we start from a sequence of points $(u_0, ..., u_N)$ and we begin by carrying out the tests at 3 points then at 4 points up to n-1 points. If we erase a term from the sequence, it is necessary to restart the tests from the point that we erased. Tests are performed in order, and terms are evaluated in order as well. In some cases, it is necessary to delete certain terms to obtain our optimal p-variation.
The code to implement the algorithm is as follows :

```
def eraser(u,p):
  u0=[]
  for x in u:
    u0.append(x)
  n=3
  while n<=len(u0):
    k=0
    while k<=len(u0)-n:
      if (abs(u0[k+n-1]-u0[k]))**p>=sum(listofdelta(u0[k:k+n],p)):
        del u0[k+1:k+n-1]
        u1=u0
        eraser(u1,p)
      else:
        k=k+1
    n+=1
  return u0
```

Here is the graph in log-log coordinates allowing to evaluate the complexity.

This plot is obtained by taking a sequence u having a cardinality between 3 and 100. We also perform the test 6 times on the suite and we see that the complexity is in $O(N^3)$. Indeed, by setting $y = C * N^\alpha$, we obtain $log(y) = log(C * N^\alpha) = log(C) + \alpha * log(N)$. It is therefore the equation of an affine function that will allow us to estimate the parameter α. We can then estimate $\alpha \approx 3.00$. We then perform the optimization of the algorithm: a sum is stored in memory to prevent the algorithm from recalculating all the sums. Indeed, some calculations are the same and it would therefore be useless. The calculation of the sum is optimized to gain an order in the complexity of the algorithm.

If we define the variable $S_k^{k+n} = \sum_{j=k+1}^{k+n} |u_{j-1} - u_j|^p$ then $S_{k+1}^{k+1+n} = S_k^{k+n} - |u_k - u_{k+1}|^p + |u_{k+n} - u_{k+n+1}|^p$.
The variable S_k^{k+n} is updated at each iteration and stored. This is what is done in the following program :

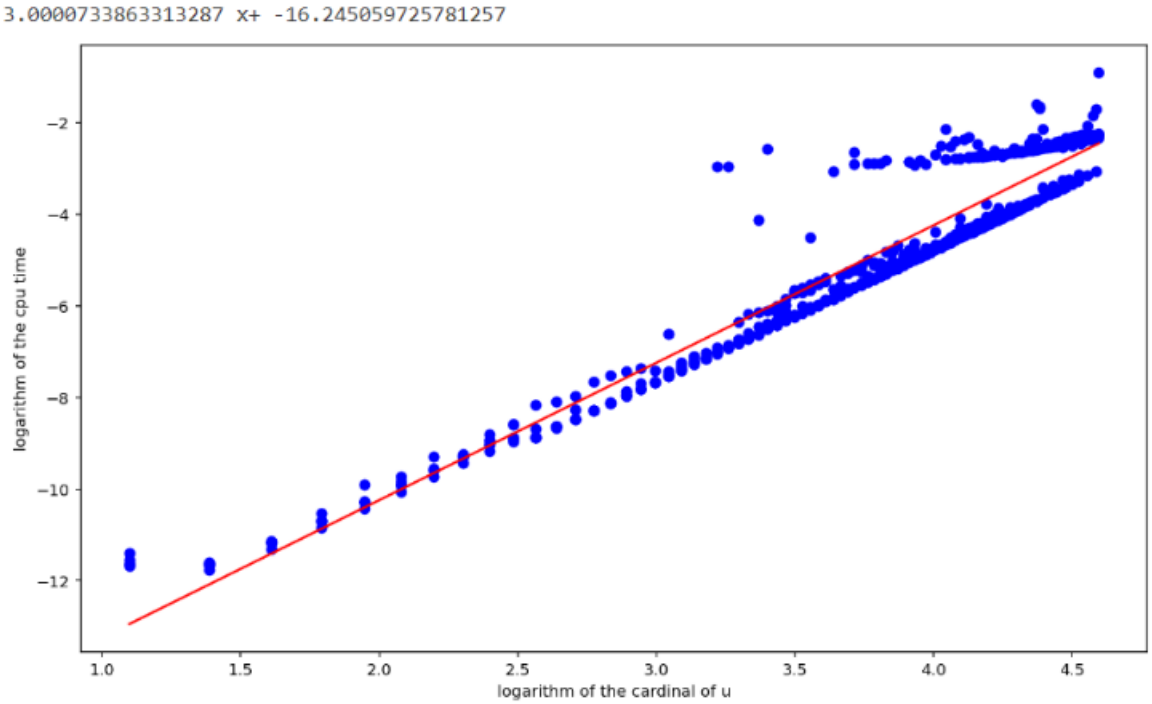

Figure 1: Complexity of the Eraser Algorithm

```
def eraser_opti(u,p):
  u0=[]
  for x in u:
    u0.append(x)
  n=3
  S0=(abs(u0[0]-u0[1]))**p
  while n<=len(u0):
    S0=S0+(abs(u0[n-1]-u0[n-2]))**p
    k=0
    S=S0
    while k <=len(u0)-n:
      if k>0:
        S=S+(abs(u0[k+n-1]-u0[k+n-2]))**p-(abs(u0[k]-u0[k-1]))**p
      else:
        S=S
      if (abs(u0[k+n-1]-u0[k]))**p>=S:
        del u0[k+1:k+n-1]
        u=u0
        eraser_opti(u,p)
      else:
        k=k+1
    n+=1

  return u0
```

We now perform the analysis of the computation time for a sequence having a cardinality varying from 3 to 100 terms.

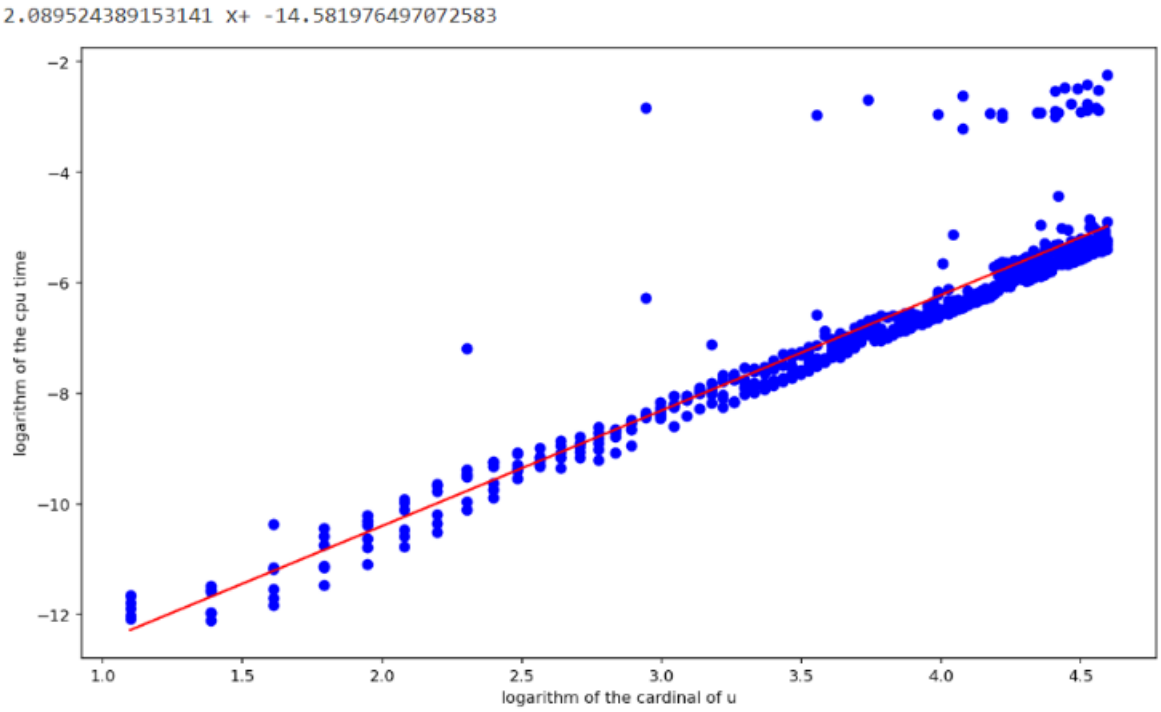

Figure 2: Complexity of the Eraser Optimized Algorithm

The complexity has decreased thanks to the optimization: we obtain a slope with a leading coefficient ≈ 2.09. This considerably reduces the calculation time and makes it possible to divide it almost by N, which is considerable. We notice that Eraser can be still improved because when we erase, we restart at the begining with tests with 3 points. We don't take account of many tests are done before the erasing. Thus this algorithm can be improved. There is an elegant solution of this problem given later by the algorithm MEI in section "Merge Eraser Initialisation".

4 Add One Point Algorithm

We study in this part the algorithm AOP (Add One Point). This one was discovered and introduced in [DJ]. It is an algorithm having the inverse process compared to the Eraser algorithm. Indeed, we start with a sequence comprising 2 terms and add the following terms by carrying out tests. The tests are always performed in the order. We will pay particular attention to the computation time and the complexity of this algorithm. Add One Point is an algorithm allowing from an optimal sequence of points $(u_0, ..., u_N)$ to construct an optimal sequence with a new term u_{N+1}.

The sequence $(u_0, ..., u_N)$ is then optimal and by adding this new term, the tests at 3 points and 4 more must be redone...and if there is an erasing you have to start over from the beginning.It is therefore a recursive algorithm.
The code to implement the algorithm is as follows :

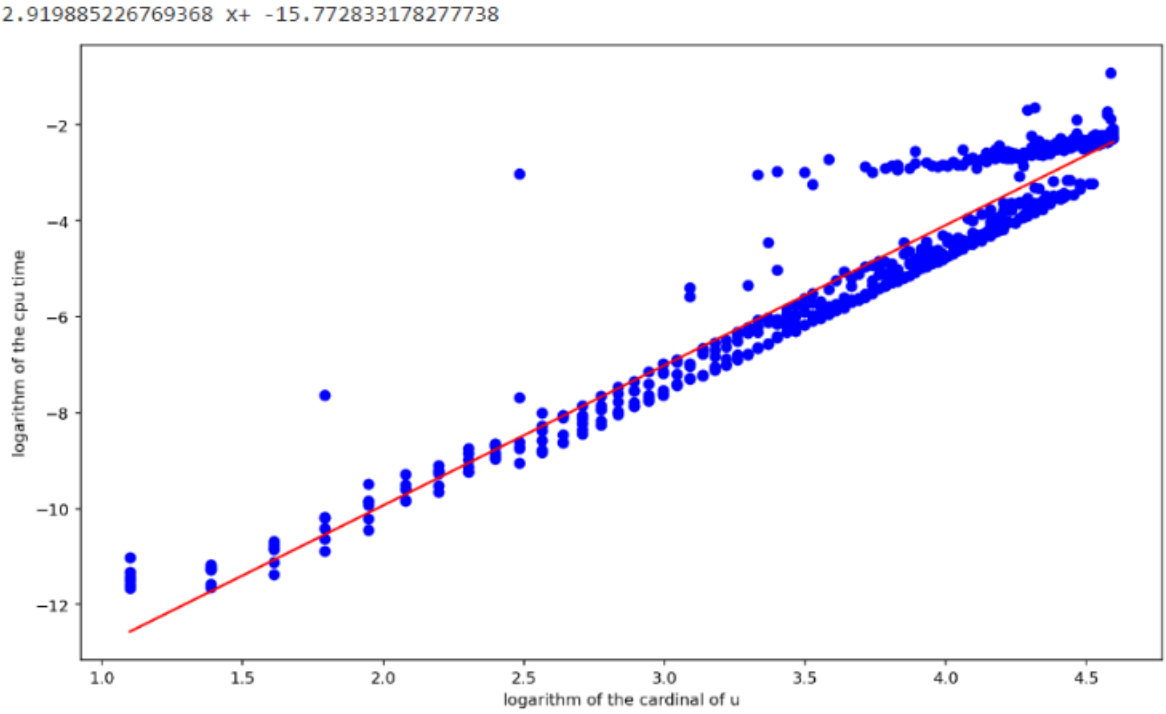

Figure 3: Complexity of the AOP algorithm

```
def AOP(u,p):
  k=2
  u_a=[u[0],u[1]]# save test points, well optimal

  while k<=len(u)-1:
    u_a.append(u[k])
    n=3
    while n<=len(u_a):
      i=len(u_a)-1
      if(abs(u_a[i-n+1]-u_a[i]))**p>=sum(listofdelta(u_a[i-n+1:i+1],p)):
        del u_a[i-n+2:i]
        n=3
      else:
        n=n+1
    k=k+1
  return u_a
```

In this algorithm, we start from a sequence comprising 2 terms u_0 and u_1 which of course form an optimal sequence because it is simply a question of evaluating $|u_1 - u_0|^p$. Each time we add a term, we do the necessary tests in order to maximize the total p-variation. We keep only the terms of the sequence which maximize this p-variation and we delete the others. We then iterate the process until we have processed all the terms of the sequence.

Here below is the graph of the complexity of the Add One Point algorithm in log-log coordinates.

Using the graph we can estimate as before $\alpha \approx 2.92$.
Our goal is now to be able to improve this algorithm: we seek to optimize it by storing variables in memory, in particular the variable sum.
The optimized algorithm obtained is the following :

```
def AOP_opti(u,p):
  k=2
  u_a=[u[0],u[1]]# save test points, well optimal

  while k<=len(u)-1:
    u_a.append(u[k])
    n=3
    l=len(u_a)-1
    S=sum(listofdelta([u_a[l-2],u_a[l-1],u_a[l]],p))
    while n<=len(u_a):
      i=len(u_a)-1
      if n>3:
        S=S+(abs(u_a[i-n+1]-u_a[i-n]))**p
      else:
        S=S
      #i=len(u_a)-1
      if(abs(u_a[i-n+1]-u_a[i]))**p>=S:
        del u_a[i-n+2:i]
        n=3
        S=sum(listofdelta([u_a[len(u_a)-3],u_a[len(u_a)-2],u_a[len(u_a)-1]],p))
      else:
        n=n+1
    k=k+1
  return u_a
```

This then reduces the complexity and reduces the computation time as we can see it below.

We observe that the directing coefficient of the line is approximately equal to 1.96. By doing many tests and averaging the results, we can surmise that α is approximately equal to 2.

This is an approximation as before but this time in a quadratic way. A least squares approach is used. We get some marginally placed points because the computation time fluctuates depending on the tasks performed by the computer. We see that we interpolate a greater number of points: this is just an interpolation but does not tell us anything about the calculation time.

5 Merge Algorithm

Merge algorithm comes from [BN] (2018) and [DJ] (2022). It is an idea based on the quick sort algorithm : "Divide and Conquer".
For the Merge algorithm, it involves merging 2 lists of sequences at the optimal origin. Let u and v denote the sequences : let $u = (u_0, ..., u_N)$ and $v = (v_0, ..., v_N)$. These 2 sequences are therefore optimal and we start by performing the 3-point tests. There are two: the test with (u_N, v_0, v_1) and the test with (u_{N-1}, u_N, v_0). You must then do the 3 tests at 4 points with $(u_N, v_0, v_1, v_2), (u_{N-1}, u_N, v_0, v_1)$ and $(u_{N-2}, u_{N-1}, u_N, v_0)$. We then continue the tests until $min(\sharp(u), \sharp(v))$ where $\sharp$ designates the cardinality (number of elements) of the sequence considered. This algorithm is a generalization of the AOP algorithm (AOP

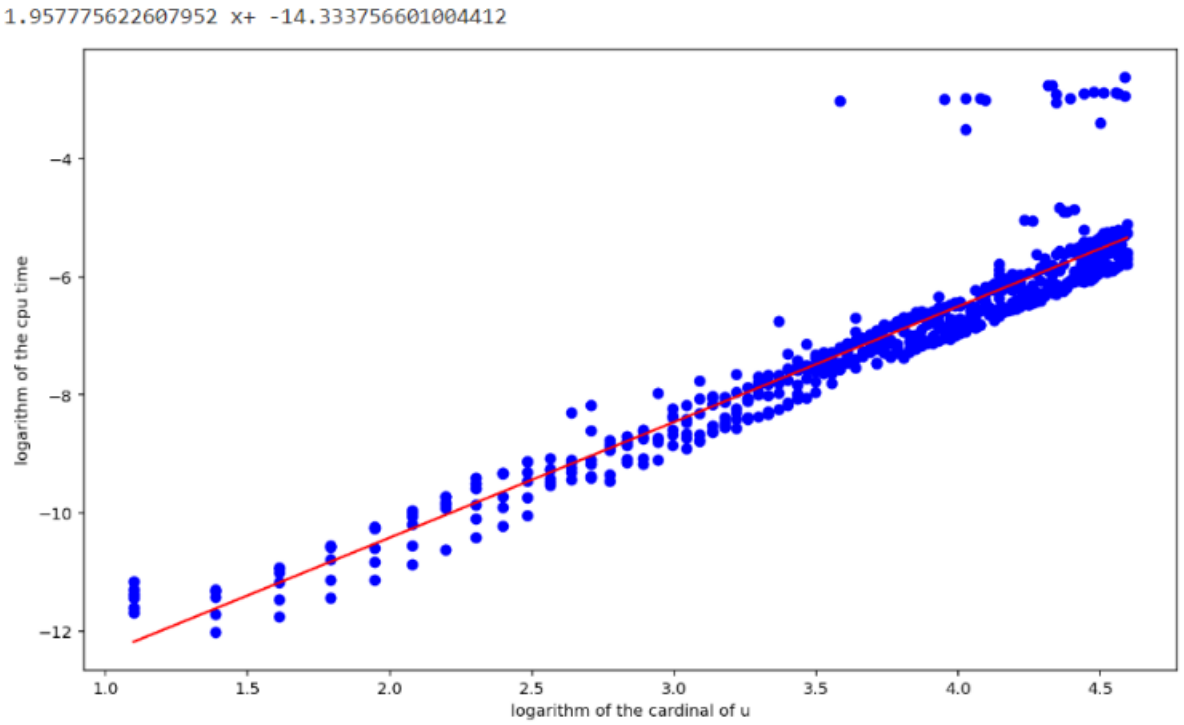

Figure 4: Complexity of the optimized AOP algorithm

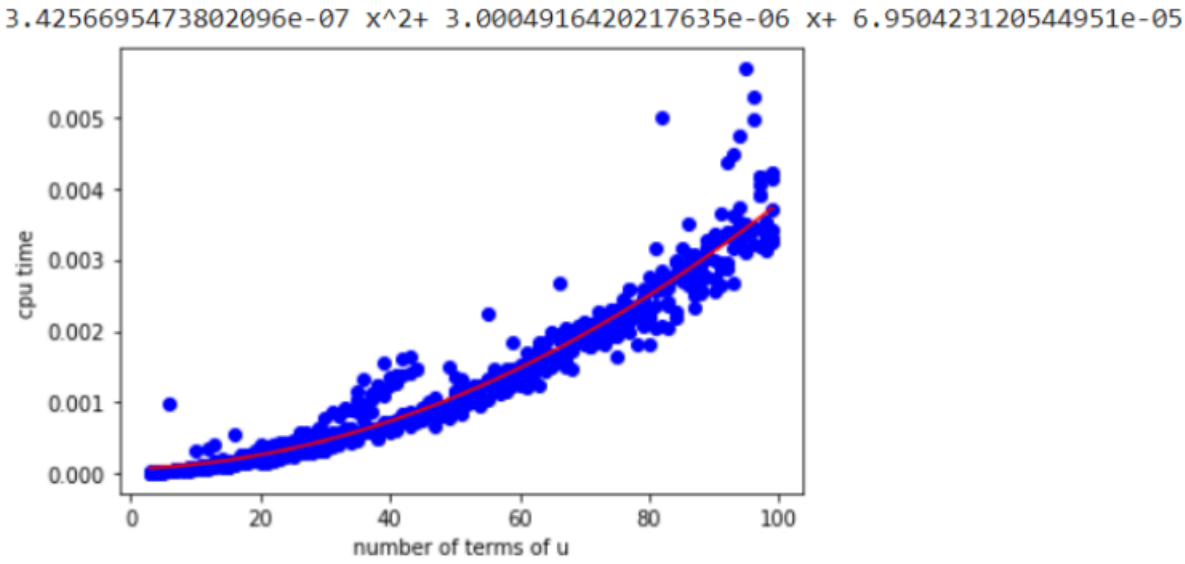

Figure 5: Quadratic interpolation of the CPU time

is a special case of Merge).
The code to implement the algorithm is as follows:

```
def twotwo(u,p):
  A=[]
  if len(u)%2==0:
    for k in range((len(u))//2):
      A.append([u[2*k],u[2*k+1]])
  else:
    for k in range((len(u)-1)//2):
      A.append([u[2*k],u[2*k+1]])
    A.append([u[len(u)-1]])
  return A
```

In this code, we test if the length of the sequence is even: indeed, if the length of the sequence is odd, there will remain a single term because we must merge two lists This program divide the whole list with lists containing only two terms.
Then, the goal of the 2nd part of the program is to merge the 2 lists. Below is the corresponding program.

```
def test2list(u1,u2,p):
  n=3
  if len(u1)+len(u2)==3:
    u_test=u1+u2
    if(abs(u_test[2]-u_test[0]))**p>=sum(listofdelta(u_test,p)):
        del u_test[1]
        return u_test
  else:
    while n<=len(u1)+len(u2)-1:
      k1=1
      while k1<n and k1<=len(u1):
        k2=n-k1
        l1=len(u1)
        l2=len(u2)
        if l1==1 and l2 != 1:
          k1=1
          k2=n-1
        if l2==1 and l1 != 1:
          k2=1
          k1=n-1
        if l1==1 and l2 ==1:
          k1=1
          k2=1
        u_test=u1[l1-k1:l1]+u2[0:k2]
      #print(u_test)
      #print(k1)
      #print(u1,u2)
        if len(u_test)>2:
          if(abs(u_test[len(u_test)-1]-u_test[0]))**p>=sum(listofdelta(u_test,p)):
            del u1[l1-k1+1:l1]
            del u2[0:k2-1]
            test2list(u1,u2,p)
          else:
            k1+=1
        else:
          return u_test
      n=n+1
  return u1+u2
```

We then write an improved version of the program with sum storage.

```
def test2list(u1,u2,p):
  n=3
  if len(u1)+len(u2)==3:
    u_test=u1+u2
    if(abs(u_test[2]-u_test[0]))**p>=sum(listofdelta(u_test,p)):
        del u_test[1]
        return u_test
  else:
    S0=(abs(u1[len(u1)-1]-u2[0]))**p
    while n<=len(u1)+len(u2)-1:
      k1=1
      S=S0
      if len(u2)>=n-1:#pour initialiser, on prend toujours le dernier element de u1
#      avec les restes dans u2, mais si la longueur de u2 n'est pas suffisant,
#       on va initialiser en ajoutant element de u1
        S=S+sum(listofdelta(u2[]))(abs(u2[n-3]-u2[n-2]))**p
      else:
        S=S+(abs(u1[len(u1)-n+1]-u1[len(u1)-n+2]))**p
      while k1<n and k1<=len(u1):
        k2=n-k1
        l1=len(u1)
        l2=len(u2)
        if k1==1:
          S=S
        else:
          S=S+(abs(u1[l1-k1]-u1[l1-k1+1]))**p-(abs(u2[k2]-u2[k2-1]))**p
          print((abs(u1[l1-k1]-u1[l1-k1+1]))**p)
          print((abs(u2[k2]-u2[k2-1]))**p)
        if l1==1 and l2 != 1:
          k1=1
          k2=n-1
        if l2==1 and l1 != 1:
          k2=1
          k1=n-1
        if l1==1 and l2 ==1:
          k1=1
          k2=1
        u_test=u1[l1-k1:l1]+u2[0:k2]
        #quand k1=2, k2=1, on annule le dernier de S_int et on ajoute celui d'avant
        print(S)
        print(sum(listofdelta(u_test,p)))
        print(u_test)
      #print(u1,u2)
        if len(u_test)>2:
          if(abs(u_test[len(u_test)-1]-u_test[0]))**p>=S:
            del u1[l1-k1+1:l1]
            del u2[0:k2-1]
            test2list(u1,u2,p)
          else:
            k1+=1
        else:
          return u_test
      n=n+1
  return u1+u2
```

We then assemble the two previous programs ("twotwo" and "Test2Lists") to form the program of the Merge algorithm.

```
def merge0(A,p):

  while len(A)>=2:
    A0=[]
    if len(A)%2==0:
      for i in range(len(A)//2):
        A0.append(test2list(A[2*i],A[2*i+1],p))

    else:
      for i in range((len(A)-1)//2):
        A0.append(test2list(A[2*i],A[2*i+1],p))
      A0.append(A[len(A)-1])
    A=A0
  return A
```

Below, we have the graph of the complexity of the Merge Algorithm

The directing coefficient of the line is approximately equal to 3.18 we can here estimate $\alpha \approx 3$.

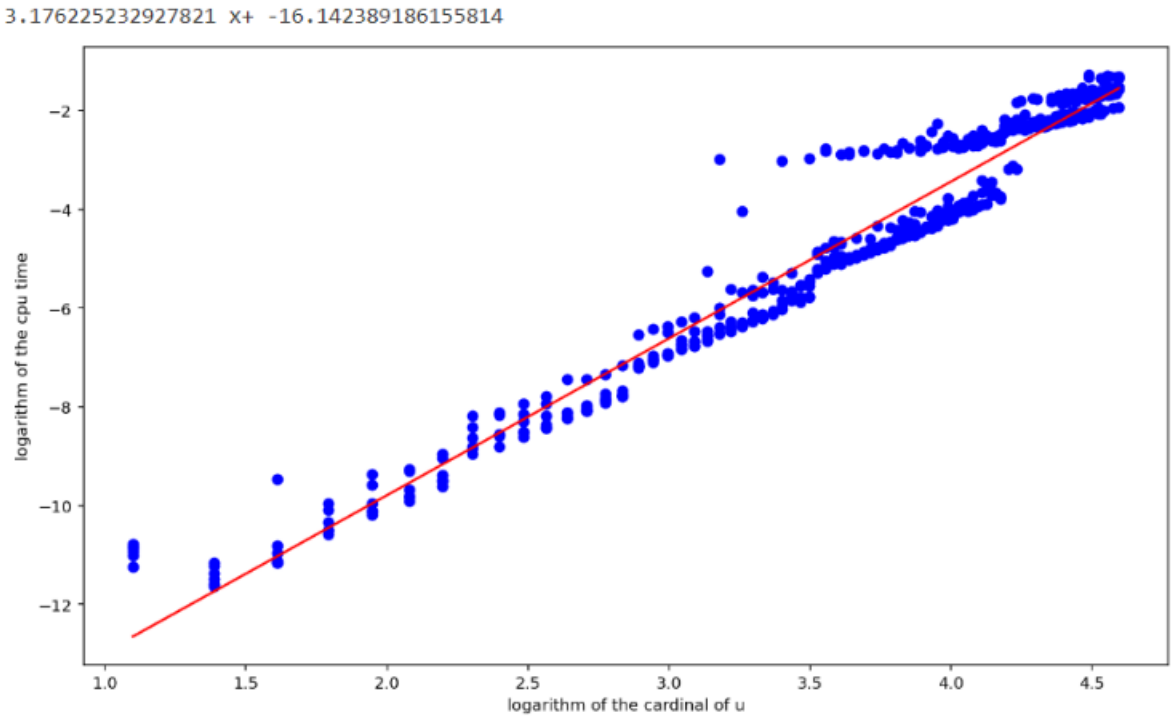

Figure 6: Complexity of the Merge algorithm

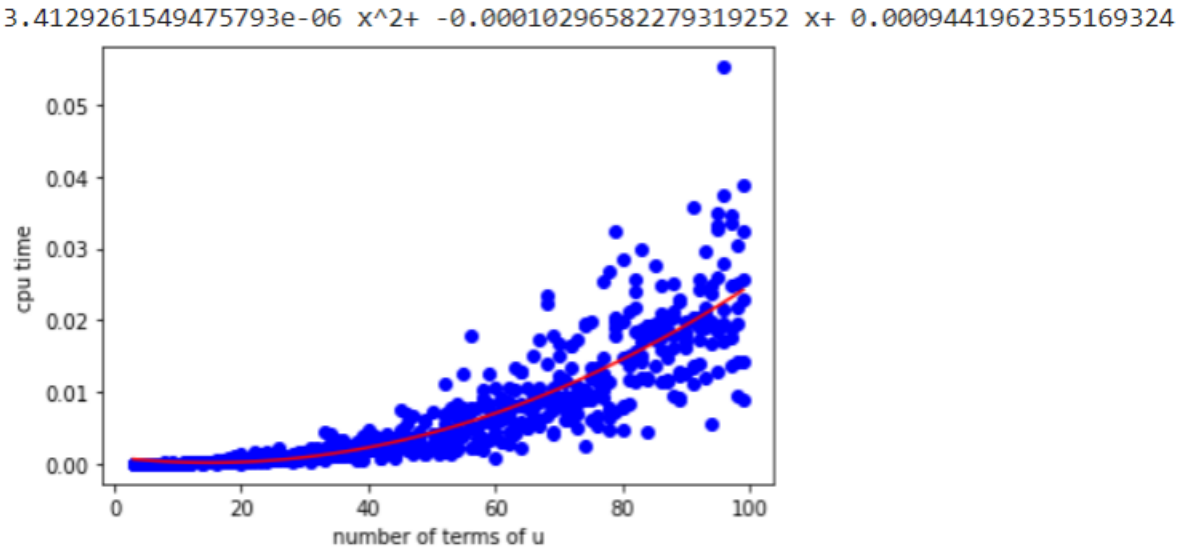

Figure 7: Quadratic interpolation of the CPU time

This last algorithm is therefore the most time-consuming: it is less suitable for solving our problem.

The choice of the optimal algorithm must therefore be made among Eraser and AOP... the goal is to know which one has the smallest constant.

6 Merge by Eraser Initialisation Algorithm

This algorithm is a prospect for future works. It improves both Eraser and Merge algorithms. It improves Merge algorithm with less merging to do and it solves the useless tests done by the Eraser algorithm.

The Eraser algorithm is not optimal because there are too many repetitions of the same calculation. Using Merge with two-point lists sums for the initialisation is worse than the AOP algorithm and also than Eraser proposed in this project. It is bad because we know that Eraser algorithm can be optimized. MEI is a way to improve both Eraser and Merge.
The principle is the following: we take a sequence $u = (u_0, ..., u_N)$ and we start with the Eraser algoritm. We use Eraser algorithm to do all the tests but after an erasing we do not restart at the begining. We obtain two sublists and we aplly the tests independently on the two sublists. So it is a recursive algorithm, cutting the initial list in many independent lists. At some points the sublists obtained are optimal. It is a question of carrying out tests on each part and after each part will be well optimized. If we do an n-point test and the length of the studied list is less than or equal to n, then the list will be well optimized.

Now, we have many sublists optimized and less points that initially. We can then use the Merge algorithm to merge the list.

This algorithm MEI is more complicated than Merge but we expect that it is more efficient than Merge.

7 Conclusions and prospects

In this project, we have programmed, simulated, compared and improved several algorithms of [BN,DJ] to calculate the p-variation: AOP, Eraser and Merge. In addition, a new MEI algorithm is proposed.

We can conclude that AOP is the simplest and the fastest algorithm used in the optimization problem to compute the p-variation. So it is the best algorithm of this project. Note that the comparisons of AOP with Eraser and Merge are done without improving the way to calculate the p-sums. Improving the computation of p-sums is easy for AOP but less so for Eraser and more complicated for Merge.

On the other hand, Eraser algorithm can be further optimized. The question of whether Eraser is more efficient than AOP remains open.

Merge is the slowest algorithm tested in the project. It is strange at the first sight since it is based on a very efficient algorithm to sort lists. Indeed, quick sort algorithm is

based on the fact that the merging of two lists with $N/2$ termes is of order N. But, for the p-variation the cost is N^3 (or N^2 if the computations of the p-sums are optimised). That means that the last merging in the algorithm Merge seems to have the same order (a little less) as AOP. Maybe that's why AOP is better than Merge. This question needs to be explored in further studies.

MEI is a very promising algorithm discovered by Hanwen Li at the end of this project. It hasn't been programmed but it looks significantly better than Merge. Indeed, it is both better than Merge and Eraser and seems to be AOP's best challenger.

Additionally, Merge and Merge Eraser Initialization (MEI) can be used on a parallel computing computer. AOP cannot be improved on a parallel computing computer. On a parallel computing computer, Merge and MEI become the best algorithms proposed in this project.

CPU time is a difficult quantity to measure in Python because it fluctuates so much. It is better to use programming languages like C++, Julia or Matlab. Below is a link that illustrates the CPU time issue with Python:

https://colab.research.google.com/drive/19KLmhggNx-wE78l4Cbj_96jyYfodnf7l?usp=sharing

A Appendix: basic programms

```
def listofdelta(u,p):
  C=[]
  for i in range(len(u)-1):
    C.append((abs(u[i+1]-u[i]))**p)
  return(C)
```

In the "listofdelta" program, we define the total p-variation by calculating the absolute value of 2 consecutive terms to the power p.

For example, for a sequence $(u_0, ..., u_N)$, the program calculates and stores $|u_1 - u_0|^p, ..., |u_N - u_{N-1}|^p$.

```
def indice(u):
  R=[]
  for i in range(len(u)):
    R.append(i)
  return R
```

The "index" program allows to assign to each term of the sequence of points an index allowing to count the number of terms (the index starts at 0). For example, for a sequence comprising 5 terms $(u_0, u_1, u_2, u_3, u_4)$, the program returns [0,1,2,3,4]. For a sequence $(u_0, ..., u_N)$ there are therefore n+1 terms in all.

```
def loglog(n,min,max,algo,p):
  i=1
  z_0=0
  z_1=0
  while i<=n:
    x,y=plottime2(min,max,algo,p)
    x1=np.log(x)
    y1=np.log(y)
    plt.xlabel('logarithm of the cardinal of u')
    plt.ylabel('logarithm of the cpu time')
    plt.scatter(x1,y1,color='b')
    z=np.polyfit(x1,y1,1)
    z_0=z_0+z[0]
    z_1=z_1+z[1]
    i=i+1

  z0=z_0/n
  z1=z_1/n
  y2=z0*x1+z1
  plt.plot(x1,y2,'r',label="Fitted line")
  plt.show
  print(z0,"x+",z1)
```

This program allows the representation in logarithmic coordinates of the complexity of the different algorithms. We choose the number of tests n to perform, the algorithm used and the minimum/maximum size of the sequence tested. The scatter plot allows the construction of the cloud of points and a line is fitted by approximation.

```
def cloudpoint(n,min,max,algo,p):
  i=1
  z_0=0
  z_1=0
  z_2=0
  while i<=n:
    x,y=plottime1(min,max,algo,p)
    z=np.polyfit(x,y,2)
    z_0=z_0+z[0]
    z_1=z_1+z[1]
    z_2=z_2+z[2]
    plt.scatter(x,y,color="b")
    plt.xlabel('number of terms of u')
    plt.ylabel('cpu time')
    i=i+1
  z0=z_0/n
  z1=z_1/n
  z2=z_2/n
  y2=z0*x**2+z1*x+z2
  plt.plot(x,y2,'r',label="Fitted line")
  plt.show
  print(z0,"x^2+",z1,"x+",z2)
```

Program allowing the approximation by a parabola based on the least squares method.

References

[BCGJ] C. Bourdarias, A. P. Choudhury, B. Guelmame and S. Junca. "Entropy solutions in BV^s for a class of triangular systems involving a transport equation". *SIAM J. Math. Anal.* 54 (2022), no. 1, 791–817.

[BGJ] C. Bourdarias, M. Gisclon and S. Junca. "Fractional BV spaces and first applications to scalar conservation laws". *Journal of Hyperbolic Differential Equations*, 2014, 11 (4), pp.655-677.

[BN] Computation of p-variation, Vygantas Buktus and Rimas Norvaisa, Lituanian Mathematical Journal, Vol. 58, No. 4, 2018, pp. 360-378.

[Br] M. Bruneau. La variation totale d'une fonction. (French). Lecture Notes in Mathematics. 413. Springer-Verlag. XIV, 332 p., 1974.

[CJJ] P. Castelli, P.-E. Jabin, S. Junca. Fractional spaces and conservation laws. Theory, numerics and applications of hyperbolic problems I, Aachen, Germany, August 2016. Springer Proceedings in Mathematics & Statistics 236, 285-293 (2018).

[DJ] Algorithms for fractional BV norm, Aimen Daoudi and Stéphane Junca, Master Thesis, Université Côte d'Azur, 49 pages, 2022

[DN] R. M. Dudley and R. Norvaiša. *Differentiability of Six Operators on Nonsmooth Functions and p-Variation*. Lecture Notes in Mathematics, 1703. Springer-Verlag, Berlin, 1999. viii+277 pp.

[GGJJ] S. S. Ghoshal, B. Guelmame, A. Jana and S. Junca. "Optimal regularity for all time for entropy solutions of conservation laws in Bv^s". *NoDEA Nonlinear Differential Equations Appl.* 27 (2020), no. 5, Paper No. 46, 30 pp.

[JR] H. Jenssen and J. Ridder. "On ϕ-variation for 1-d scalar conservation laws". *Journal of Hyperbolic Differential Equations* Vol. 17, No. 4 (2020) 843–861.

[M] E. Marconi. "Regularity estimates for scalar conservation laws in one space dimension". *J. Hyperbolic Differ. Equ.* 15 (2018), no. 4, 623–691.

[NorRa] R. Norvaiša and A. Raˇckauskas. "Convergence in law of partial sum processes in p-variation norm". *Lithuanian Math. J.* 48 (2008), no. 2, 212–227.

[NorSa] R. Norvaiša and D.M. Salopek. "Estimating the p-variation index of a sample function: An application to financial data set". *Methodol. Comput. Appl. Probab.* 4 (2002), no. 1, 27–53.

Recently the p-variation space was called fractional BV space for applications of hyperbolic PDEs such as conservation laws.

[Q] J. Qian. "The p-variation of partial sum processes and the empirical process". *Ann. Probab.* 26 (1998), no. 3, 1370–1383.