In [16]:

```
# getting the library that has some of the functions needed for simulations
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline
```

$$\mathbb{P}[Y_{n:n} := \max(Y_1, Y_2, \ldots, Y_n) \leqslant x] = F^n(x) \approx EV_\gamma\left(\frac{x - b_n}{a_n}\right),$$

which holds for large values of n, and appropriate sequences $a_n > 0$, $b_n \in \mathbb{R}$, with

$$EV_\gamma(x) = \begin{cases} \exp\{-(1+\gamma x)^{-1/\gamma}\}, & 1+\gamma x > 0 \quad \text{if } \gamma \neq 0, \\ \exp(-\exp(-x)), & x \in \mathbb{R} \quad \text{if } \gamma = 0, \end{cases}$$

In [17]:

```
#This code is for extreme value function which has pplication in simulation throughout th
e paper
def EVgamma(x, gamma, a = 1, b = 0):
  y = (x - b)/a
  if gamma == 0:
    return np.exp(-np.exp(-y))

  else:
    if (1 + gamma * y) > 0:
      return np.exp(-np.pow(1+gamma*y, -(1/gamma)))

    else:
      return np.nan
```

Nandagopalan's Estimator fot Extreamal Index we have used:

$$\theta_n^N = \theta_n^N(u) := \frac{\sum_{j=1}^{n-1} I_{[X_j > u, X_{j+1} \leqslant u]}}{\sum_{j=1}^{n} I_{[X_j > u]}} = \frac{\sum_{j=1}^{n-1} I_{[X_j \leqslant u < X_{j+1}]}}{\sum_{j=1}^{n} I_{[X_j > u]}},$$

$$\hat{\theta}_n^N(k) = \frac{1}{k} \sum_{j=1}^{n-1} I_{[X_j \leqslant X_{n-k:n} < X_{j+1}]}.$$

In [18]:

```
#Extremal index can be calculated when we are given the parameters on which it depends
def extremal_index(x, gamma, theta, a = 1, b = 0):
  a_ = a * np.pow(theta, gamma)
  b_ = b + a * ((np.pow(theta, gamma - 1) - 1)/gamma)
  return EVgamma(x, gamma, a_, b_)
```

In [19]:

```
# This part of the code is the extremal index estimator, it is basically the Non Parametr
ic extimator of the Extremal Index!
def extremal_index_non_parametric(X, u):
  """
  X: sequence
  u: threshold
  """

  numerator = 0
```

```
  denominator = 0

  for j in range(X.shape[0]):
    if (X[j] > u):
      denominator += 1

    if j != X.shape[0] - 1:
      if X[j] > u and X[j+1] <= u:
        numerator += 1

  if denominator == 0:
    return np.nan

  return numerator/denominator
```

The Mathematical form of the ARMAX process we have used:

$$X_i = \beta \max(X_{i-1}, Z_i), \quad i \geqslant 1,\ 0 < \beta < 1.$$

We have started simulating x0 with the frachet distribution then from the density function H

In [20]:

```
# In this part of the code we have utilized the Inverse Transform Method to simulate from
the ARMAX process given the Gamma, Beta parameteres in it!!

def cdf_inv_fr(u, gamma):
  return ((pow(-np.log(u) , -gamma)))

def cdf_inv_H(u , gamma, beta):
  return (pow(-np.log(u)/(pow(beta , -1/gamma)-1) , -gamma) )

def armax(beta , gamma, n, random_state = 124):
  x = np.zeros(n)
  r = np.random.RandomState(random_state)
  u = r.uniform(0,1,1)[0]
  x0 = cdf_inv_fr(u,gamma)
  xi_lag = x0
  x[0] = x0
  #print(x0)
  t = 1
  for i in range(n-1):
    r2 = np.random.RandomState(random_state + i)
    u = r2.uniform(0,1,1)[0]
    zi = cdf_inv_H(u,gamma,beta)
    xi = beta*max(xi_lag , zi)
    xi_lag = xi
    x[t] = xi
    t = t + 1
    #print(zi)
  return x
```

In [21]:

```
#Just to call the Non parametric extremal index function with another name
def theta_n_u(X, u):
  return extremal_index_non_parametric(X, u)
```

In [22]:

```
#This funcion is for the Nandagopalan's Estimator for the extremal index

def theta_n_k(X, k_=1):
  sum = 0
  k = int(k_)
```

```
  X_k = np.partition(X, n-k-1)[n-k-1]
  #if k <= 1:
  # return 1

  #X_k = max(X[n-k:n])

  for j in range(n-1):
    # k-th top order equals n-k low order
    if X[j] <= X_k and X[j+1] > X_k:
      sum += 1

  if k == 0:
    return 1

  # if sum/k >= 1:
  #   return 0.5
  return sum/k
```

The generalised jacknives estimator we have used here is of the form:

$$\hat{\theta}_n^{GJ(\delta)}(k) := \frac{(\delta^2+1)\hat{\theta}_n^N([\delta k]+1) - \delta(\hat{\theta}_n^N([\delta^2 k]+1) + \hat{\theta}_n^N(k))}{(1-\delta)^2}.$$

In [23]:

```
#This functionn is for the generalised Jacknives estimator for the extreaml Index

def theta_GJ_k(X, k, delta):
  #n = X.shape[0]

  numerator = (delta*delta + 1) * theta_n_k(X, int(np.floor(delta*k)) + 1) - delta*(thet
a_n_k(X, int(np.floor(delta*delta*k)) + 1) + theta_n_k(X, k))
  denominator = (1 - delta)**2

  return (numerator/denominator if numerator/denominator <= 1 else 1)
```

In [24]:

```
#To simulate from IID Frachet(1) distribution we have used the inverse transform method w
ith a fixed seed

def iid_fr(gamma , n , random_state = 124):
  x = np.zeros(n)
  r = np.random.RandomState(random_state)
  t = 0
  for i in range(n):
    r2 = np.random.RandomState(random_state + i)
    u = r2.uniform(0,1,1)[0]
    xi = cdf_inv_fr(u,gamma)
    x[t] = xi
    t = t+1
  return x
```

In [25]:

```
#figure 7: To demonstrate the performance of the extremal Index with the increasing Order
Statistics when data is simulated from Frachet(1) iid Density

random_start = 489
replicates = 10
runs = 50
n = 1000
all_values0 = np.zeros((runs, n))
all_values_0 = np.zeros((runs, n))
```

```
for run in range(runs):
  print("\nrun = ", run + 1, "/", runs ,"...", sep = "", end = " ")
  a = np.zeros((replicates, n))
  for i in range(replicates):
    a[i] = iid_fr(1, n, random_start + i + run+ 1 + np.random.randint(0, 10000))

  path = np.zeros((replicates, n))
  path2 = np.zeros((replicates, n))

  k_range = list(range(n))

  for k in k_range:
    if k%100 == 0:
      print(int(k/100) + 1, end = " ")
    for j in range(replicates):
      path[j][k] = theta_n_k(a[j], k)
      path2[j][k] = (path[j][k] - 1)**2

  one_run_mean = np.mean(path, axis = 0)

  all_values0[run] = np.mean(path, axis = 0)  #mean for current run
  all_values_0[run] = np.mean(path2, axis = 0)  #MSE for current run


#for generalised jacknives

random_start = 489

replicates = 10
runs = 50
n = 1000
all_values1 = np.zeros((runs, n))
all_values_1 = np.zeros((runs, n))

for run in range(runs):
  print("\nrun = ", run + 1, "/", runs ,"...", sep = "", end = " ")
  a = np.zeros((replicates, n))
  for i in range(replicates):
    a[i] = iid_fr(1, n, random_start + i + run+ 1 + np.random.randint(0, 10000))

  path = np.zeros((replicates, n))
  path2 = np.zeros((replicates, n))

  k_range = list(range(n))

  for k in k_range:
    if k%100 == 0:
      print(int(k/100) + 1, end = " ")
    for j in range(replicates):
      path[j][k] = theta_GJ_k(a[j], k, 0.25)
      path2[j][k] = (path[j][k] - 1)**2

  one_run_mean = np.mean(path, axis = 0)

  all_values1[run] = np.mean(path, axis = 0)  #mean for current run
  all_values_1[run] = np.mean(path2, axis = 0) #mean squared error for current run

fig, axs = plt.subplots(1, 2, figsize = (20,6))
fig.suptitle('Fig 1: S')
axs[0].plot(k_range, np.mean(all_values0,axis = 0), "r-")   #simple estimator
axs[0].plot(k_range, np.mean(all_values1,axis = 0), "b-")  #generalised jacknives
# axs[0].plot(k_range, straight_line, "k-")
axs[0].set_title('E[*]')
axs[0].set_xscale('log')
axs[1].plot(k_range, np.mean(all_values_0[1:], axis=0), "r-")  #simple estimator
axs[1].plot(k_range, np.mean(all_values_1[1:],axis = 0), "b-")  #generalised jacknives
# axs[1].plot(k_range[1:], straight_line[1:], "k-")
axs[1].set_title('MSE[*]')
axs[1].set_xscale('log')
axs[0].set_ylim([0,1.2])
axs[1].set_ylim([0,.02])
axs[0].set_xlim([0,1000])
```

```
axs[1].set_xlim([1,1000])
```

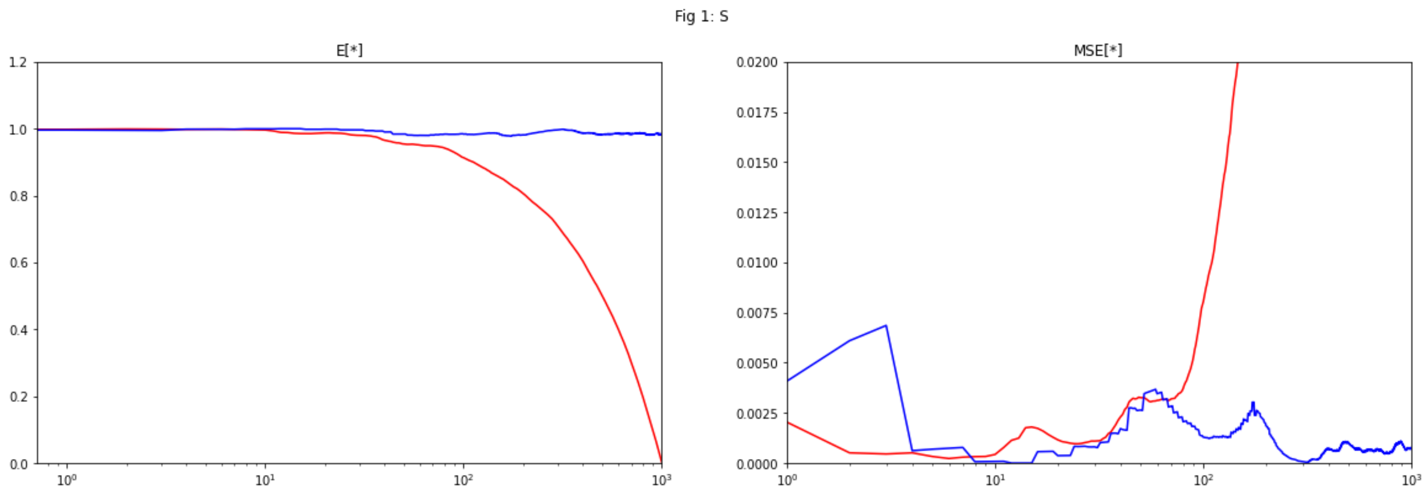
```
run = 1/50... 1 2 3 4 5 6 7 8 9 10
run = 2/50... 1 2 3 4 5 6 7 8 9 10
run = 3/50... 1 2 3 4 5 6 7 8 9 10
run = 4/50... 1 2 3 4 5 6 7 8 9 10
run = 5/50... 1 2 3 4 5 6 7 8 9 10
run = 6/50... 1 2 3 4 5 6 7 8 9 10
run = 7/50... 1 2 3 4 5 6 7 8 9 10
run = 8/50... 1 2 3 4 5 6 7 8 9 10
run = 9/50... 1 2 3 4 5 6 7 8 9 10
run = 10/50... 1 2 3 4 5 6 7 8 9 10
run = 11/50... 1 2 3 4 5 6 7 8 9 10
run = 12/50... 1 2 3 4 5 6 7 8 9 10
run = 13/50... 1 2 3 4 5 6 7 8 9 10
run = 14/50... 1 2 3 4 5 6 7 8 9 10
run = 15/50... 1 2 3 4 5 6 7 8 9 10
run = 16/50... 1 2 3 4 5 6 7 8 9 10
run = 17/50... 1 2 3 4 5 6 7 8 9 10
run = 18/50... 1 2 3 4 5 6 7 8 9 10
run = 19/50... 1 2 3 4 5 6 7 8 9 10
run = 20/50... 1 2 3 4 5 6 7 8 9 10
run = 21/50... 1 2 3 4 5 6 7 8 9 10
run = 22/50... 1 2 3 4 5 6 7 8 9 10
run = 23/50... 1 2 3 4 5 6 7 8 9 10
run = 24/50... 1 2 3 4 5 6 7 8 9 10
run = 25/50... 1 2 3 4 5 6 7 8 9 10
run = 26/50... 1 2 3 4 5 6 7 8 9 10
run = 27/50... 1 2 3 4 5 6 7 8 9 10
run = 28/50... 1 2 3 4 5 6 7 8 9 10
run = 29/50... 1 2 3 4 5 6 7 8 9 10
run = 30/50... 1 2 3 4 5 6 7 8 9 10
run = 31/50... 1 2 3 4 5 6 7 8 9 10
run = 32/50... 1 2 3 4 5 6 7 8 9 10
run = 33/50... 1 2 3 4 5 6 7 8 9 10
run = 34/50... 1 2 3 4 5 6 7 8 9 10
run = 35/50... 1 2 3 4 5 6 7 8 9 10
run = 36/50... 1 2 3 4 5 6 7 8 9 10
run = 37/50... 1 2 3 4 5 6 7 8 9 10
run = 38/50... 1 2 3 4 5 6 7 8 9 10
run = 39/50... 1 2 3 4 5 6 7 8 9 10
run = 40/50... 1 2 3 4 5 6 7 8 9 10
run = 41/50... 1 2 3 4 5 6 7 8 9 10
run = 42/50... 1 2 3 4 5 6 7 8 9 10
run = 43/50... 1 2 3 4 5 6 7 8 9 10
run = 44/50... 1 2 3 4 5 6 7 8 9 10
run = 45/50... 1 2 3 4 5 6 7 8 9 10
run = 46/50... 1 2 3 4 5 6 7 8 9 10
run = 47/50... 1 2 3 4 5 6 7 8 9 10
run = 48/50... 1 2 3 4 5 6 7 8 9 10
run = 49/50... 1 2 3 4 5 6 7 8 9 10
run = 50/50... 1 2 3 4 5 6 7 8 9 10
run = 1/50... 1 2 3 4 5 6 7 8 9 10
run = 2/50... 1 2 3 4 5 6 7 8 9 10
run = 3/50... 1 2 3 4 5 6 7 8 9 10
run = 4/50... 1 2 3 4 5 6 7 8 9 10
run = 5/50... 1 2 3 4 5 6 7 8 9 10
run = 6/50... 1 2 3 4 5 6 7 8 9 10
run = 7/50... 1 2 3 4 5 6 7 8 9 10
run = 8/50... 1 2 3 4 5 6 7 8 9 10
run = 9/50... 1 2 3 4 5 6 7 8 9 10
run = 10/50... 1 2 3 4 5 6 7 8 9 10
run = 11/50... 1 2 3 4 5 6 7 8 9 10
run = 12/50... 1 2 3 4 5 6 7 8 9 10
run = 13/50... 1 2 3 4 5 6 7 8 9 10
run = 14/50... 1 2 3 4 5 6 7 8 9 10
run = 15/50... 1 2 3 4 5 6 7 8 9 10
run = 16/50... 1 2 3 4 5 6 7 8 9 10
run = 17/50... 1 2 3 4 5 6 7 8 9 10
run = 18/50... 1 2 3 4 5 6 7 8 9 10
run = 19/50... 1 2 3 4 5 6 7 8 9 10
run = 20/50... 1 2 3 4 5 6 7 8 9 10
```

```
run = 21/50... 1 2 3 4 5 6 7 8 9 10
run = 22/50... 1 2 3 4 5 6 7 8 9 10
run = 23/50... 1 2 3 4 5 6 7 8 9 10
run = 24/50... 1 2 3 4 5 6 7 8 9 10
run = 25/50... 1 2 3 4 5 6 7 8 9 10
run = 26/50... 1 2 3 4 5 6 7 8 9 10
run = 27/50... 1 2 3 4 5 6 7 8 9 10
run = 28/50... 1 2 3 4 5 6 7 8 9 10
run = 29/50... 1 2 3 4 5 6 7 8 9 10
run = 30/50... 1 2 3 4 5 6 7 8 9 10
run = 31/50... 1 2 3 4 5 6 7 8 9 10
run = 32/50... 1 2 3 4 5 6 7 8 9 10
run = 33/50... 1 2 3 4 5 6 7 8 9 10
run = 34/50... 1 2 3 4 5 6 7 8 9 10
run = 35/50... 1 2 3 4 5 6 7 8 9 10
run = 36/50... 1 2 3 4 5 6 7 8 9 10
run = 37/50... 1 2 3 4 5 6 7 8 9 10
run = 38/50... 1 2 3 4 5 6 7 8 9 10
run = 39/50... 1 2 3 4 5 6 7 8 9 10
run = 40/50... 1 2 3 4 5 6 7 8 9 10
run = 41/50... 1 2 3 4 5 6 7 8 9 10
run = 42/50... 1 2 3 4 5 6 7 8 9 10
run = 43/50... 1 2 3 4 5 6 7 8 9 10
run = 44/50... 1 2 3 4 5 6 7 8 9 10
run = 45/50... 1 2 3 4 5 6 7 8 9 10
run = 46/50... 1 2 3 4 5 6 7 8 9 10
run = 47/50... 1 2 3 4 5 6 7 8 9 10
run = 48/50... 1 2 3 4 5 6 7 8 9 10
run = 49/50... 1 2 3 4 5 6 7 8 9 10
run = 50/50... 1 2 3 4 5 6 7 8 9 10
```

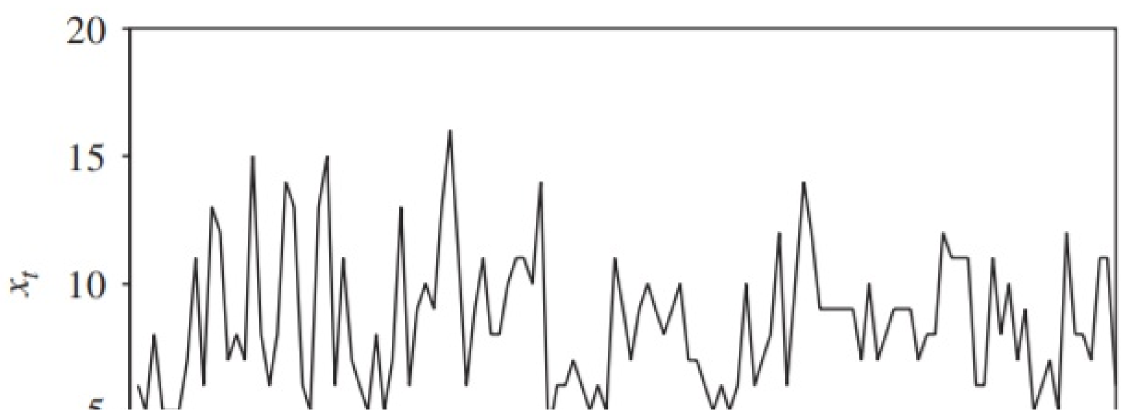
```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:81: UserWarning: Attempted t
o set non-positive left xlim on a log-scaled axis.
Invalid limit will be ignored.
```

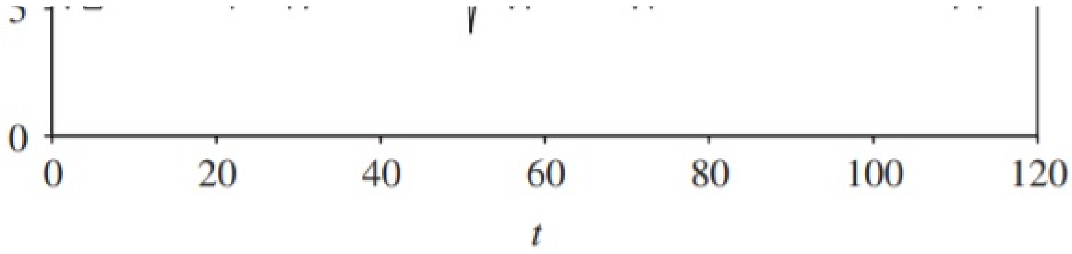
Out[25]:

```
(1, 1000)
```



Weekly maximum ozone concetraion data used in the paper:





In [26]:

```
#This is basically the ozone case part, here we have collcted the data from the
#R studio's extreme library which is not exactly the same data as the one use in
#paper but they do match to a certain extent and then using the weekly maxima
#of ozone concentration in ppm we have estimated the extreaml index for that data
#ozone

import pandas as pd
df = pd.read_csv("ozone_data.csv")

df = df['r1']

data = df.to_numpy()

import matplotlib.pyplot as plt
# plt.plot(data[:120])

#for weekly maxima data but since data not enough so taking four consecutive
#day's maxima!!
d = np.zeros(130)
t = 0
i = 0
while i < 513:
  d[t] = max(data[i:i+4])
  t = t + 1
  i = i + 4

#To plot the data we are using in place of simulations
import matplotlib.pyplot as plt
plt.plot(d[:120])


# For figure 2
n = d.shape[0]

sample = d
sample_log = np.log(sample)
# sample = (sample-np.mean(sample))/np.std(sample)


k_range = list(range(n))
straight_line = 0.7 * np.ones(n)

path1 = np.zeros(n)
path2 = np.zeros(n)
logpath1 = np.zeros(n)
logpath2 = np.zeros(n)

for k in range(n):
  if k%100 == 0:
    print(k, end = " ")
  path1[k] = theta_n_k(sample, k)
  path2[k] = theta_GJ_k(sample, k, delta = 0.25)


#To plot the extremal Index with descending order statistics
fig, axs = plt.subplots(1, 2, figsize = (20,6))
axs[0].plot(k_range, path1, "g-")
axs[0].plot(k_range, path2, "r-")
axs[0].plot(k_range, straight_line, "k-")
axs[0].set_title('Extremal index with descending o.s. associated to data')
```

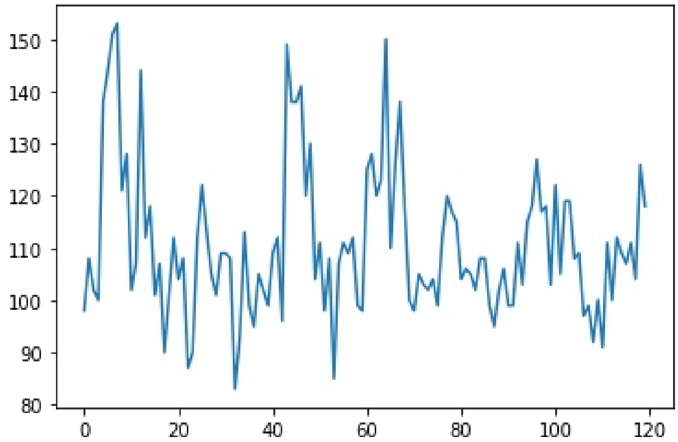
```
# axs[0].set_title('')

#To plot the extreaml index with ascending ordere statistics which is basically same as p
lotting the increasing order statistics on a log scale according to the paper!
k_range1 = np.arange(130, 0, -1)
axs[1].set_xlim([2,16])
axs[1].plot(k_range1/10, path1, "g-")
axs[1].plot(k_range1/10, path2, "r-")
axs[1].plot(k_range1/10, straight_line , 'k-')
axs[1].set_xscale('log' , basex =2 )
axs[1].set_title('Extremal index with ascending o.s. associated to data')
```

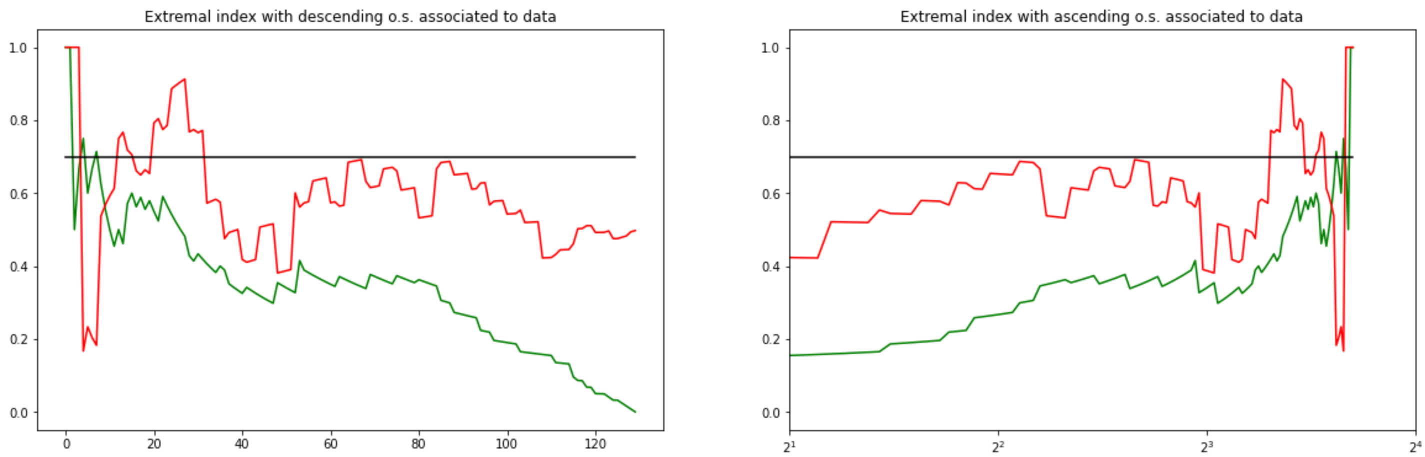
```
0 100
```

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:36: RuntimeWarning: divide b
y zero encountered in log
```

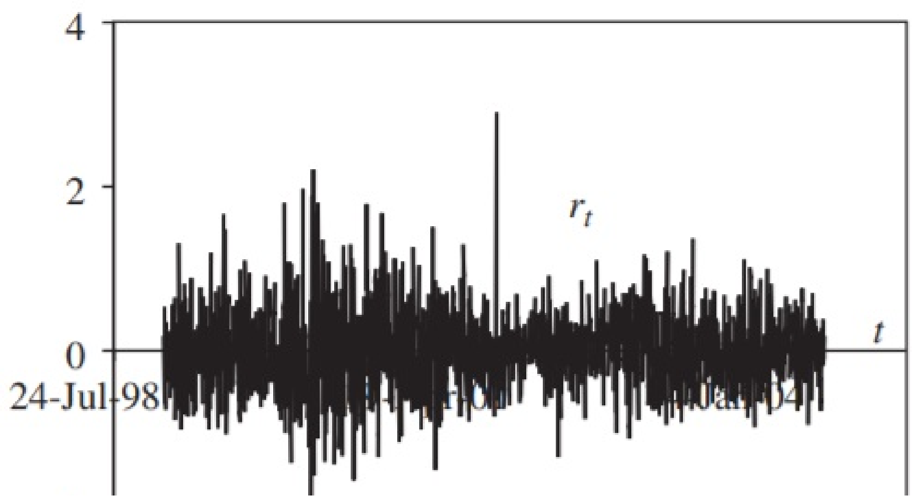
Out[26]:

```
Text(0.5, 1.0, 'Extremal index with ascending o.s. associated to data')
```





Financial Log return data used in the paper:



In [27]:

```
#This part of the code is to estimate the etremal index on the log return data of Euro-UK pound daily exchange rate. The data we have used is mostly same as the one used in paper
#To work with as the paper suggests we have taken the positive log return from the data set and then estimated the extreaml index from them
# Financial Log Returns

import pandas as pd
df = pd.read_csv("e.csv" , encoding = "ISO-8859-1")

df = df['Libra esterlina / Pound sterling']
data = df.to_numpy()

xt = data
import matplotlib.pyplot as plt
# plt.plot(xt)

rt = 100*np.log(xt[1:1551]/xt[0:1550])

#To plot the data we are using
plt.plot(rt)


sample = rt[rt>0]

n = sample.shape[0]

sample_log = np.log(sample)
# sample = (sample-np.mean(sample))/np.std(sample)


k_range = list(range(n))
straight_line = np.ones(n)

path1 = np.zeros(n)
path2 = np.zeros(n)
logpath1 = np.zeros(n)
logpath2 = np.zeros(n)

for k in range(n):
  if k%100 == 0:
    print(k, end = " ")
  path1[k] = theta_n_k(sample, k)
  path2[k] = theta_GJ_k(sample, k, delta = 0.25)

#To plot the extremal index estiamtes with descending order statistics
fig, axs = plt.subplots(1, 2, figsize = (20,6))
# axs.set_ylim([0, 1.2])
# axs.set_xlim([0,1000])
axs[0].set_ylim([.0,1.2])
axs[0].plot(k_range, path1, 'g-')
axs[0].plot(k_range, path2, 'r-')
axs[0].plot(k_range , straight_line , 'k')
axs[0].set_title('Extremal index with descending o.s. associated to data')

#To plot the extremal index estiamtes with ascending order statistics which is again equivalent to plot the desceding one on logarithmic scale so doing that:
k_range1 = np.log(np.arange(714, 0, -1))/np.log(10)
axs[1].set_ylim([0,1.2])
# axs[1].set_xlim([0,2.5])
axs[1].plot(k_range1, path1, "g-")
axs[1].plot(k_range1, path2, "r-")
```

```
axs[1].plot(k_range1, straight_line , 'k-')
# axs[1].set_xscale('linear')
axs[1].set_title('Extremal index with ascending o.s. associated to data')
```

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:21: RuntimeWarning: invalid value encountered in greater
```

```
0 100 200 300 400 500 600 700
```

Out[27]:

```
Text(0.5, 1.0, 'Extremal index with ascending o.s. associated to data')
```