Importing Libraries

In [26]:
```
# getting the library that has some of the functions needed for simulat
ions
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from numba import jit
%matplotlib inline
```

A sequence of i.i.d. r.v.'s from an underlying parent d.f.F. LetY[i:n], 1<=i<=n, be the set of associated ascending order statistics (o.s.). The tail index may be roughly defined by:

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

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 [27]:
```
# The following code is defined for the function EV-Gamma, as defined a
bove.
@jit(nopython=True)
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
```

The stationary sequence {Xn} n>=1 is said to have an extremal index (0<1)if, for all tau> 0, we may find a sequence of levels u(n) = u_n(tau) such that it follows the gamma extremal index may be informally defined by the approximations:

$$P[\max(X_1, X_2, \ldots, X_n) \leqslant x] \approx F^{n\theta}(x) \approx EV_\gamma^\theta\left(\frac{x - b_n}{a_n}\right)$$
$$= EV_\gamma\left(\frac{x - b_n'}{a_n'}\right), \quad \begin{cases} a_n' = a_n\theta^\gamma, \\ b_n' = b_n + a_n\left(\frac{\theta^\gamma - 1}{\gamma}\right). \end{cases}$$

In [28]:
```
# The following code is defined for the function EV-Gamma, as defined a
bove.
@jit(nopython=True)
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 [29]:
```
@jit(nopython=True)
def extremal_index_non_parametric(X, u):
  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
```

$$F(x) \equiv \Phi_\gamma(x) = \exp(-x^{-1/\gamma}),\ x > 0,\ \gamma > 0.$$

$$F(x) = F(x/\beta)H(x/\beta)$$

If we consider Fréchet innovations, such that $H(x) = \Phi_\gamma^{\beta^{-1/\gamma}-1}(x)$, we then get $F(x) = \Phi_\gamma(x)$, and

$$\theta = \lim_{x\to\infty} \frac{P(X_i > x, X_{i+1} \leqslant x)}{P(X_i > x)} = 1 - \lim_{x\to\infty} \frac{1 - F(x/\beta)}{1 - F(x)} = 1 - \beta^{1/\gamma}.$$

For the particular case $\gamma = 1$, considered later on for illustration, we thus get $\theta = 1 - \beta$.

In [30]:
```
@jit(nopython=True)
def cdf_inv_fr(u, gamma):
  return ((pow(-np.log(u) , -gamma)))

@jit(nopython=True)
def cdf_inv_H(u , gamma, beta):
  return (pow(-np.log(u)/(pow(beta , -1/gamma)-1) , -gamma) )
```

In [31]:
```
@jit(nopython=True)
def theta_n_u(X, u):
  return extremal_index_non_parametric(X, u)
```

Nandagopalan'sestimator is given by:

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

In [32]:
```
@jit(nopython=True)
def theta_n_k(X, k_=1):
  sum = 0
  k = int(k_)
  X_k = np.partition(X, n-k-1)[n-k-1]
  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

  return sum/k
```

An extremal index Generalized Jackknife estimator:

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

In [33]:
```
@jit(nopython=True)
def theta_GJ_k(X, k, delta):
  numerator = (delta*delta + 1) * theta_n_k(X, int(np.floor(delta*k)) +
1) - delta*(theta_n_k(X, int(np.floor(delta*delta*k)) + 1) + theta_n_k
(X, k))
  denominator = (1 - delta)**2
  if numerator < 0:
    return 0
  return numerator/denominator
```

Autoregressive processes of Order 1

$$X_j = \rho X_{j-1} + \varepsilon_j, \quad j \geqslant 1, \ X_0 \frown \text{Exponential}(1),$$

with $\{\varepsilon_j\}_{j \geqslant 1}$ standard exponential r.v.'s. For this type of sequences we have $\theta = 1$

In [34]:
```
# Defining auto-regressive process of order 1
@jit(nopython=True)
def ar(r, n):
  x = np.zeros(n) #generating a zero-array of size-n
  x[0] = np.random.exponential(size=2, scale=1)[0] #initiating with a e
xponential(1)
  for i in range(1,n,1):
    e_i = np.random.exponential(size=2, scale=1)[0] #Initiating as stan
dard exponential random variable
    x[i] = x[i-1]/r + e_i #updating the next value
  return x
```

Simulation of AR Process

for r = 10

In [35]:
```
# For figure 8.1
n = 1000
r = 10
times = 5000
path1 = np.zeros(n)
path2 = np.zeros(n)
logpath1 = np.zeros(n)
logpath2 = np.zeros(n)
for t in range(0, times, 1):
  if t%1000 == 0:
    print(t, end=" ")
  sample = np.array(ar(r, n)) #simulating the AR(1) array
  sample_log = np.log(sample) #making log of the array
  k_range = list(range(n))
  straight_line = 1 * np.ones(n) #making a straight line
  for k in k_range:
    path1[k] += theta_n_k(sample, k) # Calculating Theta_N_K
    path2[k] += theta_GJ_k(sample, k, delta = 0.25) #Calculating theta of Generalised Jackknife
    logpath1[k] += theta_n_k(sample_log, k)
    logpath2[k] += theta_GJ_k(sample_log, k, delta = 0.25)

path1 = path1/times
path2 = path2/times
logpath1 = logpath1/times
logpath2 = logpath2/times
```

```
0 1000 2000 3000 4000
```

For r = 2

In [36]:
```
# For figure 8.2
n = 1000
r = 2
times = 5000
path_1 = np.zeros(n)
path_2 = np.zeros(n)
logpath_1 = np.zeros(n)
logpath_2 = np.zeros(n)

for t in range(0, times, 1):
  if t%1000 == 0:
    print(t, end=" ")
  sample = np.array(ar(r, n))
  sample_log = np.log(sample)
  k_range = list(range(n))
  straight_line = 1 * np.ones(n)
  for k in k_range:
    path_1[k] += theta_n_k(sample, k)
    path_2[k] += theta_GJ_k(sample, k, delta = 0.25)
    logpath_1[k] += theta_n_k(sample_log, k)
    logpath_2[k] += theta_GJ_k(sample_log, k, delta = 0.25)

path_1 = path_1/times
path_2 = path_2/times
logpath_1 = logpath_1/times
logpath_2 = logpath_2/times
```

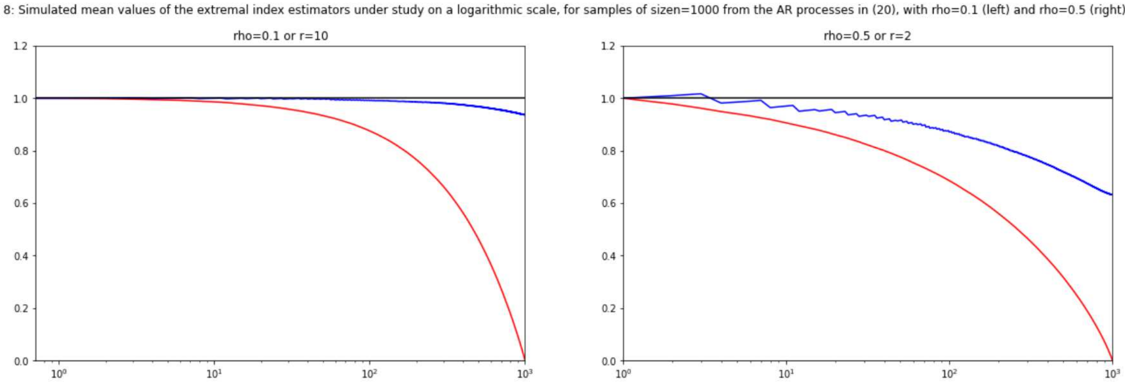
```
0 1000 2000 3000 4000
```

In [37]:
```
fig, axs = plt.subplots(1, 2, figsize = (20,6))
axs[0].set_xscale('log')

fig.suptitle('Fig 8: Simulated mean values of the extremal index estima
tors under study on a logarithmic scale, for samples of sizen=1000 from
the AR processes in (20), with rho=0.1 (left) and rho=0.5 (right).')
axs[0].plot(k_range, logpath1, "r-")
axs[0].plot(k_range, logpath2, "b-")
axs[0].plot(k_range, straight_line, "k-")
axs[0].set_title('rho=0.1 or r=10')

axs[1].plot(k_range, logpath_1, "r-")
axs[1].plot(k_range, logpath_2, "b-")
axs[1].plot(k_range, straight_line, "k-")
axs[1].set_title('rho=0.5 or r=2')
axs[1].set_xscale('log')

axs[0].set_ylim([0,1.2])
axs[1].set_ylim([0,1.2])
axs[0].set_xlim([0,1000])
axs[1].set_xlim([1,1000])
```

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:18: User
Warning: Attempted to set non-positive left xlim on a log-scaled axi
s.
Invalid limit will be ignored.
```

Out[37]: (1, 1000)

Fig 8: Simulated mean values of the extremal index estimators under study on a logarithmic scale, for samples of sizen=1000 from the AR processes in (20), with rho=0.1 (left) and rho=0.5 (right).



ARr Process:

a fixed $r \geqslant 1$, and a sequence of i.i.d. r.v.'s $\{\varepsilon_n\}_{n \geqslant 1}$ such that $P(\varepsilon_1 = k/r) = 1/r$, $k = 0, 1, \ldots r-1$,

$$X_j := \frac{1}{r} X_{j-1} + \varepsilon_j, \ \ j \geqslant 1 \quad \text{and} \quad X_0 \frown \text{Uniform}(0, 1)$$

In [38]:
```
@jit(nopython=True)
def arr(r, n):
  x = np.zeros(n) #generating a zero-array of size-n
  x[0] = np.random.uniform(0,1,1)[0] #initiating with a uniform(0,1)
  for i in range(1,n,1):
    e_i = np.random.randint(0,r,1)[0]/r #taking random integer between 0 and r-1 and dividing with r
    x[i] = x[i-1]/r + e_i #computing next variable, from the previous value
  return x #returning the results for plotting
```

Simulation of ARr Process

for r = 2

In [39]:
```
# For figure 9.1
n = 1000
r = 2
times = 1000
path1 = np.zeros(n)
path2 = np.zeros(n)
logpath1 = np.zeros(n)
logpath2 = np.zeros(n)

for t in range(0, times, 1):
  if t%1000 == 0:
    print(t, end=" ")
  sample = np.array(arr(r, n))
  sample_log = np.log(sample)
  k_range = list(range(n))
  straight_line1 = 0.5 * np.ones(n)
  for k in k_range:
    # if k%100 == 0:
    #   # print(k, end = " ")
    path1[k] += theta_n_k(sample, k)
    path2[k] += theta_GJ_k(sample, k, delta = 0.25)
    logpath1[k] += theta_n_k(sample_log, k)
    logpath2[k] += theta_GJ_k(sample_log, k, delta = 0.25)

path1 = path1/times
path2 = path2/times
logpath1 = logpath1/times
logpath2 = logpath2/times
```

```
0
```

r = 10

In [40]:
```
# For figure 9.2
n = 1000
r = 5
times = 1000
path1_ = np.zeros(n)
path2_ = np.zeros(n)
logpath1_ = np.zeros(n)
logpath2_ = np.zeros(n)

for t in range(0, times, 1):
  if t%1000 == 0:
    print(t, end=" ")
  sample = np.array(arr(r, n))
  sample_log = np.log(sample)
  k_range = list(range(n))
  straight_line2 = 0.8 * np.ones(n)
  for k in k_range:
    # if k%100 == 0:
    #   # print(k, end = " ")
    path1_[k] += theta_n_k(sample, k)
    path2_[k] += theta_GJ_k(sample, k, delta = 0.25)
    logpath1_[k] += theta_n_k(sample_log, k)
    logpath2_[k] += theta_GJ_k(sample_log, k, delta = 0.25)

path1_ = path1_/times
path2_ = path2_/times
logpath1_ = logpath1_/times
logpath2_ = logpath2_/times
```

```
0
```

In [41]:
```
# For figure 9.3
n = 1000
r = 10
times = 1000
_path1 = np.zeros(n)
_path2 = np.zeros(n)
_logpath1 = np.zeros(n)
_logpath2 = np.zeros(n)

for t in range(0, times, 1):
  if t%1000 == 0:
    print(t, end=" ")
  sample = np.array(arr(r, n))
  sample_log = np.log(sample)
  k_range = list(range(n))
  straight_line3 = 0.9 * np.ones(n)
  for k in k_range:
    # if k%100 == 0:
    #   # print(k, end = " ")
    _path1[k] += theta_n_k(sample, k)
    _path2[k] += theta_GJ_k(sample, k, delta = 0.25)
    _logpath1[k] += theta_n_k(sample_log, k)
    _logpath2[k] += theta_GJ_k(sample_log, k, delta = 0.25)

_path1 = _path1/times
_path2 = _path2/times
_logpath1 = _logpath1/times
_logpath2 = _logpath2/times
```

```
0
```

In [42]:
```
fig, axs = plt.subplots(1, 3, figsize = (20,6))
fig.suptitle('Fig. 9. Simulated mean values of the extremal index estimators under study in a lnk-scale, for samples of size n=1000 from the ARr(1) processes in (21), withr=2 (left),r=5 (center) and r=10 (right)')

axs[0].plot(k_range, logpath1, "g-")
axs[0].plot(k_range, logpath2, "r-")
axs[0].plot(k_range, straight_line1, "k-")
axs[0].set_title('r=2')
axs[0].set_xscale('log')
# axs[0].set_yscale('log')

axs[1].plot(k_range, logpath1_, "g-")
axs[1].plot(k_range, logpath2_, "r-")
axs[1].plot(k_range, straight_line2, "k-")
axs[1].set_title('r=5')
axs[1].set_xscale('log')
# axs[1].set_yscale('log')

axs[2].plot(k_range, _logpath1, "g-")
axs[2].plot(k_range, _logpath2, "r-")
axs[2].plot(k_range, straight_line3, "k-")
axs[2].set_title('r=10')
axs[2].set_xscale('log')
# axs[2].set_yscale('log')


axs[0].set_xlim([1,1000])
axs[0].set_ylim([0,1.0])
axs[1].set_xlim([1,1000])
axs[1].set_ylim([0,1.0])
axs[2].set_xlim([1,1000])
axs[2].set_ylim([0,1.0])
```

Out[42]: (0.0, 1.0)

Fig. 9. Simulated mean values of the extremal index estimators under study in a lnk-scale, for samples of size n=1000 from the ARr(1) processes in (21), withr=2 (left),r=5 (center) and r=10 (right)