In [11]:
```python
# getting the library that has some of the functions needed for simulations
import numpy as np
import matplotlib.pyplot as plt
from numba import jit
import time
from numba import njit, prange
import pandas as pd

%matplotlib inline
```

Generating the Path of the Armax

The extreme value (EV) is defined by

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

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

ARMAX is defined by

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

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

and there exists a relation

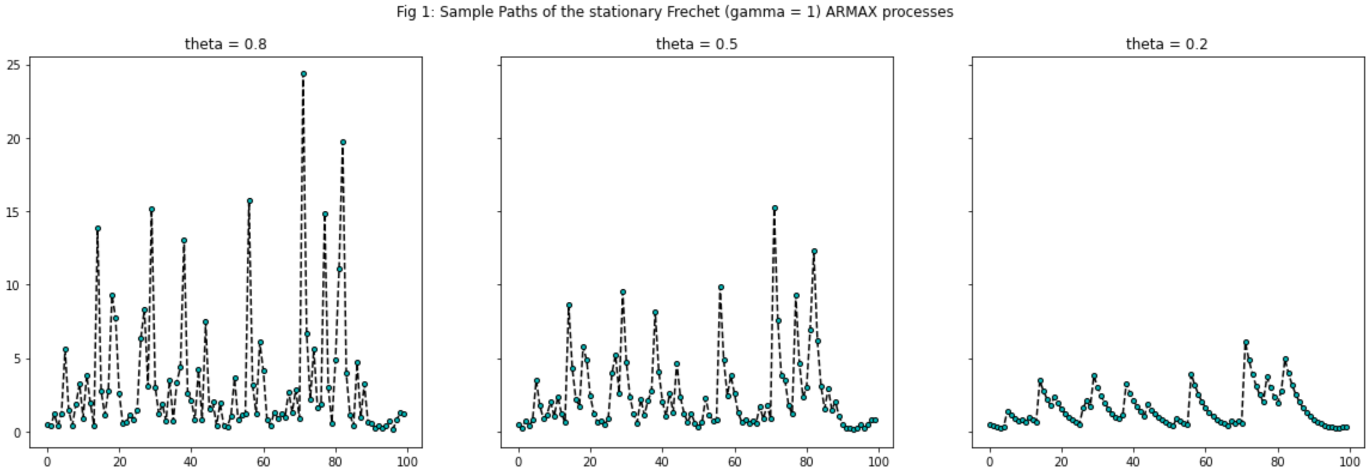
In [12]:
```python
# getting the cdf inverse
def cdf_inv_fr(u, gamma):
    return ((pow(-np.log(u) , -gamma)))

# calculating cdf inverse of H
def cdf_inv_H(u , gamma, beta):
    return (pow(-np.log(u)/(pow(beta , -1/gamma)-1) , -gamma) )

# generating the armax
def armax(beta , gamma, n, random_state = 124):
    """
    Generates ARMAX
    Inputs:
        beta: the value of beta
        gamma: the value of gamma as given above
        n: number of samples
        random_state: to fix the random seed for consistent results over different iterations (

    Returns:
        ARMAX sample
    """
  # array to store samples
    x = np.zeros(n)

  # for fixing random seed for the current iteration only
    r = np.random.RandomState(random_state)

  # generating uniform distribution
    u = r.uniform(0,1,1)[0]

  # x0
```

```
    x0 = cdf_inv_fr(u,gamma)

  # the lag for further calculations
    xi_lag = x0
  # setting the lag for first element
    x[0] = x0

  # variable to iterate over time
    t = 1
  # the calculations for generating armax
    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

  # return armax
    return x
```

In [13]:
```
# generatin armax with different betas (theta = 1 - beta)
armx1 = armax(0.2 , 1 , 100)
armx2 = armax(0.5 , 1 , 100)
armx3 = armax(0.8 , 1 , 100)
```

In [14]:
```
fig, axs = plt.subplots(1, 3, sharey = True, figsize = (20,6))
fig.suptitle('Fig 1: Sample Paths of the stationary Frechet (gamma = 1) ARMAX processes')
axs[0].plot(armx1, "--ko", ms = 4, mec = "k", mfc = "c")
axs[0].set_title('theta = 0.8')
axs[1].plot(armx2, '--ko', ms = 4, mec = "k", mfc = "c")
axs[1].set_title('theta = 0.5')
axs[2].plot(armx3, "--ko", ms = 4, mec = "k", mfc = "c")
axs[2].set_title("theta = 0.2")
```

Out[14]: `Text(0.5, 1.0, 'theta = 0.2')`



Estimating theta

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

In [15]:
```
def theta_n_k(X, k_=1):
    """
      Defining the function to calculate the estimate of theta
    """
```

```
    # calculating sum
    sum = 0

    # handling non-integral k
    k = int(k_)

    # getting the k-th descending order statistic
    X_k = np.partition(X, n-k-1)[n-k-1]

    # counting the elements meeting the condition
    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

    # to handle division by 0 in some steps
    if k == 0:
        return 1

    # return the estimate of theta
    return sum/k
```

Estimating Generalised Jackknife estimate of theta

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

In [16]:

```
def theta_GJ_k(X, k, delta):
    """
        Defining the function to calculate the generalised jackknife estimate of theta
    """
    numerator = (delta*delta + 1) * theta_n_k(X, int(np.floor(delta*k)) + 1) - delta*(theta_n_k
    denominator = (1 - delta)**2

    if numerator < 0:
        return 0

    return numerator/denominator
```

Sample paths for the extremal index estimator

In [17]:

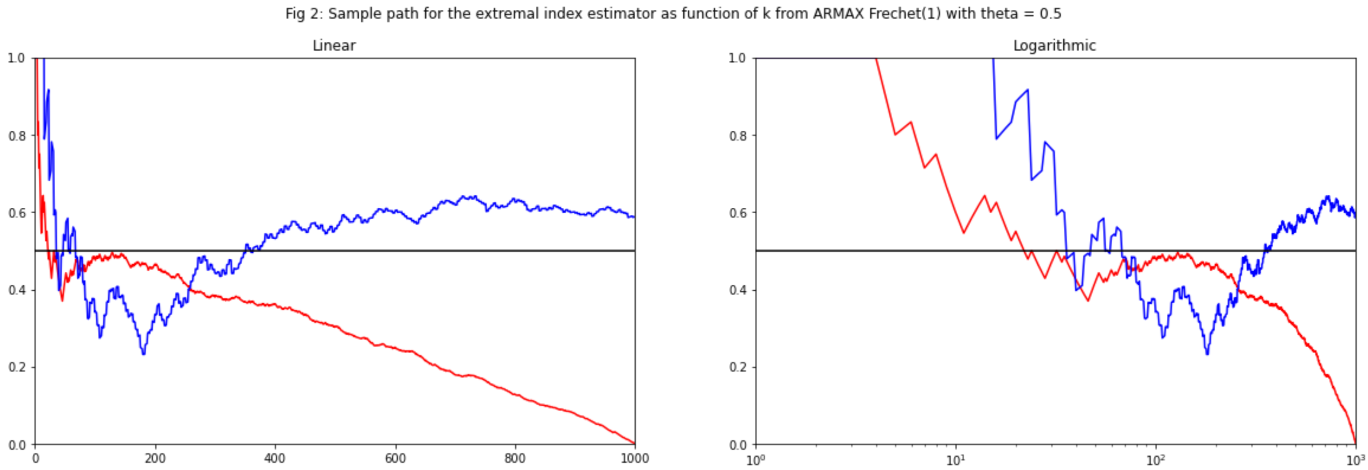
```
# For figure 2
n = 1000

# getting the armax sample
sample = np.array(armax(0.5, 1, n))

k_range = list(range(n))
straight_line = 0.5 * np.ones(n)

# allocating space for calculations
path1 = np.zeros(n)
path2 = np.zeros(n)

# getting the values
for k in k_range:
    path1[k] = theta_n_k(sample, k)
    path2[k] = theta_GJ_k(sample, k, delta = 0.25)
```

In [19]:

```
# plotting
fig, axs = plt.subplots(1, 2, figsize = (20,6))
fig.suptitle('Fig 2: Sample path for the extremal index estimator as function of k from ARMAX F
axs[0].plot(k_range, path1, "r-")
axs[0].plot(k_range, path2, "b-")
```

```python
axs[0].plot(k_range, straight_line, "k-")
axs[0].set_title('Linear')
axs[1].plot(k_range[1:], path1[1:], "r-")
axs[1].plot(k_range[1:], path2[1:], "b-")
axs[1].plot(k_range[1:], straight_line[1:], "k-")
axs[1].set_title('Logarithmic')
axs[1].set_xscale('log')
axs[0].set_ylim([0,1])
axs[1].set_ylim([0,1])
axs[0].set_xlim([0,1000])
axs[1].set_xlim([1,1000])
```

Out[19]: (1, 1000)



Rewriting functions again for the more efficient (faster) computation for the simulations of expected value and MSE

The only change is adding '@jit(nopython=True)' or '@njit' which would help to make the function acceptable to the library for parallel processing

In [ ]:
```python
@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) )

@jit(nopython=True)
def armax(beta , gamma, n):
  x = np.zeros(n)
  # Note random state has now been removed
  #r = np.random.RandomState(random_state)
  u = np.random.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 = np.random.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 [ ]:
```python
@jit(nopython=True)
def theta_n_k(X, n, 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 1
  return sum/k


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

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

  if numerator < 0:
    return 0

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

In [ ]:

```
@jit(nopython = True)
def np_apply_along_axis(func1d, axis, arr):
    """
       A workaround through a problem of calculating mean while parallelizing
    """
  assert arr.ndim == 2
  assert axis in [0, 1]
  if axis == 0:
    result = np.empty(arr.shape[1])
    for i in range(len(result)):
      result[i] = func1d(arr[:, i])
  else:
    result = np.empty(arr.shape[0])
    for i in range(len(result)):
      result[i] = func1d(arr[i, :])
  return result
```

In [ ]:

```
@jit(nopython=True)
def simulate_mean_mse(n, theta, runs = 15, replicates = 10):
    """
        Function to get simulated mean and mse for the estimator of theta
        n: size of each sample generated
        theta: true value of estimator
        runs: number of runs
        replicates: number of replicates in each run

        Returns arrays for mean, mse
    """
  # allot space for arrays
  all_values_mean = np.zeros((runs, n))
  all_values_mse = np.zeros((runs, n))

  # loop for runs
  for run in range(runs):
    a = np.zeros((replicates, n))
```

```python
    # loop for replicates
    for i in range(replicates):
      a[i] = armax(1-theta, 1, n)

    # space for storing mean, mse
    path = np.zeros((replicates, n))
    path2 = np.zeros((replicates, n))

    k_range = list(range(n))

    # calculating mean, mse of the replicates
    for k in k_range:
      for j in range(replicates):
        path[j][k] = theta_n_k(a[j], n, k)
        path2[j][k] = (path[j][k] - theta)**2

    # store the values
    all_values_mean[run] = np_apply_along_axis(np.mean, 0, path)
    all_values_mse[run] = np_apply_along_axis(np.mean, 0, path2)

  # returning calculated averaged values
  return np_apply_along_axis(np.mean, 0, all_values_mean), np_apply_along_axis(np.mean, 0, all_
```

In [ ]:

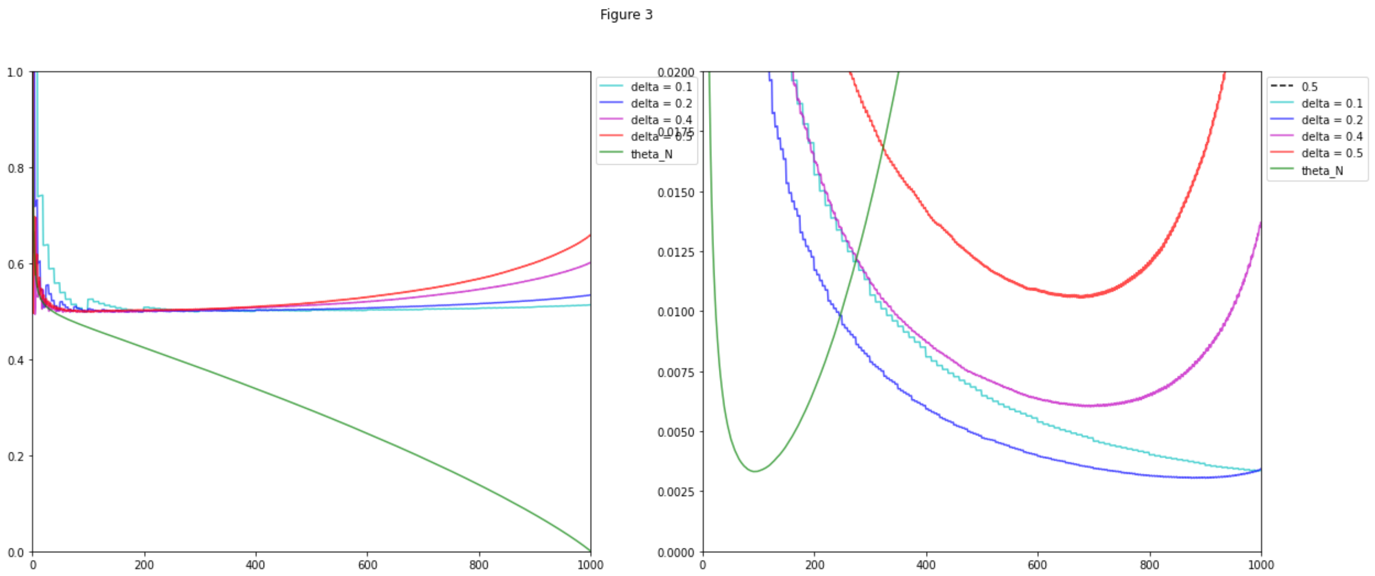
```python
@jit(nopython=True)
def simulate_mean_mse_GJ(n, theta, delta = 0.25, runs = 15, replicates = 10):
        """
        Function to get simulated mean and mse for the generalised jackknife estimator of theta
        n: size of each sample generated
        theta: true value of estimator
        runs: number of runs
        replicates: number of replicates in each run

        Returns arrays for mean, mse
    """
  # array to store values
  all_values_mean = np.zeros((runs, n))
  all_values_mse = np.zeros((runs, n))

  # loop for runs
  for run in range(runs):
    a = np.zeros((replicates, n))

    # loop for replicates
    for i in range(replicates):
      a[i] = armax(1-theta, 1, n)

    # allocating space
    path = np.zeros((replicates, n))
    path2 = np.zeros((replicates, n))

    k_range = list(range(n))

    # calculating mean, mse
    for k in k_range:
      for j in range(replicates):
        path[j][k] = theta_GJ_k(a[j], n, k, delta)
        path2[j][k] = (path[j][k] - theta)**2


    # adding to the array the value of e, mse
    all_values_mean[run] = np_apply_along_axis(np.mean, 0, path)
    all_values_mse[run] = np_apply_along_axis(np.mean, 0, path2)

  # return the averaged values
  return np_apply_along_axis(np.mean, 0, all_values_mean), np_apply_along_axis(np.mean, 0, all_
```

In [ ]:

```python
%%time
# plotting the calculations from the above results
```

```
runs = 5000
replicates = 10
n = 1000
theta = 0.5
e_1000_5, mse_1000_5 = simulate_mean_mse(n, theta, runs = runs, replicates = replicates)
e_1000_5_1, mse_1000_5_1 = simulate_mean_mse_GJ(n, theta, delta = 0.1, runs = runs, replicates
e_1000_5_2, mse_1000_5_2 = simulate_mean_mse_GJ(n, theta, delta = 0.2, runs = runs, replicates
e_1000_5_4, mse_1000_5_4 = simulate_mean_mse_GJ(n, theta, delta = 0.4, runs = runs, replicates
e_1000_5_5, mse_1000_5_5 = simulate_mean_mse_GJ(n, theta, delta = 0.5, runs = runs, replicates

k_range = np.arange(n)
straight_line = theta*np.ones(n)

a = 0.7
fig, axs = plt.subplots(1, 2, figsize = (20,8))
fig.suptitle("Figure 3")
axs[0].set_ylim([0, 1])
axs[0].set_xlim([0,1000])
axs[0].plot(k_range, e_1000_5_1, 'c-', label = 'delta = 0.1', alpha = a)
axs[0].plot(k_range, e_1000_5_2, 'b-', label = 'delta = 0.2', alpha = a)
axs[0].plot(k_range, e_1000_5_4, 'm-', label = 'delta = 0.4', alpha = a)
axs[0].plot(k_range, e_1000_5_5, 'r-', label = 'delta = 0.5', alpha = a)
axs[0].plot(k_range, e_1000_5, 'g-', label = 'theta_N', alpha = a)
axs[1].plot(k_range, straight_line, 'k--', label = '0.5')
axs[0].legend(bbox_to_anchor=(1.2, 1))

axs[1].set_ylim([0, 0.02])
axs[1].set_xlim([0,1000])
axs[1].plot(k_range, mse_1000_5_1, 'c-', label = 'delta = 0.1', alpha = a)
axs[1].plot(k_range, mse_1000_5_2, 'b-', label = 'delta = 0.2', alpha = a)
axs[1].plot(k_range, mse_1000_5_4, 'm-', label = 'delta = 0.4', alpha = a)
axs[1].plot(k_range, mse_1000_5_5, 'r-', label = 'delta = 0.5', alpha = a)
axs[1].plot(k_range, mse_1000_5, 'g-', label = 'theta_N', alpha = a)
axs[1].legend(bbox_to_anchor=(1.2, 1))
```

```
CPU times: user 3h 4min 18s, sys: 9.43 s, total: 3h 4min 27s
Wall time: 3h 4min 16s
```

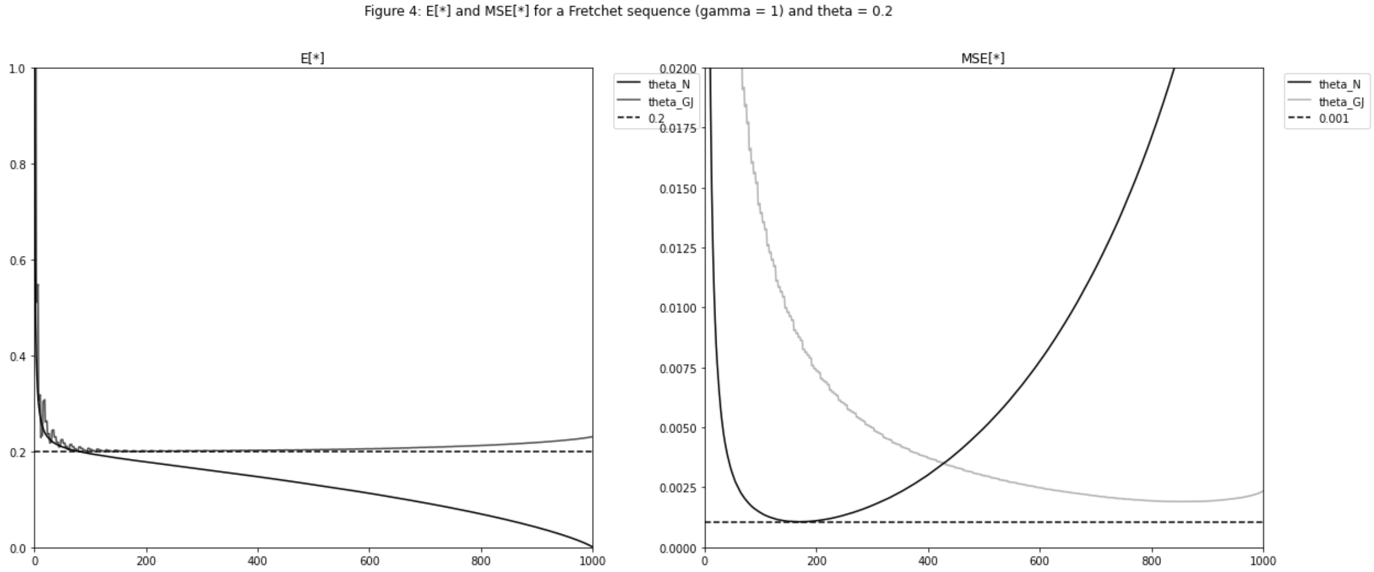


In [ ]:

```
%%time

# calculating from above functions and plotting
n = 1000
theta = 0.2
alph = 0.3
delta = 0.25
runs = 5000
replicates = 10

e_1000_2, mse_1000_2 = simulate_mean_mse(n, theta, runs, replicates)
e_1000_2_25, mse_1000_2_25 = simulate_mean_mse_GJ(n, theta, delta, runs, replicates)

straight_line = theta*np.ones(n)
```

```
fig, axs = plt.subplots(1, 2, figsize = (20,8))
fig.suptitle("Figure 4: E[*] and MSE[*] for a Fretchet sequence (gamma = 1) and theta = " + str
axs[0].set_ylim([0, 1])
axs[0].set_xlim([0,1000])
axs[0].set_title("E[*]")
axs[0].plot(k_range, e_1000_2, 'k-', label = 'theta_N')
axs[0].plot(k_range, e_1000_2_25, 'k-', label = 'theta_GJ', alpha = alph)
axs[0].plot(k_range, straight_line, 'k--', label = str(theta))
axs[0].legend(bbox_to_anchor=(1.2, 1))

m = mse_1000_2
mse_line = m[np.where(m == m.min())[0][0]] *np.ones(n)

axs[1].set_ylim([0, 0.02])
axs[1].set_xlim([0,1000])
axs[1].set_title("MSE[*]")
axs[1].plot(k_range, mse_1000_2, 'k-', label = 'theta_N')
axs[1].plot(k_range, mse_1000_2_25, 'k-', label = 'theta_GJ', alpha = alph)
axs[1].plot(k_range, mse_line, 'k--', label = str(m[np.where(m == m.min())[0][0]])[:5])
axs[1].legend(bbox_to_anchor=(1.2, 1))
```

```
CPU times: user 47min 1s, sys: 2.39 s, total: 47min 3s
Wall time: 47min 1s
```



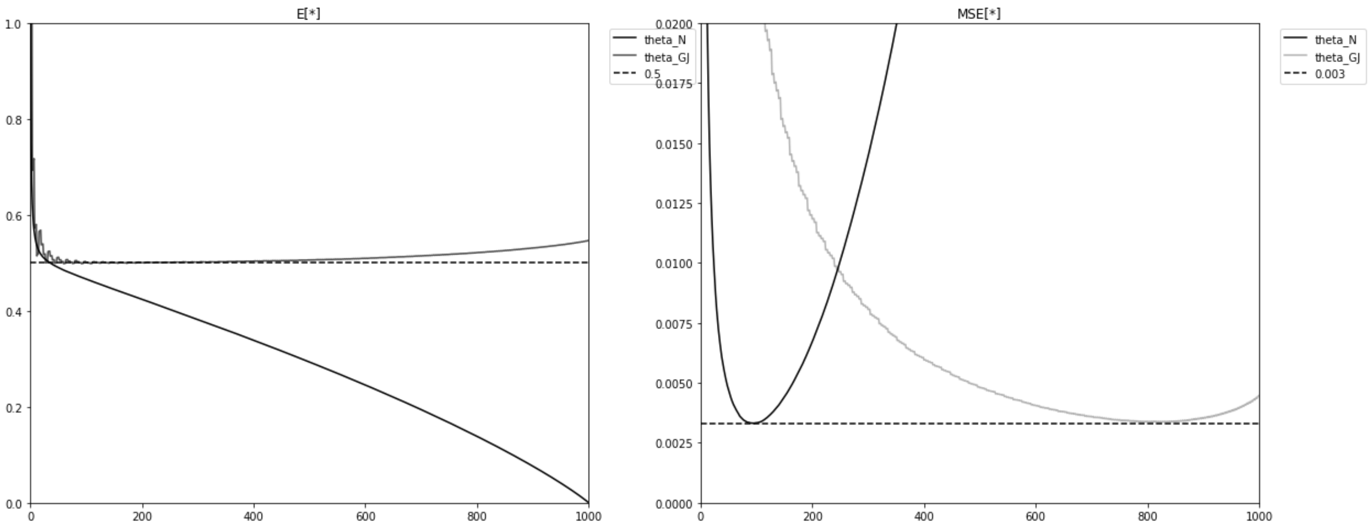
In [ ]:

```
%%time

# calculating and plotting
n = 1000
theta = 0.5
alph = 0.3
delta = 0.25
runs = 5000
replicates = 10

e_1000_2, mse_1000_2 = simulate_mean_mse(n, theta, runs, replicates)
e_1000_2_25, mse_1000_2_25 = simulate_mean_mse_GJ(n, theta, delta, runs, replicates)

straight_line = theta*np.ones(n)
fig, axs = plt.subplots(1, 2, figsize = (20,8))
fig.suptitle("Figure 5: E[*] and MSE[*] for a Fretchet sequence (gamma = 1) and theta = " + str
axs[0].set_ylim([0, 1])
axs[0].set_xlim([0,1000])
axs[0].set_title("E[*]")
axs[0].plot(k_range, e_1000_2, 'k-', label = 'theta_N')
axs[0].plot(k_range, e_1000_2_25, 'k-', label = 'theta_GJ', alpha = alph)
axs[0].plot(k_range, straight_line, 'k--', label = str(theta))
axs[0].legend(bbox_to_anchor=(1.2, 1))

m = mse_1000_2
mse_line = m[np.where(m == m.min())[0][0]] *np.ones(n)

axs[1].set_ylim([0, 0.02])
```

```
axs[1].set_xlim([0,1000])
axs[1].set_title("MSE[*]")
axs[1].plot(k_range, mse_1000_2, 'k-', label = 'theta_N')
axs[1].plot(k_range, mse_1000_2_25, 'k-', label = 'theta_GJ', alpha = alph)
axs[1].plot(k_range, mse_line, 'k--', label = str(m[np.where(m == m.min())[0][0]])[:5])
axs[1].legend(bbox_to_anchor=(1.2, 1))
```

```
CPU times: user 57min 49s, sys: 5.09 s, total: 57min 54s
Wall time: 57min 49s
```

Figure 5: E[*] and MSE[*] for a Fretchet sequence (gamma = 1) and theta = 0.5



In [ ]:

```
%%time

# calculating and plotting
n = 1000
theta = 0.8
alph = 0.3
delta = 0.25
runs = 5000
replicates = 10

e_1000_2, mse_1000_2 = simulate_mean_mse(n, theta, runs, replicates)
e_1000_2_25, mse_1000_2_25 = simulate_mean_mse_GJ(n, theta, delta, runs, replicates)

straight_line = theta*np.ones(n)
fig, axs = plt.subplots(1, 2, figsize = (20,8))
fig.suptitle("Figure 6: E[*] and MSE[*] for a Fretchet sequence (gamma = 1) and theta = " + str
axs[0].set_ylim([0, 1])
axs[0].set_xlim([0,1000])
axs[0].set_title("E[*]")
axs[0].plot(k_range, e_1000_2, 'k-', label = 'theta_N')
axs[0].plot(k_range, e_1000_2_25, 'k-', label = 'theta_GJ', alpha = alph)
axs[0].plot(k_range, straight_line, 'k--', label = str(theta))
axs[0].legend(bbox_to_anchor=(1.2, 1))

m = mse_1000_2
mse_line = m[np.where(m == m.min())[0][0]] *np.ones(n)

axs[1].set_ylim([0, 0.02])
axs[1].set_xlim([0,1000])
axs[1].set_title("MSE[*]")
axs[1].plot(k_range, mse_1000_2, 'k-', label = 'theta_N')
axs[1].plot(k_range, mse_1000_2_25, 'k-', label = 'theta_GJ', alpha = alph)
axs[1].plot(k_range, mse_line, 'k--', label = str(m[np.where(m == m.min())[0][0]])[:5])
axs[1].legend(bbox_to_anchor=(1.2, 1))
```

```
CPU times: user 1h 1min 45s, sys: 3.41 s, total: 1h 1min 48s
Wall time: 1h 1min 45s
```

Figure 6: E[*] and MSE[*] for a Fretchet sequence (gamma = 1) and theta = 0.8