[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm
import time
```

1 Dans la suite on estime un echantillon de loi exponentielle(1)

1.1 Estimateur de Parzen-Rosenblatt

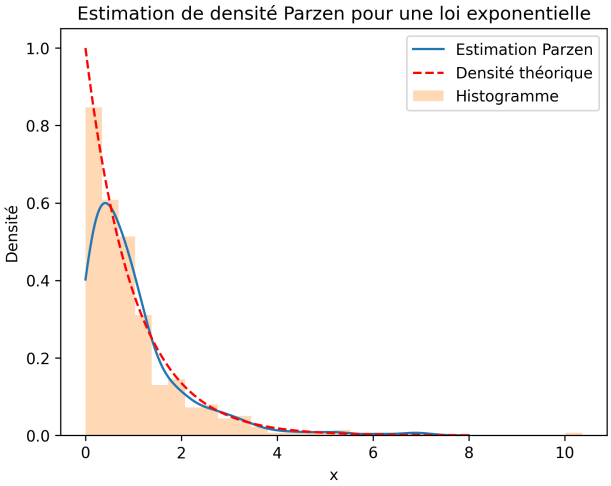
$f_n(x) = \frac{1}{nh_n} \sum_{k=1}^{n} K(\frac{x-X_k}{h_n})$

Où $K(x) = \frac{1}{\sigma\sqrt{2\pi}} exp(\frac{-x^2}{2\sigma^2})$

[2]:
```
#suite deterministe stric pos et decroissante vers 0, dite fenetre
def window(n, alpha=0.2):
    """Largeur de fenêtre décroissante avec n."""
    return n ** (-alpha)

#fonction K dite noyau gaussien
def gaussian_kernel(u, sigma=1):
    """Noyau Gaussien standard."""
    return np.exp(-u**2 / (2*sigma**2)) / (sigma * np.sqrt(2 * np.pi))


#premier estimateur de densité de Parzen-Rozenblatt
def parzen_estimator(x, sample, sigma=1, alpha=0.4):
    """
    Estimateur de Parzen-Rosenblatt en un point x.

    Args:
        x (float): Point où estimer la densité.
        sample (array): Échantillon de données.
        hn (float): Largeur de fenêtre dépendant de n.
        sigma (float): Paramètre du noyau (optionnel).

    Returns:
        float: Estimation de la densité en x.
    """
    n = len(sample)
    hn = window(n)
    u = (x - sample) / hn  # Normalisation par la fenêtre
    kernel_values = gaussian_kernel(u, sigma)
    return np.sum(kernel_values) / (n* hn)
```

[23]:
```
# Generate sample data from an exponential law (λ=1)
sample = np.random.exponential(scale=1, size=400)
x_grid = np.linspace(0, 8, 1000)  # Support de l'exponentielle: x >= 0
```

```
# Compute density estimates using the Parzen estimator
start_time = time.perf_counter()
density = np.array([parzen_estimator(x, sample) for x in x_grid])
end_time = time.perf_counter()

plt.plot(x_grid, density, label="Estimation Parzen")
plt.plot(x_grid, true_density, "r--", label="Densité théorique")
plt.hist(sample, bins=30, density=True, alpha=0.3, label="Histogramme")
plt.xlabel("x")
plt.ylabel("Densité")
plt.title("Estimation de densité Parzen pour une loi exponentielle")
plt.legend()
plt.savefig("estimation_parzen_exponentielle.png", dpi=300, bbox_inches="tight")
plt.show()

print(f"Temps Parzen-Rosenblatt: {end_time - start_time:.4f}s")
```



Temps Parzen-Rosenblatt: 0.0283s

L'estimateur de Parzen-Rosenblatt est le plus rapide

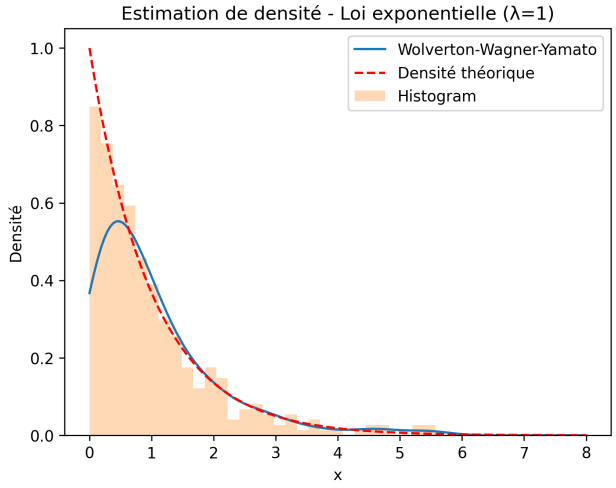
1.2 Estimateur de Wolverton-Wagner-Yamato

$f_n(x) = \frac{1}{n}\sum_{k=1}^{n} \frac{1}{h_k} K(\frac{x-X_k}{h_k})$

[3]:
```
#estimateur de W-W-Y
def estimator_WWY(x, sample, sigma=1, alpha=0.1):
    """
    Estimateur de WWY en un point x.

    Args:
        x (float): Point où estimer la densité.
```

```
        sample (array): Échantillon de données.
        h (float): Largeurs des fenetres.
        sigma (float): Paramètre du noyau (optionnel).

    Returns:
        float: Estimation de la densité en x.
    """
    n = len(sample)
    h = np.array([window(i) for i in range(1,n+1)])
    u = (x - sample) / h  # Normalisation par la fenêtre
    kernel_values = gaussian_kernel(u, sigma)
    return np.sum(kernel_values / h) / n
```

[4]:
```
# Generate sample data from an exponential law (λ = 1)
sample = np.random.exponential(scale=1, size=400)
x_grid = np.linspace(0, 8, 1000)  # Support de l'exponentielle: x  0

# Compute density estimates using the estimator WWY
start_time = time.perf_counter()  # Début du chronométrage
density = np.array([estimator_WWY(x, sample) for x in x_grid])
end_time = time.perf_counter()    # Fin du chronométrage

# Compute theoretical density for the exponential law (λ=1): f(x) = exp(-x)
true_density = np.exp(-x_grid)

# Plot the density estimation, histogram, and theoretical density
plt.plot(x_grid, density, label="Wolverton-Wagner-Yamato")
plt.plot(x_grid, true_density, "r--", label="Densité théorique")
plt.hist(sample, bins=30, density=True, alpha=0.3, label="Histogram")
plt.xlabel("x")
plt.ylabel("Densité")
plt.title("Estimation de densité - Loi exponentielle (λ=1)")
plt.legend()
plt.savefig("estimation_WWY_exponentielle.png", dpi=300, bbox_inches="tight")
plt.show()


print(f"Temps Wolverton-Wagner-Yamato: {end_time - start_time:.4f} secondes")
```



Temps Wolverton-Wagner-Yamato: 0.1462 secondes

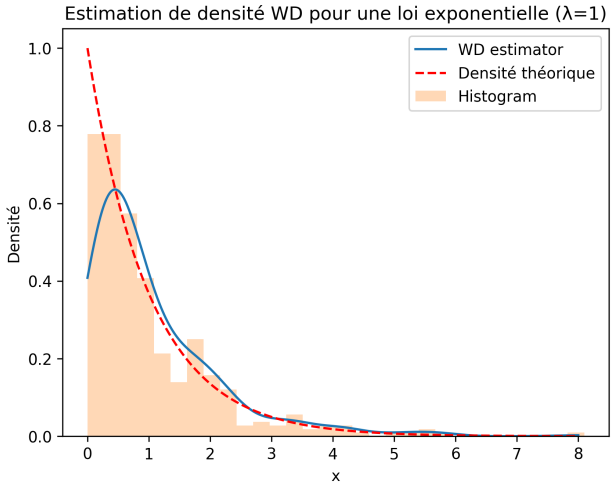
1.3 Estimateur de Weglan-Davies

$f_n(x) = \frac{1}{nh_n^{1/2}} \sum_{k=1}^{n} \frac{1}{h_k^{1/2}} K(\frac{x-X_k}{h_k})$

[5]:
```python
#estimateur de W-D
def estimator_WD(x, sample, sigma=1, alpha=0.8):
    """
    Estimateur de W-D en un point x.

    Args:
        x (float): Point où estimer la densité.
        sample (array): Échantillon de données.
        hn (float): Largeur de fenêtre dépendant de n.
        sigma (float): Paramètre du noyau (optionnel).

    Returns:
        float: Estimation de la densité en x.
    """
    n = len(sample)
    h = np.array([window(i) for i in range(1,n+1)])
    u = (x - sample) / h  # Normalisation par la fenêtre
    kernel_values = gaussian_kernel(u, sigma)
    return np.sum(kernel_values / np.sqrt(h)) / (n*np.sqrt(h[-1]))
```

[6]:
```python
# Generate sample data from an exponential law (λ=1)
sample = np.random.exponential(scale=1, size=400)
x_grid = np.linspace(0, 8, 1000)  # Le support de l'exponentielle est [0, )

# Compute density estimates using the WD estimator
start_time = time.perf_counter()  # Début du chronométrage
```

```
density = np.array([estimator_WD(x, sample) for x in x_grid])
end_time = time.perf_counter()     # Fin du chronométrage

# Plot
plt.plot(x_grid, density, label="WD estimator")
plt.plot(x_grid, true_density, "r--", label="Densité théorique")
plt.hist(sample, bins=30, density=True, alpha=0.3, label="Histogram")
plt.xlabel("x")
plt.ylabel("Densité")
plt.title("Estimation de densité WD pour une loi exponentielle (λ=1)")
plt.legend()
plt.savefig("estimation_WD_exponentielle.png", dpi=300, bbox_inches="tight")
plt.show()

print(f"Temps Weglan-Davies: {end_time - start_time:.4f} secondes")
```



Temps Weglan-Davies: 0.1203 secondes

1.4 Estimateur de Revesz

$\widehat{f_n}(x) = (1-\gamma_n)\widehat{f_{n-1}}(x) + \gamma_n W_n(x)$

Avec: $W_n(x) = \frac{1}{h_n} K(\frac{x-X_n}{h_n})$

$\widehat{f_0}(x) = 0$

(γ_n) une suite deterministe, positive, décroissante vers 0, dont la série est divergente.

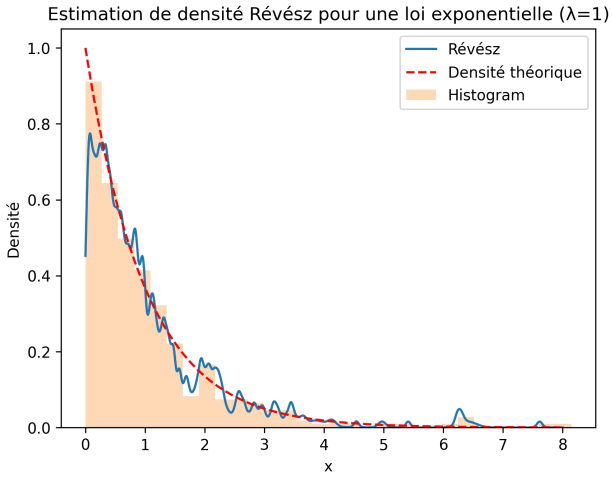
On prend ici $\gamma_n = \frac{a_n}{A_n}$ avec:

$$A_n = \sum_{k=0}^{n} a_k$$

[7]:
```python
def estimator_R(x, sample, sigma=1, alpha=0.6):
    """
    Paramètres
    ----------
    x : ndarray
    Points d'évaluation où calculer la densité (shape: (m,))
    sample : ndarray
    Échantillon de données observées séquentiellement (shape: (n,))
    sigma : float, optionnel
    Paramètre de lissage du noyau gaussien (défaut=1)
    alpha : float, optionnel
    Paramètre de décroissance de la fenêtre (défaut=0.6)

    Retourne
    -------
    ndarray
    Estimation de densité aux points x (shape: (m,))
    """
    n = len(sample)

    # Cas de base
    if n == 0:
        return np.zeros_like(x)

    # Calcul de a_k et A_n
    a_k = np.array([window(k,alpha) for k in range(1, n+1)])
    A_n = np.sum(a_k)
    gamma_n = a_k[-1] / A_n if n > 1 else 1  # γ = a/A

    # Termes pour la récursion
    h_n = window(n, alpha)
    u = (x - sample[-1]) / h_n  # Dernière observation
    W_n = (1/h_n) * gaussian_kernel(u, sigma)

    # Appel récursif
    f_prev = estimator_R(x, sample[:-1], sigma, alpha) if n > 1 else np.
↪zeros_like(x)

    return (1 - gamma_n) * f_prev + gamma_n * W_n
```

[8]:
```python
# Generate sample data from an exponential law (λ=1)
sample = np.random.exponential(scale=1, size=400)
x_grid = np.linspace(0, 8, 1000)  # Support de l'exponentielle : x  0

import time  # Import de time
start_time = time.perf_counter()  # Début du chronométrage
density = estimator_R(x_grid, sample)
```

```
end_time = time.perf_counter()  # Fin du chronométrage

plt.plot(x_grid, density, label="Révész")
plt.plot(x_grid, true_density, "r--", label="Densité théorique")
plt.hist(sample, bins=30, density=True, alpha=0.3, label="Histogram")
plt.xlabel("x")
plt.ylabel("Densité")
plt.title("Estimation de densité Révész pour une loi exponentielle (λ=1)")
plt.legend()
plt.savefig("estimation_R_exponentielle.png", dpi=300, bbox_inches="tight")
plt.show()

print(f"Temps Révész: {end_time - start_time:.4f} secondes")
```

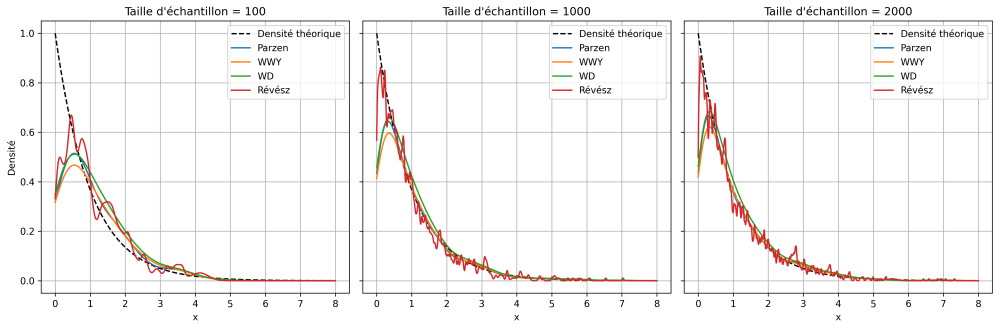


Temps Révész: 0.0531 secondes

Dans le prochain plot, on estime avec les 4 premiers estimateurs la loi exponentielle pour différentes tailles d'échantillon.

[9]:
```
# Définition des tailles d'échantillon et de la grille d'évaluation
sample_sizes = [100, 1000, 5000]
x_grid = np.linspace(0, 8, 1000)  # On couvre l'essentiel du support [0, infini)

# Dictionnaire des estimateurs déjà définis dans le notebook
estimators = {
    'Parzen': parzen_estimator,
    'WWY': estimator_WWY,
    'WD': estimator_WD,
    'Révész': estimator_R
}

# Fonction wrapper déjà définie pour assurer une sortie scalaire
def safe_estimator(estimator, x, sample, **kwargs):
```

```
    res = estimator(np.array([x]), sample, **kwargs)
    return res[0] if isinstance(res, np.ndarray) else res

# Fonction pour calculer la courbe d'estimation sur la grille
def estimate_curve(estimator, sample, x_grid, **kwargs):
    return np.array([safe_estimator(estimator, x, sample, **kwargs) for x in␣
 ↪x_grid])

# Densité théorique de l'exponentielle (λ = 1)
true_density = np.exp(-x_grid)

# Création d'une figure avec un subplot par taille d'échantillon
fig, axs = plt.subplots(1, len(sample_sizes), figsize=(15, 5), sharey=True)
for i, n in enumerate(sample_sizes):
    # Génération d'un échantillon issu de la loi exponentielle (scale=1 <=> λ =␣
 ↪1)
    sample = np.random.exponential(scale=1, size=n)
    ax = axs[i]

    # Tracé de la densité théorique
    ax.plot(x_grid, true_density, 'k--', label='Densité théorique')

    # Tracé des densités estimées pour chaque estimateur
    for name, estimator in estimators.items():
        estimated_curve = estimate_curve(estimator, sample, x_grid)
        ax.plot(x_grid, estimated_curve, label=name)

    ax.set_title(f"Taille d'échantillon = {n}")
    ax.set_xlabel("x")
    if i == 0:
        ax.set_ylabel("Densité")
    ax.legend()
    ax.grid(True)

plt.tight_layout()
plt.savefig("comparaison_est.png", dpi=300, bbox_inches="tight")
plt.show()
```

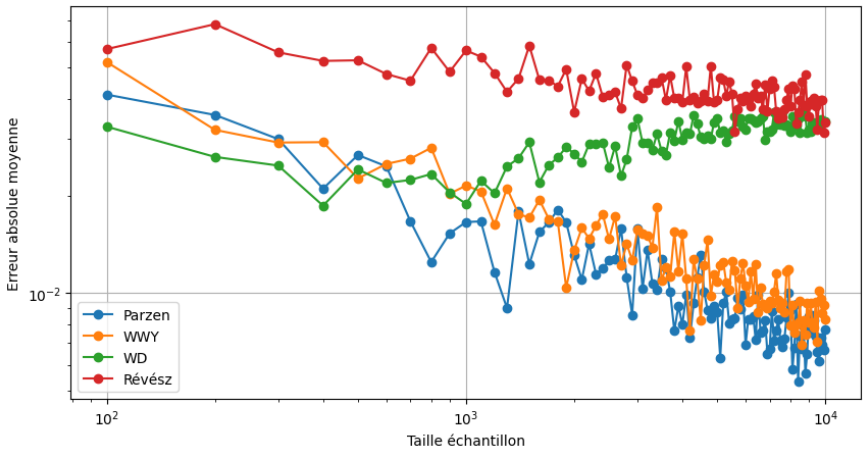


Le plot suivant montre les erreurs absolues moyennes des estimateurs pour des échantillons de loi Normale(0,1).

[10]:
```python
# Configuration
sample_sizes = np.linspace(100, 10000, 100, dtype= int)  # Tailles d'échantillon
n_iter = 20                        # Nombre d'itérations
x_test = 0.0
true_density = 0.3989              # Densité N(0,1) en x=0

# Fonction wrapper pour uniformiser les sorties
def safe_estimator(estimator, x, sample, **kwargs):
    res = estimator(np.array([x]), sample, **kwargs)
    return res[0] if isinstance(res, np.ndarray) else res

# Fonction de test modifiée
def test_convergence(estimator):
    results = {}
    for n in sample_sizes:
        errors = []
        for _ in range(n_iter):
            sample = np.random.randn(n)
            estimate = safe_estimator(estimator, x_test, sample)
            errors.append(np.abs(estimate - true_density))
        results[n] = np.mean(errors)
    return results

# Dictionnaire des estimateurs
estimators = {
    'Parzen': parzen_estimator,
    'WWY': estimator_WWY,
    'WD': estimator_WD,
    'Révész': estimator_R_iterative
}

# Calcul des résultats
results = {}
for name, estimator in estimators.items():
    print(f"Traitement de {name}...")
    results[name] = test_convergence(estimator)

# Visualisation
plt.figure(figsize=(10, 5))
for name, res in results.items():
    sizes, errs = zip(*sorted(res.items()))
    plt.plot(sizes, errs, 'o-', label=name)

plt.xscale('log')
```

```
plt.yscale('log')
plt.xlabel('Taille échantillon')
plt.ylabel('Erreur absolue moyenne')
plt.legend()
plt.grid(True)
plt.show()
```

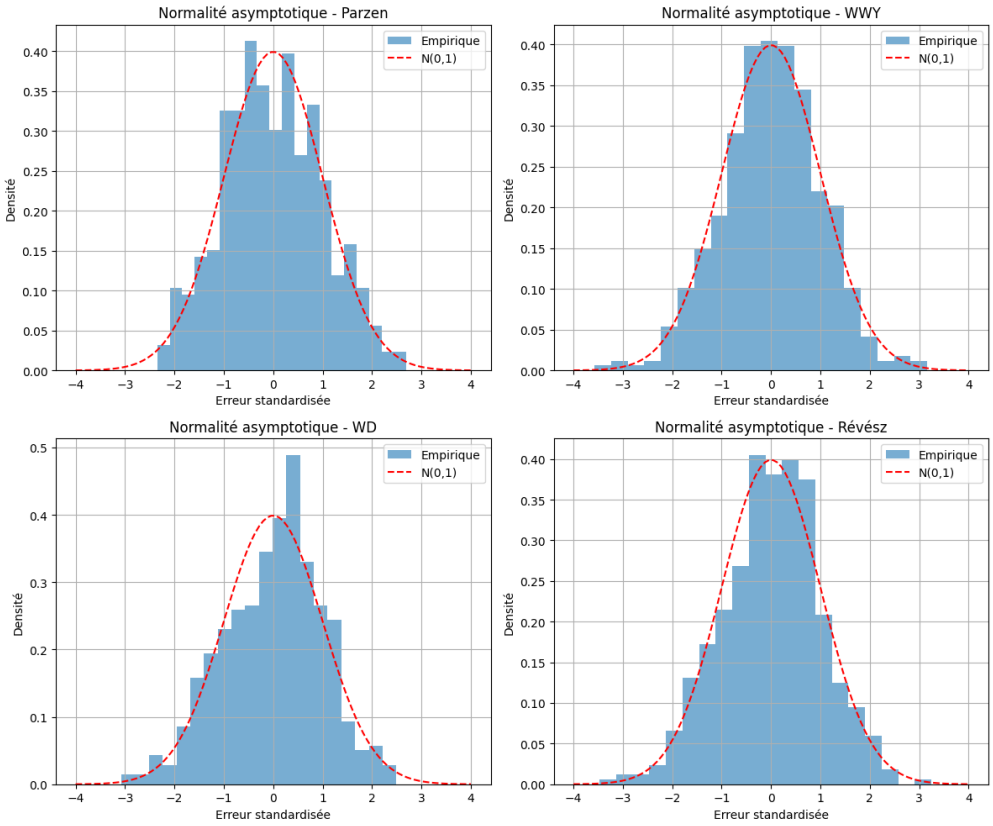
```
Traitement de Parzen...
Traitement de WWY...
Traitement de WD...
Traitement de Révész...
```



Au point x=0, et pour une loi normale centrée réduite, les estimateurs de PR et WWY convergent mieux que les 2 autres estimateurs.

[11]:
```
# Paramètres pour l'étude de normalité asymptotique
n_fixed = 5000        # Taille fixe de l'échantillon
n_iter_norm = 500     # Nombre d'itérations (réplicats)
x_test = 1.0          # Point d'évaluation (choisi dans le support [0, infini) de␣
 ↪l'exponentielle)
true_density = np.exp(-x_test)  # Pour une exponentielle de paramètre λ=1,␣
 ↪f(x)=exp(-x)

# Dictionnaire des estimateurs (ils doivent être déjà définis dans ton notebook)
estimators = {
    'Parzen': parzen_estimator,
    'WWY': estimator_WWY,
    'WD': estimator_WD,
    'Révész': estimator_R_iterative
}

# Fonction wrapper pour uniformiser la sortie
def safe_estimator(estimator, x, sample, **kwargs):
```

```
    res = estimator(np.array([x]), sample, **kwargs)
    return res[0] if isinstance(res, np.ndarray) else res

# Création de la figure avec un sous-graphe par estimateur
fig, axs = plt.subplots(2, 2, figsize=(12, 10))
axs = axs.flatten()

for idx, (name, estimator) in enumerate(estimators.items()):
    estimates = []
    # Répétitions pour construire la distribution empirique de l'estimation
    for _ in range(n_iter_norm):
        # Génération d'un échantillon issu de la loi exponentielle (scale=1 <=>␣
↪λ=1)
        sample = np.random.exponential(scale=1, size=n_fixed)
        est = safe_estimator(estimator, x_test, sample)
        estimates.append(est)
    estimates = np.array(estimates)

    # Calcul des erreurs : différence entre l'estimation et la valeur théorique
    errors = estimates - true_density

    # Standardisation : on soustrait la moyenne empirique et divise par␣
↪l'écart-type
    mean_error = np.mean(errors)
    std_error = np.std(errors)
    errors_std = (errors - mean_error) / std_error

    # Tracé de l'histogramme des erreurs standardisées
    ax = axs[idx]
    ax.hist(errors_std, bins=20, density=True, alpha=0.6, label="Empirique")

    # Superposition de la densité d'une loi normale standard
    x_vals = np.linspace(-4, 4, 100)
    normal_pdf = norm.pdf(x_vals)
    ax.plot(x_vals, normal_pdf, 'r--', label="N(0,1)")

    ax.set_title(f"Normalité asymptotique - {name}")
    ax.set_xlabel("Erreur standardisée")
    ax.set_ylabel("Densité")
    ax.legend()
    ax.grid(True)

plt.tight_layout()
plt.savefig
plt.show()
```



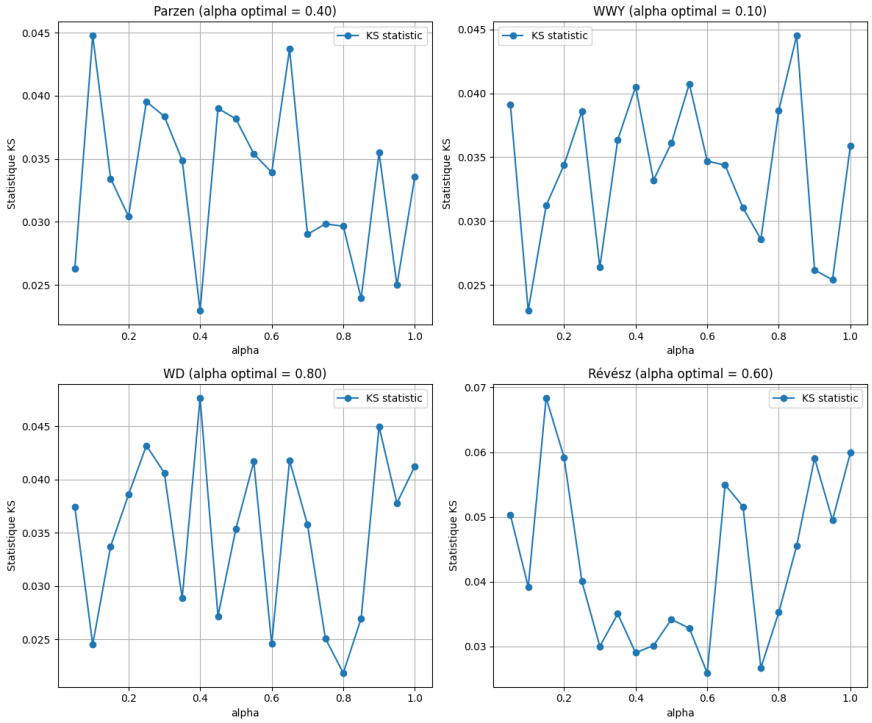
On voit bien que les 4 premiers estimateurs tendent vers la normalité asymptotique ce qui est est juste en théorie d'après la question 8)

1.5 Ainsi Parzen-Rosenblatt, par sa vitesse d'éxecution et ses caractéristiques de convergence, est le meilleur estimateur de cette étude.

[12]:
```
# Paramètres pour l'étude
alpha_values = np.linspace(0.05, 1.0, 20)  # Grille d'alphas à tester
n_fixed = 5000        # Taille fixe de l'échantillon
n_iter_norm = 300     # Nombre d'itérations (réplicats) pour l'estimation de la␣
 ↪distribution
x_test = 1.0          # Point d'évaluation (doit être dans le support de␣
 ↪l'exponentielle, ici x >= 0)
true_density = np.exp(-x_test)  # Densité théorique de l'exponentielle (λ = 1)␣
 ↪en x_test

# Dictionnaire des estimateurs (ils doivent être déjà définis dans le notebook)
estimators = {
    'Parzen': parzen_estimator,
    'WWY': estimator_WWY,
    'WD': estimator_WD,
    'Révész': estimator_R_iterative
}
```

```python
# Wrapper pour garantir une sortie scalaire pour un x donné
def safe_estimator(estimator, x, sample, **kwargs):
    res = estimator(np.array([x]), sample, **kwargs)
    return res[0] if isinstance(res, np.ndarray) else res

# Dictionnaire pour stocker l'alpha optimal pour chaque estimateur
optimal_alphas = {}

# Création de la figure avec un subplot par estimateur
fig, axs = plt.subplots(2, 2, figsize=(12, 10))
axs = axs.flatten()

for idx, (name, estimator) in enumerate(estimators.items()):
    ks_stats = []
    # Pour chaque valeur d'alpha, on évalue la normalité asymptotique
    for alpha in alpha_values:
        estimates = []
        for _ in range(n_iter_norm):
            # Génération d'un échantillon de la loi exponentielle (scale=1 <=>␣
 ↪λ=1)
            sample = np.random.exponential(scale=1, size=n_fixed)
            est = safe_estimator(estimator, x_test, sample, alpha=alpha)
            estimates.append(est)
        estimates = np.array(estimates)
        # Calcul de l'erreur d'estimation (différence par rapport à la densité␣
 ↪théorique)
        errors = estimates - true_density
        mean_error = np.mean(errors)
        std_error = np.std(errors)
        # Standardisation (si std_error > 0)
        if std_error == 0:
            errors_std = errors
        else:
            errors_std = (errors - mean_error) / std_error

        # Test KS de normalité (comparaison avec une loi N(0,1))
        ks_stat, _ = st.kstest(errors_std, 'norm')
        ks_stats.append(ks_stat)

    ks_stats = np.array(ks_stats)
    # L'alpha optimal minimise la statistique KS
    opt_alpha = alpha_values[np.argmin(ks_stats)]
    optimal_alphas[name] = opt_alpha

    ax = axs[idx]
    ax.plot(alpha_values, ks_stats, 'o-', label='KS statistic')
```

```
    ax.set_title(f"{name} (alpha optimal = {opt_alpha:.2f})")
    ax.set_xlabel("alpha")
    ax.set_ylabel("Statistique KS")
    ax.legend()
    ax.grid(True)

plt.tight_layout()
plt.show()

# Affichage des alphas optimaux pour chaque estimateur
for est_name, opt_a in optimal_alphas.items():
    print(f"Optimal alpha for {est_name}: {opt_a:.2f}")
```




```
Optimal alpha for Parzen: 0.40
Optimal alpha for WWY: 0.10
Optimal alpha for WD: 0.80
Optimal alpha for Révész: 0.60
```

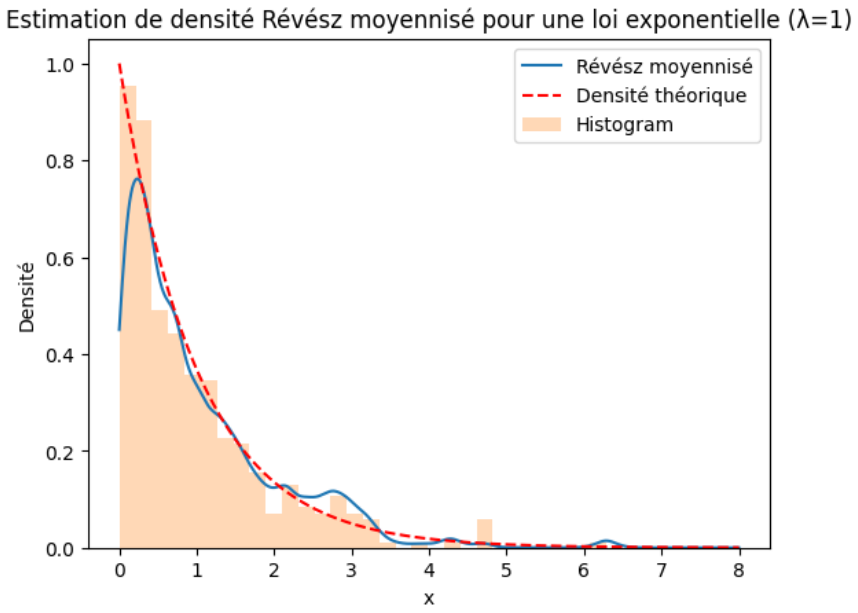
2 Passons à l'algorithme de Revesz moyénnisé

[13]:
```
def revesz_averaged_estimator(x, sample, gamma=0.395, sigma=1.0, alpha=0.6):
    """
    Estimateur de densité Révész moyennisé au point x.

    Pour un échantillon de taille n, on définit pour k = 1,...,n :
```

```
    - La fenêtre h_k = window(k, alpha) = k^(-alpha)
    - L'estimation de densité sur les k premières observations :
        f_k(x) = (1/(k * h_k)) * sum_{i=1}^{k} gaussian_kernel((x - X_i)/
↪h_k, sigma)
    - Le poids c_k = 1 / k^gamma

  L'estimateur final est la moyenne pondérée :
      f_n(x) = (sum_{k=1}^{n} c_k * f_k(x)) / (sum_{k=1}^{n} c_k)

  Args:
      x (float): Point d'évaluation.
      sample (array-like): Échantillon de données.
      gamma (float): Paramètre de pondération (défaut = 0.0).
      sigma (float): Paramètre du noyau gaussien (défaut = 1.0).
      alpha (float): Paramètre pour la fenêtre, via h_k = k^(-alpha) (défaut =␣
↪0.2).

  Returns:
      float: Estimation de la densité en x.
  """
  n = len(sample)
  if n == 0:
      return 0.0

  # Calcul des poids c_k et de leur somme
  c_vals = np.array([1.0 / (k ** gamma) for k in range(1, n + 1)])
  C_n = np.sum(c_vals)

  f_k_values = []
  for k in range(1, n + 1):
      h_k = window(k, alpha)
      sample_k = sample[:k]  # Les k premières observations
      u = (x - sample_k) / h_k
      # Application du noyau gaussien
      kernel_vals = gaussian_kernel(u, sigma)
      # Estimation de densité avec k observations
      f_k = np.sum(kernel_vals) / (k * h_k)
      f_k_values.append(f_k)

  f_k_values = np.array(f_k_values)
  # Moyenne pondérée des f_k(x) avec les poids c_k
  return np.sum(c_vals * f_k_values) / C_n
```

[14]:
```
# Génération d'un échantillon issu d'une loi exponentielle (λ=1)
sample = np.random.exponential(scale=1, size=400)

# Définition de la grille d'évaluation (support de l'exponentielle : x  0)
```

```
x_grid = np.linspace(0, 8, 1000)

# Mesure du temps pour le calcul de l'estimation de densité Révész moyennisé
start_time = time.perf_counter()
density = np.array([revesz_averaged_estimator(x, sample, gamma=0.3, sigma=1.0,␣
 ↪alpha=0.4) for x in x_grid])
end_time = time.perf_counter()

# Tracé de la densité estimée, de la densité théorique et de l'histogramme de␣
 ↪l'échantillon
plt.plot(x_grid, density, label="Révész moyennisé")
plt.plot(x_grid, np.exp(-x_grid), "r--", label="Densité théorique")  # Densité␣
 ↪théorique de l'exponentielle (λ=1)
plt.hist(sample, bins=30, density=True, alpha=0.3, label="Histogram")
plt.xlabel("x")
plt.ylabel("Densité")
plt.title("Estimation de densité Révész moyennisé pour une loi exponentielle␣
 ↪(λ=1)")
plt.legend()
plt.savefig("revesz_moyennise_exponentielle.png", dpi=300, bbox_inches="tight")
plt.show()


print(f"Temps Révész moyennisé: {end_time - start_time:.4f} secondes")
```

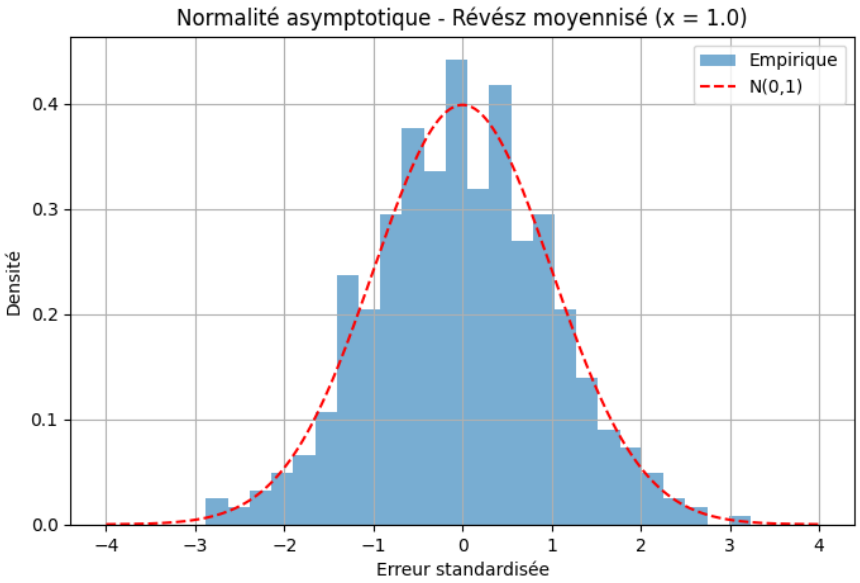


Temps Révész moyennisé: 5.7316 secondes

Le temps d'éxecution est relativement lent comparé aux précedents estimateurs

Regardons la normalité asymptotique de cet algorithme

[15]:
```
# Configuration pour l'étude de la normalité asymptotique avec l'estimateur␣
↪Révész moyennisé
n_fixed = 2000        # Taille fixe de l'échantillon
n_iter = 500          # Nombre de réplicats
x_test = 1.0          # Point d'évaluation (doit être dans le support de␣
↪l'exponentielle, ici x  0)
true_density = np.exp(-x_test)  # Densité théorique de l'exponentielle (λ = 1),␣
↪f(x)=exp(-x)

# Calcul des erreurs d'estimation sur n_iter réplicats
errors = []
for _ in range(n_iter):
    sample = np.random.exponential(scale=1, size=n_fixed)
    est = revesz_averaged_estimator(x_test, sample, gamma=0.3, sigma=1.0,␣
↪alpha=0.4)
    errors.append(est - true_density)
errors = np.array(errors)

# Standardisation des erreurs
mean_error = np.mean(errors)
std_error = np.std(errors)
errors_std = (errors - mean_error) / std_error if std_error > 0 else errors

# Visualisation de la distribution des erreurs standardisées
plt.figure(figsize=(8, 5))
plt.hist(errors_std, bins=25, density=True, alpha=0.6, label="Empirique")
x_vals = np.linspace(-4, 4, 200)
plt.plot(x_vals, norm.pdf(x_vals), 'r--', label="N(0,1)")
plt.xlabel("Erreur standardisée")
plt.ylabel("Densité")
plt.title("Normalité asymptotique - Révész moyennisé (x = {})".format(x_test))
plt.legend()
plt.grid(True)
plt.show()
```

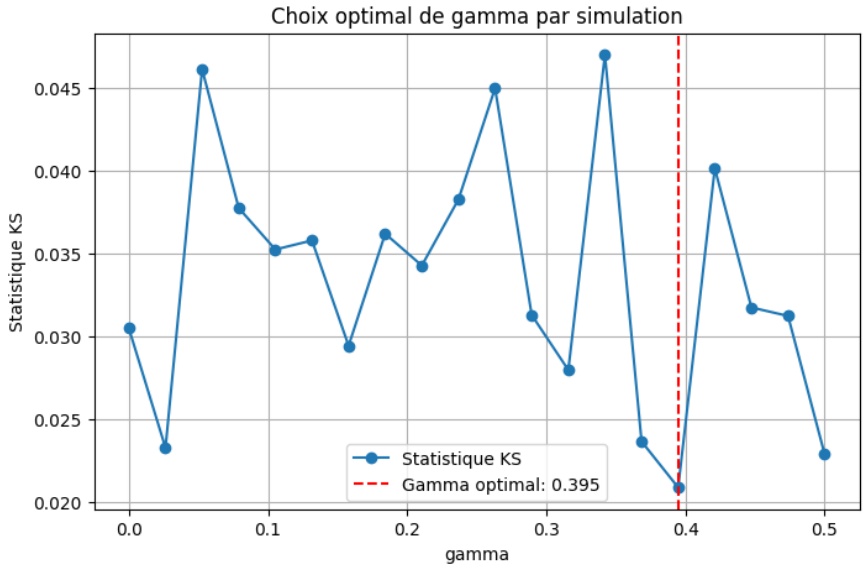


Cet algorithme vérifie bien la normalité asymptotique

2.1 Quelle valeur de gamma permet de minimiser l'erreur dans notre cas?

[16]:
```
# Grille de valeurs candidate pour gamma
gamma_values = np.linspace(0.0, 0.5, 20)
n_fixed = 2000        # Taille fixe de l'échantillon
n_iter = 500          # Nombre de réplicats
x_test = 1.0          # Point d'évaluation (pour l'exponentielle, x  0)
true_density = np.exp(-1)  # Densité théorique de l'exponentielle (λ=1) en x_test

ks_stats = []  # Pour stocker la statistique KS pour chaque gamma

for gamma in gamma_values:
    errors = []
    for _ in range(n_iter):
        # Générer un échantillon issu de la loi exponentielle (scale=1␣
↪correspond à λ=1)
        sample = np.random.exponential(scale=1, size=n_fixed)
        # Calcul de l'estimation de densité Révész moyennisé au point x_test␣
↪pour ce gamma
        est = revesz_averaged_estimator(x_test, sample, gamma=gamma, sigma=1.0,␣
↪alpha=0.4)
        errors.append(est - true_density)
    errors = np.array(errors)

    # Standardisation des erreurs
    mean_error = np.mean(errors)
    std_error = np.std(errors)
```

```
    if std_error > 0:
        errors_std = (errors - mean_error) / std_error
    else:
        errors_std = errors

    # Test de Kolmogorov-Smirnov pour comparer aux N(0,1)
    ks_stat, _ = st.kstest(errors_std, 'norm')
    ks_stats.append(ks_stat)

ks_stats = np.array(ks_stats)
optimal_gamma = gamma_values[np.argmin(ks_stats)]

plt.figure(figsize=(8,5))
plt.plot(gamma_values, ks_stats, 'o-', label="Statistique KS")
plt.axvline(optimal_gamma, color='r', linestyle='--',
            label=f"Gamma optimal: {optimal_gamma:.3f}")
plt.xlabel("gamma")
plt.ylabel("Statistique KS")
plt.title("Choix optimal de gamma par simulation")
plt.legend()
plt.grid(True)
plt.show()

print(f"gamma optimal: {optimal_gamma:.3f}")
```




```
Optimal gamma: 0.395
```

La statistique de Kolmogorov-Smirnov donne un gamma optimal à 0.4.