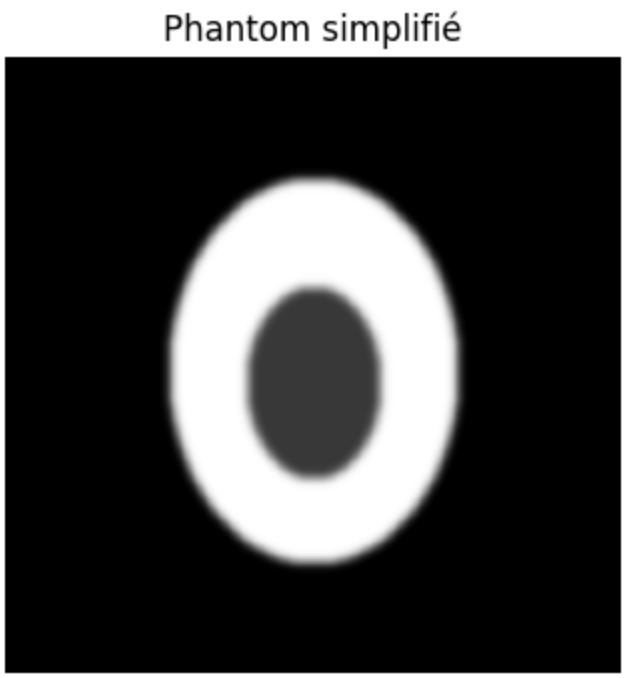
TP3 : Reconstruction d'images tomographiques

Nom et Prenom : [GHOMARI Alae]
Spécialité : [IAA]
Groupe : [02]

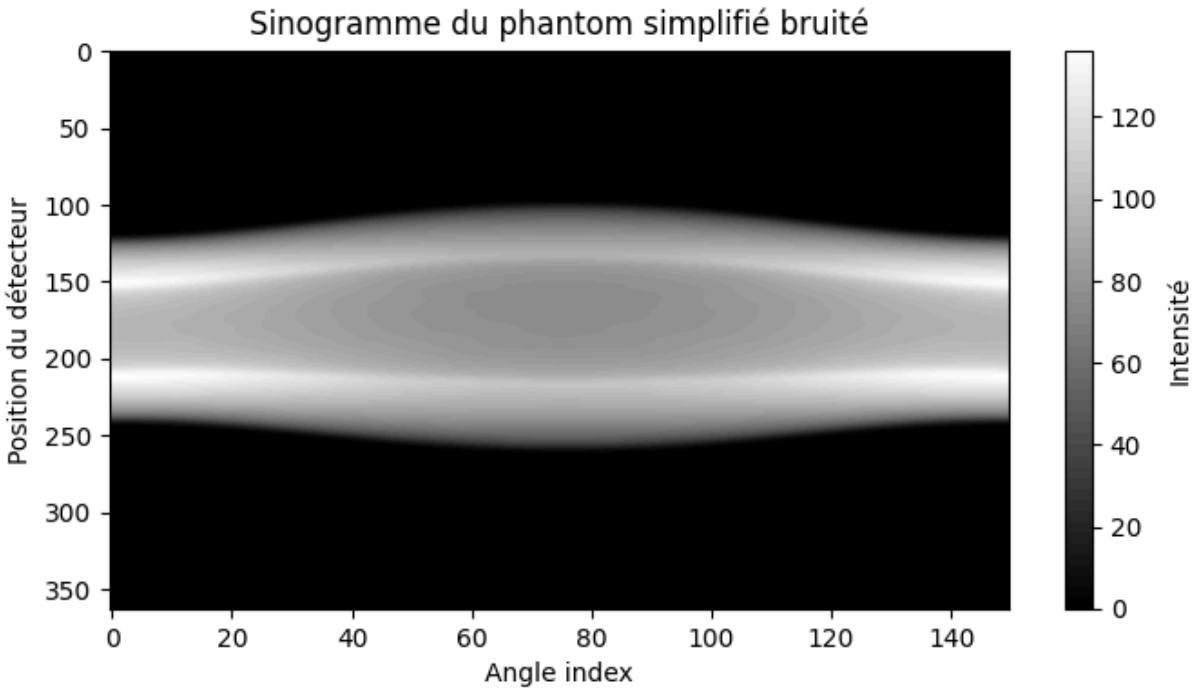
Creation du Phantom cardiaque

In [2]:
```python
# -----------------------
# Affichage du phantom simplifié
# -----------------------
import numpy as np
import matplotlib.pyplot as plt
from skimage.draw import ellipse
from scipy import ndimage as ndi

IMG_SIZE = 256

# Création du phantom
phantom = np.zeros((IMG_SIZE, IMG_SIZE), dtype=np.float32)

# Myocarde (grande ellipse)
rr, cc = ellipse(130, 128, 80, 60, shape=phantom.shape)
phantom[rr, cc] = 0.9

# Cavité ventriculaire (ellipse intérieure)
rr, cc = ellipse(135, 128, 40, 28, shape=phantom.shape)
phantom[rr, cc] = 0.2

# Lissage et normalisation
phantom = ndi.gaussian_filter(phantom, sigma=2)
phantom = phantom / phantom.max()

# Affichage
plt.figure(figsize=(4,4))
plt.imshow(phantom, cmap='gray')
plt.title("Phantom simplifié")
plt.axis('off')
plt.show()
```

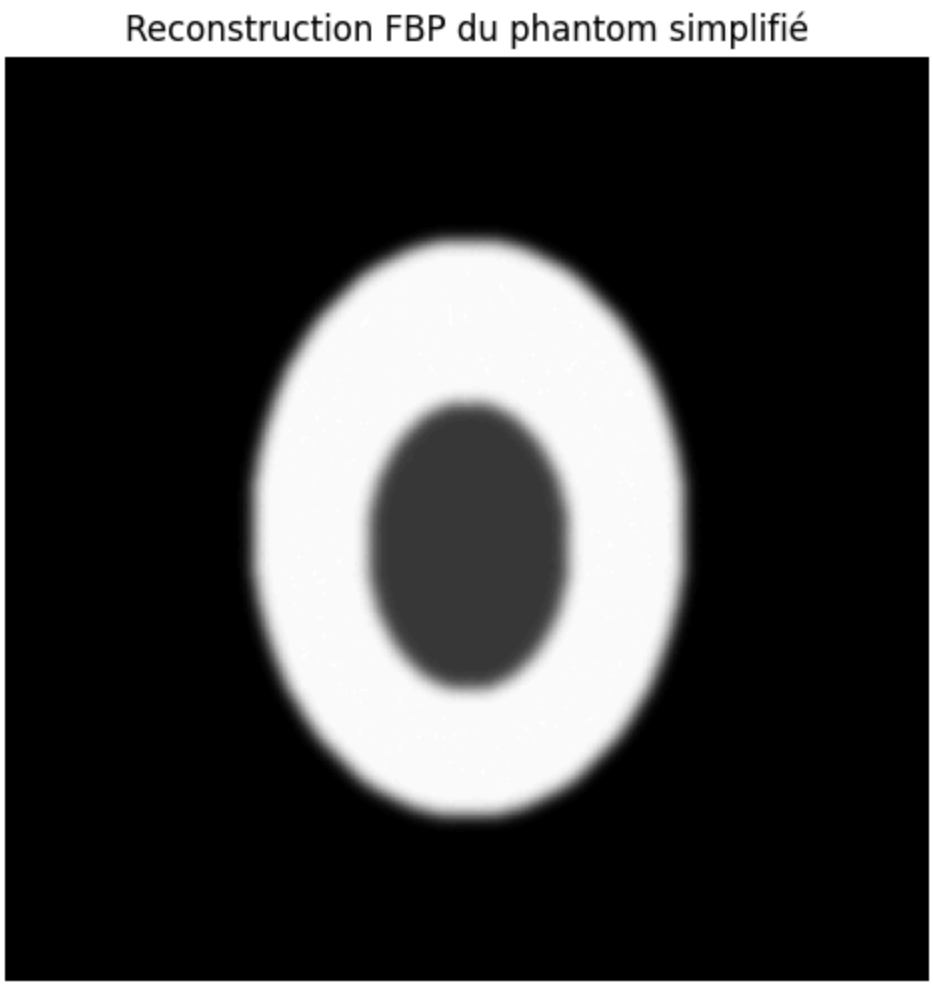


In [3]:
```python
# -----------------------
# 2) Ajouter du bruit de Poisson
# -----------------------
def add_poisson_noise_gray(image, scale=1e4, seed=None):
    if seed is not None:
        np.random.seed(seed)
    image = np.clip(image, 0, 1)
    counts = image * scale
    noisy = np.random.poisson(counts).astype(np.float32) / scale
    return np.clip(noisy, 0, 1)

noisy_phantom = add_poisson_noise_gray(phantom, scale=1e4, seed=42)

# -----------------------
# 3) Affichage
# -----------------------
plt.figure(figsize=(4,4))
plt.imshow(noisy_phantom, cmap='gray')
plt.title("Phantom simplifié avec bruit de Poisson")
plt.axis('off')
plt.show()
```

In [4]:
```python
from skimage.transform import radon
import numpy as np
import matplotlib.pyplot as plt

# --- Calcul du sinogramme ---
theta = np.linspace(0., 180., 150, endpoint=False)  # angles pour Radon
sinogram = radon(noisy_phantom, theta=theta, circle=False)

# --- Affichage du sinogramme ---
plt.figure(figsize=(8,4))
plt.imshow(sinogram, cmap='gray', aspect='auto')
plt.title("Sinogramme du phantom simplifié bruité")
plt.xlabel("Angle index")
plt.ylabel("Position du détecteur")
plt.colorbar(label='Intensité')
plt.show()
```

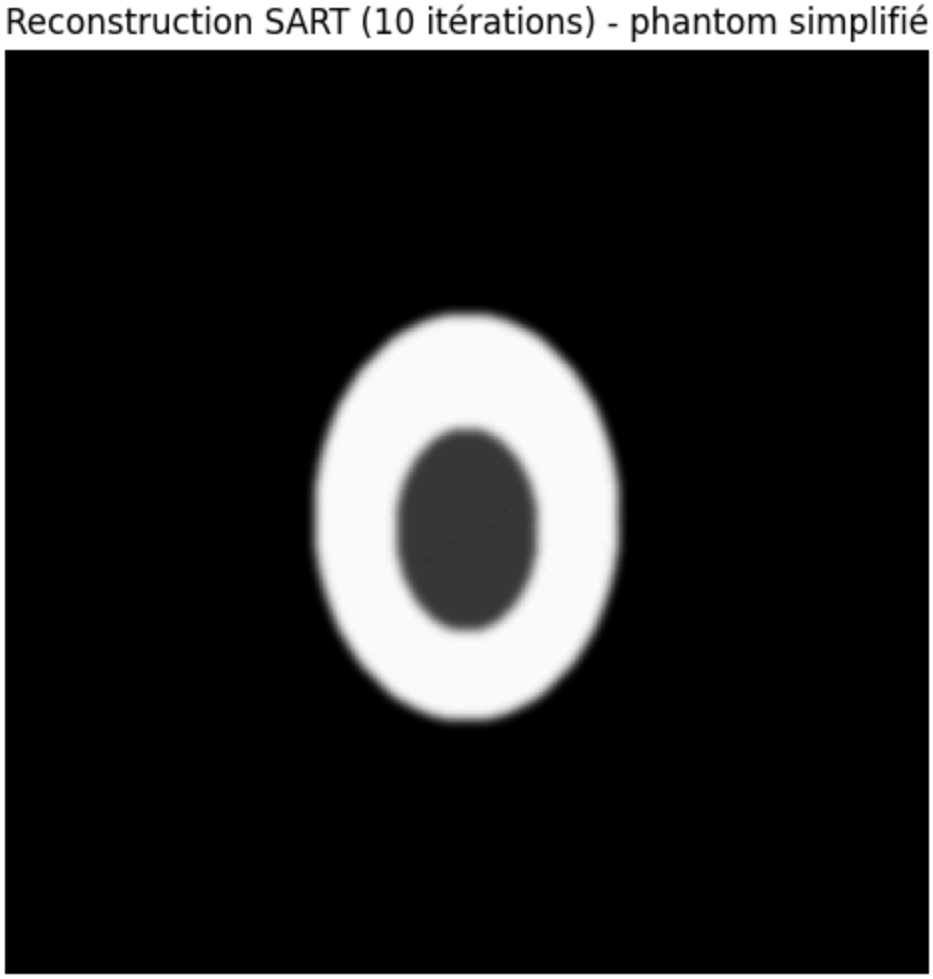


In [5]:
```
from skimage.transform import iradon, resize
import matplotlib.pyplot as plt

# --- Reconstruction par rétroprojection filtrée (FBP) ---
fbp_recon = iradon(sinogram, theta=theta, filter_name='ramp', circle=False)

# Redimensionner si nécessaire pour correspondre au phantom
fbp_recon = resize(fbp_recon, (noisy_phantom.shape[0], noisy_phantom.shape[1]), pre

# --- Affichage ---
plt.figure(figsize=(6,6))
plt.imshow(fbp_recon, cmap='gray')
plt.title("Reconstruction FBP du phantom simplifié")
plt.axis('off')
plt.show()
```



In [6]:
```
from skimage.transform import iradon_sart, resize
import numpy as np
import matplotlib.pyplot as plt

# --- Reconstruction SART ---
# Initialisation
radon_rows = sinogram.shape[0]
sart_img = np.zeros((radon_rows, radon_rows), dtype=np.float32)

# Itérations successives
SART_ITERS = 10
for i in range(SART_ITERS):
    sart_img = iradon_sart(sinogram, theta=theta, image=sart_img)

# Redimensionner pour correspondre au phantom
sart_recon = resize(sart_img, (noisy_phantom.shape[0], noisy_phantom.shape[1]), pre

# Normalisation
sart_recon = np.clip(sart_recon, 0, None)
if sart_recon.max() > 0:
    sart_recon = sart_recon / sart_recon.max()

# --- Affichage ---
plt.figure(figsize=(6,6))
plt.imshow(sart_recon, cmap='gray')
```

```
plt.title(f"Reconstruction SART ({SART_ITERS} itérations) - phantom simplifié")
plt.axis('off')
plt.show()
```



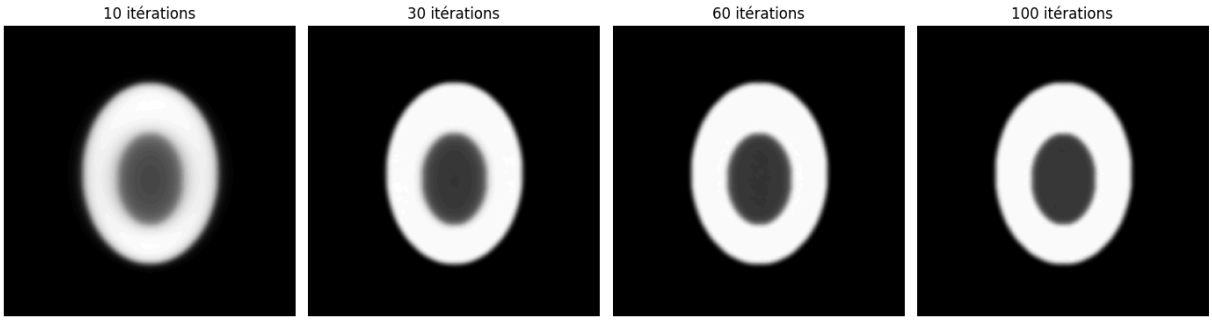
Reconstruction MLEM

Cette partie applique la méthode **MLEM** pour reconstruire l'image à partir du sinogramme(comme SART et FBP). On veut obtenir une image plus précise en améliorant la qualité de la reconstruction itérativement.

In [13]:
```
import numpy as np
import matplotlib.pyplot as plt
from skimage.transform import radon, iradon

# --- Paramètres ---
theta = np.linspace(0., 180., 150, endpoint=False)
eps = 1e-12

# --- Test de plusieurs nombres d'itérations ---
iterations = [10, 30, 60, 100]
reconstructions = []

for MLEM_ITERS in iterations:
    recon = np.ones_like(noisy_phantom)
```

```
    for it in range(MLEM_ITERS):
        proj_est = radon(recon, theta=theta, circle=False)
        ratio = sinogram / (proj_est + eps)
        backproj = iradon(ratio, theta=theta, filter_name=None, circle=False)
        recon *= backproj
        recon = np.clip(recon, 0, None)

    recon = recon / recon.max()
    reconstructions.append(recon)
    # On garde la dernière reconstruction (100 itérations)
    recon_mlem = reconstructions[-1]

# --- Affichage comparatif ---
plt.figure(figsize=(14, 6))
for i, (n, r) in enumerate(zip(iterations, reconstructions)):
    plt.subplot(1, len(iterations), i + 1)
    plt.imshow(r, cmap='gray')
    plt.title(f"{n} itérations")
    plt.axis('off')

plt.suptitle("Effet du nombre d'itérations sur la reconstruction MLEM", fontsize=14
plt.tight_layout()
plt.show()
```

Effet du nombre d'itérations sur la reconstruction MLEM



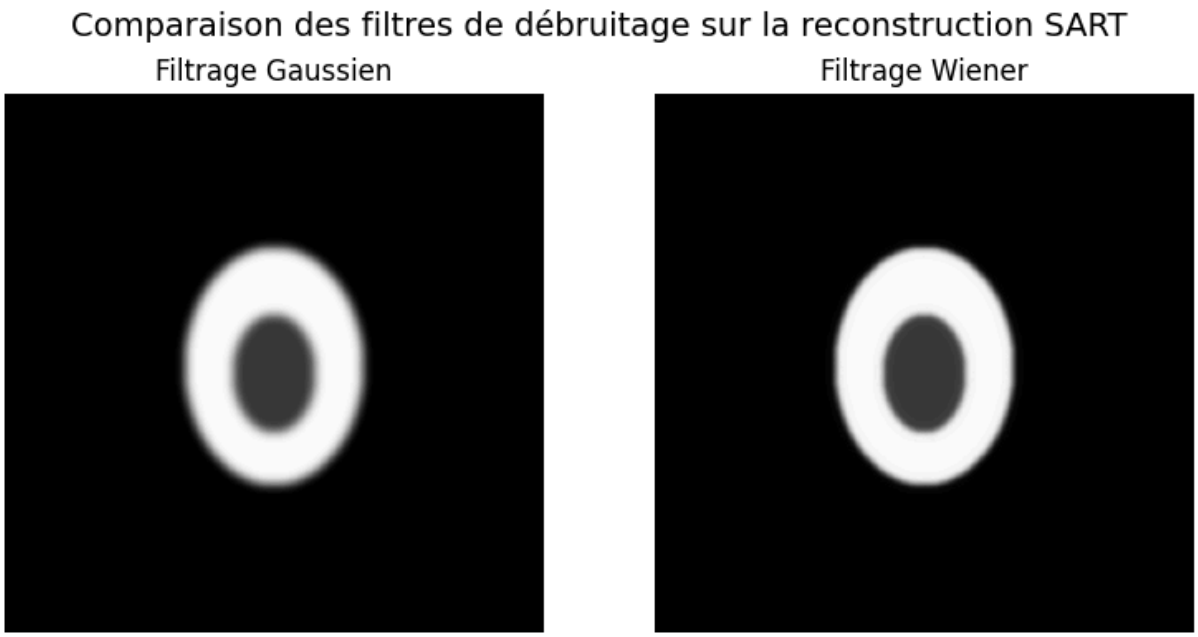
On remarque que MLEM donne une bonne reconstruction si on donne le nbre d itterations entre 50-60.

In [10]:
```
import numpy as np
from scipy.signal import fftconvolve
from scipy import ndimage

# --- Filtre gaussien ---
def gaussian_kernel(size=9, sigma=1.5):
    """
    Génère un noyau gaussien 2D normalisé.
    """
    ax = np.arange(-size//2 + 1., size//2 + 1.)
    xx, yy = np.meshgrid(ax, ax)
    kernel = np.exp(-(xx**2 + yy**2) / (2 * sigma**2))
    return kernel / np.sum(kernel)
```

```
def gaussian_denoise(img, size=9, sigma=1.5):
    """
    Applique un flou gaussien (filtrage passe-bas) pour réduire le bruit.
    """
    k = gaussian_kernel(size, sigma)
    return fftconvolve(img, k, mode='same')

# --- Filtre de Wiener ---
def wiener_denoise(img, mysize=7, noise_var=None):
    """
    Filtre de Wiener local.
    """
    local_mean = ndimage.uniform_filter(img, size=mysize)
    local_var = ndimage.uniform_filter(img**2, size=mysize) - local_mean**2

    if noise_var is None:
        noise_var = np.mean(local_var)

    result = local_mean + (np.maximum(local_var - noise_var, 0) / (local_var + 1e-8
    return np.clip(result, 0, 1)
```

In [14]:

```
# --- Application des deux méthodes de débruitage sur la reconstruction SART ---
sart_gauss = gaussian_denoise(sart_recon, sigma=2)
sart_wiener = wiener_denoise(sart_recon)

# --- Affichage comparatif ---
plt.figure(figsize=(8, 4))
titles = ["Filtrage Gaussien", "Filtrage Wiener"]

for i, img in enumerate([sart_gauss, sart_wiener]):
    plt.subplot(1, 2, i + 1)
    plt.imshow(img, cmap='gray', vmin=0, vmax=1)
    plt.title(titles[i])
    plt.axis('off')

plt.suptitle("Comparaison des filtres de débruitage sur la reconstruction SART", fo
plt.tight_layout()
plt.show()
```



In [15]:
```
from skimage.metrics import mean_squared_error, peak_signal_noise_ratio, structural
from skimage.transform import resize
import pandas as pd
import numpy as np

def compute_metrics(ref, img):
    """
    Calcule les métriques de similarité entre une image de référence (phantom)
    et une image reconstruite.
    """
    # Redimensionnement à la taille du phantom
    img_resized = resize(np.maximum(img, 0), ref.shape, mode='reflect', anti_aliasi

    # Calcul des métriques
    mse = mean_squared_error(ref, img_resized)
    mae = np.mean(np.abs(ref - img_resized))
    psnr = peak_signal_noise_ratio(ref, img_resized, data_range=ref.max() - ref.min
    ssim = structural_similarity(ref, img_resized, data_range=ref.max() - ref.min()
    rel_err = np.linalg.norm(ref - img_resized) / np.linalg.norm(ref)

    return mse, mae, psnr, ssim, rel_err

# --- Comparaison des différentes reconstructions ---
names = ["FBP", "SART", "SART + Gaussien", "SART + Wiener", "MLEM"]
images = [fbp_recon, sart_recon, sart_gauss, sart_wiener, recon_mlem]

results = [compute_metrics(phantom, im) for im in images]

# --- Tableau récapitulatif ---
df = pd.DataFrame(results, columns=["MSE", "MAE", "PSNR", "SSIM", "Erreur relative"
print("Métriques comparatives (plus proche de 0 = meilleure pour MSE et Erreur rela
display(df.round(4))
```

```
Métriques comparatives (plus proche de 0 = meilleure pour MSE et Erreur relative, pl
us élevé = meilleure pour PSNR et SSIM) :
```

	MSE	MAE	PSNR	SSIM	Erreur relative
FBP	0.0000	0.0016	50.1385	0.9954	0.0076
SART	0.1163	0.1363	9.3443	0.7494	0.8296
SART + Gaussien	0.1132	0.1362	9.4614	0.7447	0.8185
SART + Wiener	0.1155	0.1362	9.3745	0.7492	0.8267
MLEM	0.0000	0.0023	45.2562	0.9980	0.0133

Déduction

MLEM donne une image plus nette, plus fidèle et globalement meilleure que *SART* et *FBP*(un PSNR élevé et un SSIM proche de 1). (MSE et erreur relative) sont très faibles comparées à celles de SART, indiquant une reconstruction plus précise et moins bruitée. *FBP* reste rapide mais génère du bruit.
SART produit une image moins précise et plus floue, même après débruitage.

In [20]:
```
from skimage.filters import threshold_otsu
import matplotlib.pyplot as plt
import numpy as np

# --- Image utilisée(reconstruction MLEM) ---
img = recon_mlem

k = 0.8  #seuil d'Otsu(pour detection des zones froides)
seuil = threshold_otsu(img) * k

# --- Création du masque ---
zones_froides = img < seuil

# --- Affichage ---
plt.figure(figsize=(6,6))
plt.imshow(img, cmap='gray')
plt.imshow(zones_froides, cmap='cool', alpha=0.5)
plt.title(f"Zones froides détectées ")
plt.axis('off')
plt.show()
```

Détection des zones froides par seuillage d'intensité

On applique la méthode de *seuillage d'Otsu* sur l'image reconstruite par *MLEM* pour détecter automatiquement les zones froides (régions de faible intensité).
Les zones détectées sont affichées en *bleu* sur l'image.