Ralentissement

Collision élastique

1. Résoudre l'exercice `Diffusion_Elastique_ex.pdf` 1.1 Ecrire la vitesse du centre de masse. Exprimer celle-ci en fonction du ratio de masse atomique $A = M/m$.
   1.2 Exprimer la vitesse du neutron et du noyau dans le centre de masse que nous noterons respectivement w et W.
   1.3 Exprimez la relation entre l'énergie cinétique avant le choc dans le référentiel du laboratoire (E_L) et le référentiel du centre de masse (E_{B}).
   1.4 Dans le centre de masse calculez la quantité de mouvement totale.
   1.5 Dans le centre de masse, en appliquant les lois de conservations de la quantité de mouvement totale et de l'énergie cinétique, montrez que l'amplitude des vecteurs vitesses ne changent pas.
   1.6 Exprimez la vitesse après choc du neutron dans le laboratoire
2. Implémenter une fonction qui renvoit l'énergie de sortie et le cosinus de diffusion dans le laboratoire en fonction de l'énergie d'entrée, de l'angle de diffusion dans le centre de masse $\cos(\theta)$ et de A le ratio de masse atomique.
3. En supposant, la distribution $cos(\theta)$ uniforme entre [0, 1], tracez $E_{out} = f(\cos(\theta))$ pour $E_{in} = 1$ fixé et $\cos(\psi) = f(\cos(\theta))$ pour différentes valeurs de $A = [1,2,10,50,100,200,235]$

Commentez.

On commence par déterminer la vitesse du centre de masse B : $$ \vec{v_B} = \frac{m \vec{v_{in}}+M \vec{V_{in}}}{m+M} = \frac{1}{1+A} \vec{v_{in}} $$

car $ \vec{V_{in}}= \vec{0} $

On peut exprimer les vitesses du neutrons $\vec{w_{in}} $ et du noyau $\vec{W_{in}}$ dans le centre de masse $$\vec{w_{in}} = \vec{v_{in}} - \vec{v_{B}} = \vec{v_{in}} (1 - \frac{1}{1+A}) = \frac {A}{A+1} \vec{v_{in}} $$ $$\vec{W_{in}} = \vec{V_{in}} - \vec{v_{B}} = \vec{v_{in}} (\frac{A}{1+A} - 1) = - \frac {1}{A+1} \vec{v_{in}}$$

Dans le référentiel du laboratoire, l'énergie cinétique avant le choc vaut :

$$ E_c = E_{c1} + E_{c2} = \frac{1}{2} m v_{in}^2 $$

car $ E_{c2}=0 $ car $ \vec{V_{in}}= \vec{0} $

Dans le référentiel du centre de masse, l'énergie cinétique vaut:

$$ E_B = E_{c1}' + E_{c2}' = \frac{1}{2} m (\frac{A}{1+A})^2 v_{in}^2 + \frac{1}{2} M (\frac{1}{1+A})^2 v_{in}^2 = \frac {A}{A+1} E_c$$

On exprime la quantité de mouvement : $$ \vec{p_t} = m \vec{w_{in}} + M \vec{W_{in}} = \vec{0}$$

Conservation quantité de mouvement :

$$\frac{d\vec{p}}{dt} = \vec{0} \Rightarrow \vec{p} = \vec{cste} \Rightarrow m \vec{w_{out}} + M \vec{W_{out}} = \vec{0} = m \vec{w_{in}} + M \vec{W_{in}}$$

Conservation de l'énergie cinétique :

$$\frac{1}{2} m w_{in}^2 + \frac{1}{2} M W_{in}^2 = \frac{1}{2} m w_{out}^2 + \frac{1}{2} M W_{out}^2$$

Ce qui permet d'obtenir après manipulation :

$$ \begin{cases} w_{out}^2 = w_{in}^2 \\ W_{out}^2 = W_{in}^2 \end{cases} $$

On considère θ connu et l'on cherche $\vec{v_{out}}$. Pour cela il faut une amplitude et une direction
D'après la question 2 :

$$\vec{v_{out}} = \vec{w_{out}} + \vec{v_{B}} \Rightarrow v_{out}^2 = w_{out}^2 + v_{B}^2 + 2 \vec{w_{out}} \cdot \vec{v_{B}}$$

$$v_{out}^2 = v_{in}^2 [(\frac{A}{A+1})^2 + \frac{2A}{(A+1)^2} \cos(\theta) + \frac{1}{(1+A)^2}] = v_{in}^2[\frac{A^2 + 2A \cos(\theta) + 1}{(A+1)^2}]$$

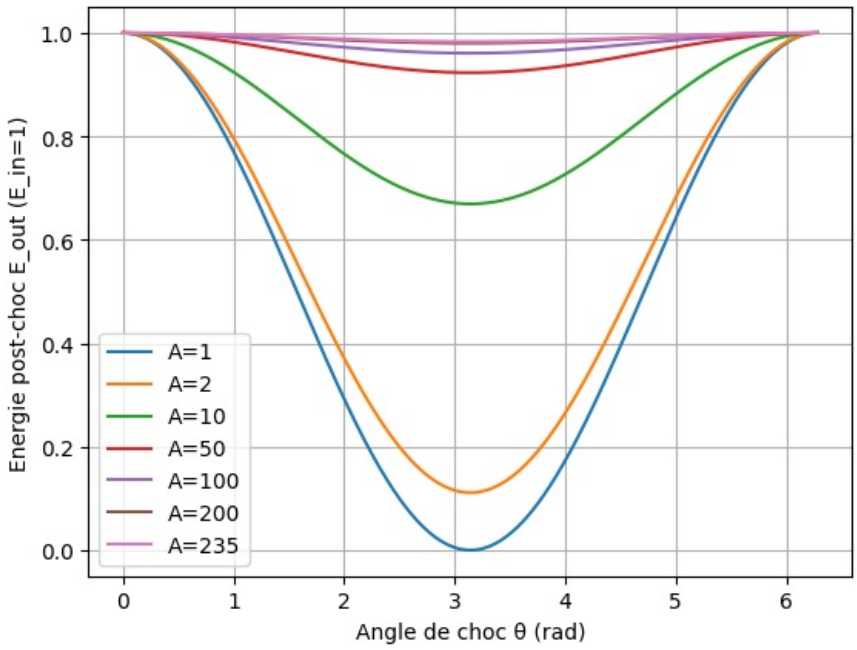
On calcule aussi le fraction d'éerngie cinétique en sortie par rapport à celle en entrée :

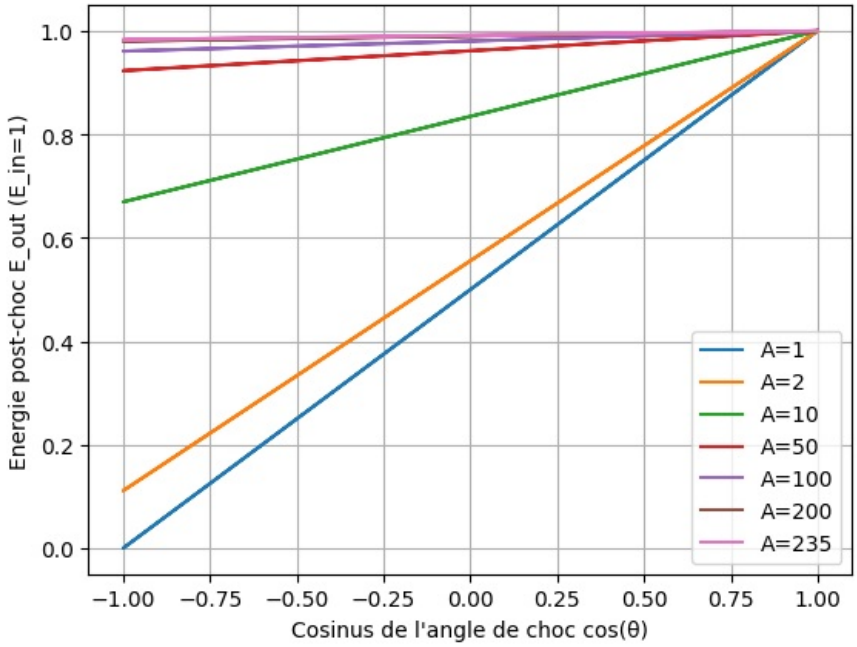
$$ \frac{E_{c_{out}}}{E_{c_{in}}} = \frac{1 + 2A \cos(\theta) + A^2}{(A+1)^2}$$

$$\frac{E_{out}}{E_{in}}=\frac{A^2+2A\mathrm{cos}(\theta)+1}{(A+1)^2}$$

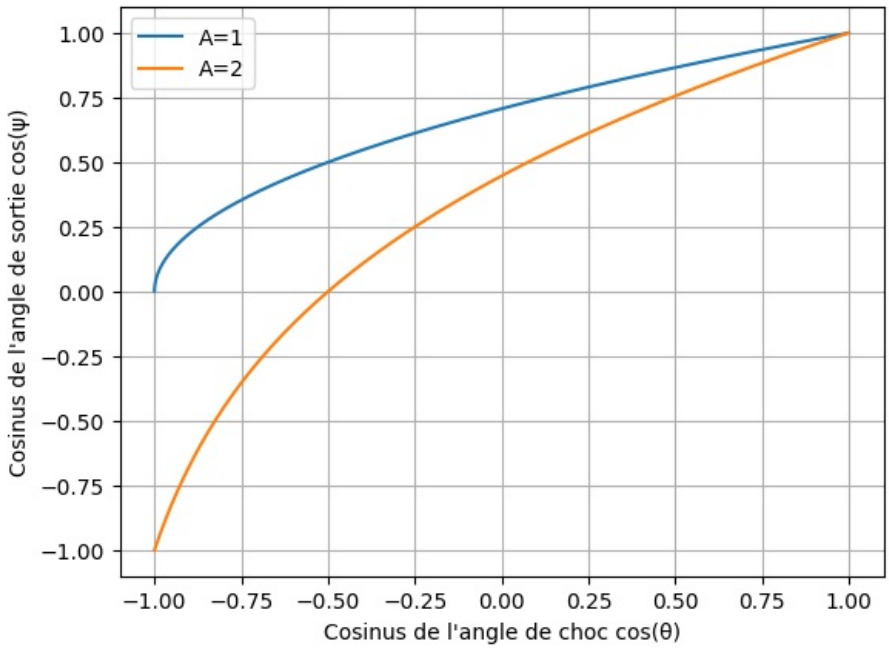
In [29]:
```
import numpy as np
```

```python
import matplotlib.pyplot as plt


def f(theta,Ein=1,A=1):
    """Calcul de l'énergie post-choc en fonction de l'angle de choc et du rapport de masse A

    Args:
        theta (int): _angle de choc_
        Ein (int, optional): Energie pré-choc. Defaults to 1.
        A (int, optional): Rapport de masse. Defaults to 1.

    Returns:
        int: Energie post-choc
    """
    Eout=Ein*(A**2+2*np.cos(theta)*A+1)/(1+A)**2
    return Eout

def cosPsi(cosTheta,A=1):
    return (1+A*cosTheta)/np.sqrt(A**2+2*A*cosTheta+1)


LA=[1,2,10,50,100,200,235]
Tx=np.linspace(0,2*np.pi,1000)

for A in LA:
    Ty=f(Tx,A=A)
    plt.plot(Tx, Ty, label='A={}'.format(A))
plt.legend()
plt.grid()
plt.xlabel('Angle de choc θ (rad)')
plt.ylabel('Energie post-choc E_out (E_in=1)')
plt.show()

for A in LA:
    plt.plot(np.cos(Tx), f(Tx,A=A), label='A={}'.format(A))
plt.legend()
plt.grid()
plt.xlabel('Cosinus de l\'angle de choc cos(θ)')
plt.ylabel('Energie post-choc E_out (E_in=1)')
plt.show()


Tx=np.linspace(0,np.pi,1000)
plt.plot(np.cos(Tx), cosPsi(np.cos(Tx)), label='A=1')
plt.plot(np.cos(Tx), cosPsi(np.cos(Tx),A=2), label='A=2')
plt.legend()
plt.grid()
plt.xlabel('Cosinus de l\'angle de choc cos(θ)')
plt.ylabel('Cosinus de l\'angle de sortie cos(ψ)')
plt.show()
```






```
/tmp/ipykernel_494902/2826665364.py:20: RuntimeWarning: invalid value encountered in divide
  return (1+A*cosTheta)/np.sqrt(A**2+2*A*cosTheta+1)
```



Minimale en $\theta=\pi$ et maximale en $\theta=0$. On remarque que plus A est grand (ie, plus il y a d'écart de masse entre le noyau et la masse d'un neutron), moins l'énergie de sortie diminue donc on a avantage à prendre des noyaux lourds.

Une approche Monte Carlo du ralentissement

Nous ne considérerons que la variable énergétique.
En partant d'une source située en $E = 1 MeV$,
modélisez via la méthode de Monte Carlo, le ralentissement dans un milieu ressemblant à l'hydrogène (collision élastique avec $A=1$) sans absorbant.

Considérez que $\Sigma_s(E) = 1.0$ et $\Sigma_a(E) = 0.0$.
Utilisez une source de 1E4 neutrons.
Tracer le spectre en fonction de l'énergie et comparer à la solution analytique.

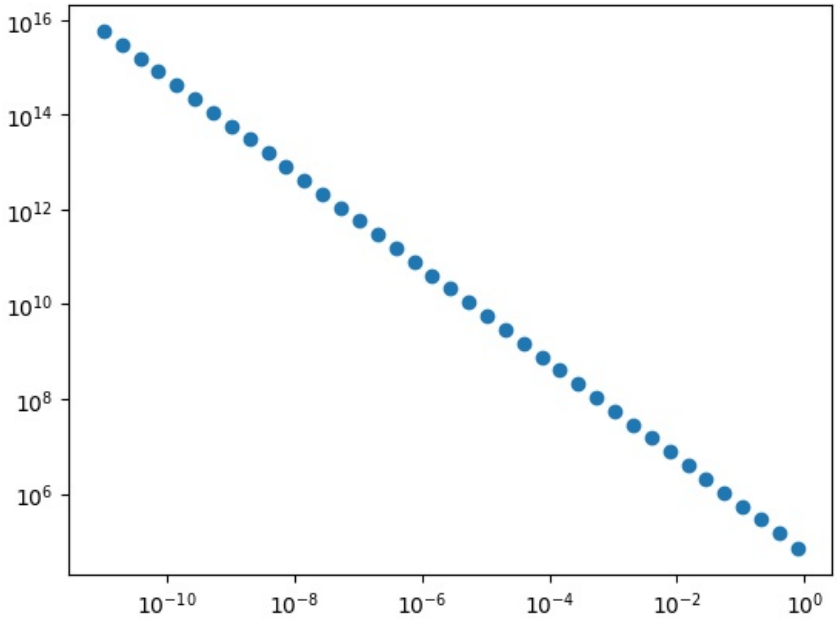
Faites varier le nombre de neutrons, faites varier A (par exemple $A=16$ correspond à l'O16).

Reprenez avec une section d'absorption non nulle.

Tracer le spectre en fonction de l'énergie et comparer à la solution analytique.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from dataclasses import dataclass, field

# @dataclass
# class NuclearData:
#     elastic_xs : float = 1.0
#     absorption_xs : float = 0.0 # Sigma_t
#     total_xs : float = field(init=False)
#     atomic_weight_ratio : float = 1.0 # A
#     def __post_init__(self):
#         self.total_xs = self.elastic_xs + self.absorption_xs

#     def get_total_xs(self, energy):
#         return self.total_xs

E_lower_limit = 1e-11
rng = np.random.default_rng(0) # séléction de la seed pour la reproductibilité
N_neutrons = 10000 # nombre de neutrons simulés
N_source=1e4 # nombre de neutrons émis par la source
E_source = 1.5 # énergie de la source en MeV

A=16
nb_intervalle=40
# intervalles=np.arange(0,E_source+taille_intervalle,taille_intervalle)
intervalles=np.logspace(np.log10(E_lower_limit),np.log10(E_source),nb_intervalle) # intervalles en logarithme
score_intervalle=np.zeros(len(intervalles) - 1)


alpha=((A-1)/(A+1))**2
print(alpha)
n=0
sigma_t=1
# Monte Carlo : on lance N_neutrons neutrons
for i in range(N_neutrons):
    energie=E_source # énergie initiale du neutron
    while energie>E_lower_limit: # tant que l'énergie du neutron est supérieure à la limite inférieure
        n+=1
        energie=rng.uniform(alpha*energie,energie) # énergie post-choc entre alpha*E et E
        index_intervalle = np.searchsorted(intervalles, energie) - 1 # trouver l'index de l'intervalle correspo
        if 0 <= index_intervalle < len(score_intervalle):
            score_intervalle[index_intervalle] += 1/sigma_t # incrémenter le score de l'intervalle correspondan

score_intervalle*=N_source/N_neutrons # normalisation par le nombre de neutrons émis et simulés

print('Nombre moyen de collisions par neutron:',n/N_neutrons)
score_intervalle /= np.diff(intervalles)

plt.loglog(intervalles[:-1],score_intervalle,'o')
plt.show()
```

```
0.7785467128027681
Nombre moyen de collisions par neutron: 215.2896
```

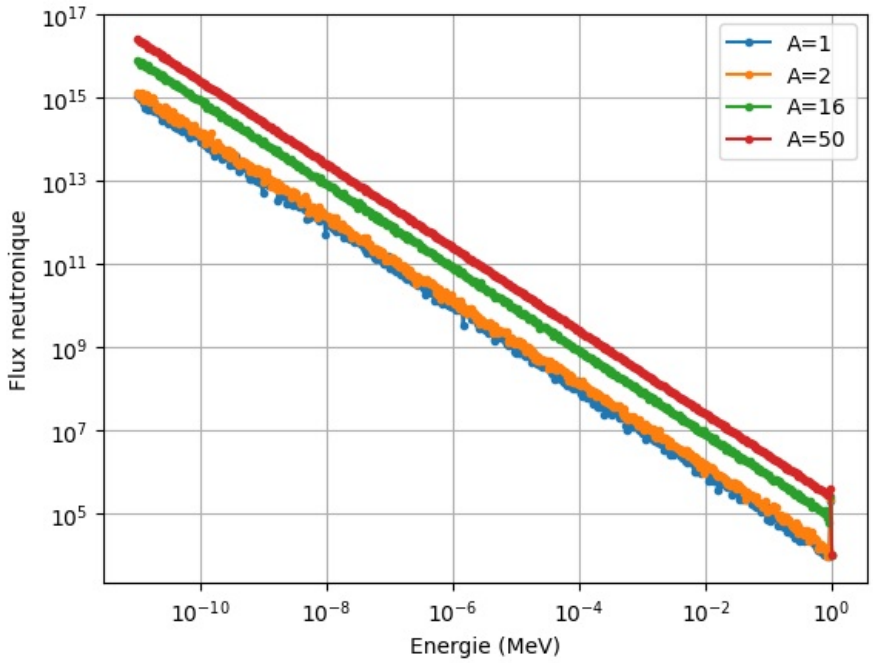


Si on n'a pas d'absorption, le flux est linéaire (en échelle log-log) par rapport à l'énergie et décroit.

Parallélisation de calcul pour optimiser le temps de calcul

In [1]:
```
import multiprocessing
from multiprocessing import Pool
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm
processes=multiprocessing.cpu_count()

def evolution_neutron(args):
    """Simule la vie d'un neutron

    Args:
        args (tuple): E_source,E_lower_limit,alpha,intervalles
        E_source (float): Energie initiale du neutron
        E_lower_limit (float): Energie minimale du neutron
        alpha (float): Coefficient de ralentissement
        intervalles (np.array): Intervalles d'énergie

    Returns:
        np.array: Score local des intervalles d'énergie
    """
    E_source, E_lower_limit, alpha, intervalles = args
    energie=E_source
    score_local = np.zeros(len(intervalles))
    score_local[-1] = 1

    while energie>E_lower_limit:
        index_intervalle = np.searchsorted(intervalles, energie) - 1
        if 0 <= index_intervalle < len(score_local):
            score_local[index_intervalle] += 1
        energie=rng.uniform(alpha*energie,energie)
    return(score_local)


# Initialisation
E_lower_limit = 1e-11
rng = np.random.default_rng(0)
N_neutrons = 10000
E_source = 1.0

N_source = 1e4

nb_intervalle=500
intervalles=np.logspace(np.log10(E_lower_limit),np.log10(E_source),nb_intervalle) # intervalles en logarithme

LA=[1,2,16,50]
# LA=[1,16]
L_phi_MC=[]
# Boucle sur les différents rapports de masse
for A in LA:
    alpha=((A-1)/(A+1))**2

    sigma_t=1.0
    # Monte Carlo : on lance N_neutrons neutrons
    # On parallélise le calcul sur le nombre de processeurs disponibles (16 pour mon ordinateur) ce qui accélèr
    # tqdm permt l'affichage de l'évolution des calculs en temps réel
    with Pool(processes) as pool:
        results = list(tqdm(pool.imap(evolution_neutron, [(E_source, E_lower_limit, alpha, intervalles)] * N_neu


    # Calcul du score total
    score_total = np.sum(results, axis=0)
    score_total[:-1] /= np.diff(intervalles)
    score_total*= sigma_t*N_source/N_neutrons

    plt.loglog(intervalles,score_total,'.-',label='A={}'.format(A)) # Affichage en log-log
    # plt.semilogx(intervalles[:-1],score_total,'o-',label='A={}'.format(A))
    L_phi_MC.append(score_total)
plt.legend()
plt.grid()
plt.xlabel('Energie (MeV)')
plt.ylabel('Flux neutronique')
plt.show()

intervalles_MC=intervalles
```

```
100%|██████████| 10000/10000 [00:01<00:00, 6935.61it/s]
100%|██████████| 10000/10000 [00:01<00:00, 6971.14it/s]
100%|██████████| 10000/10000 [00:03<00:00, 3079.41it/s]
100%|██████████| 10000/10000 [00:07<00:00, 1387.17it/s]
```



Quand on trace pour différentes valeurs de A (ie différents atomes), on garde la même pentes, il y a juste l'ordonnée à l'origine qui croit quand A croit.

Section d'absorption non nulle

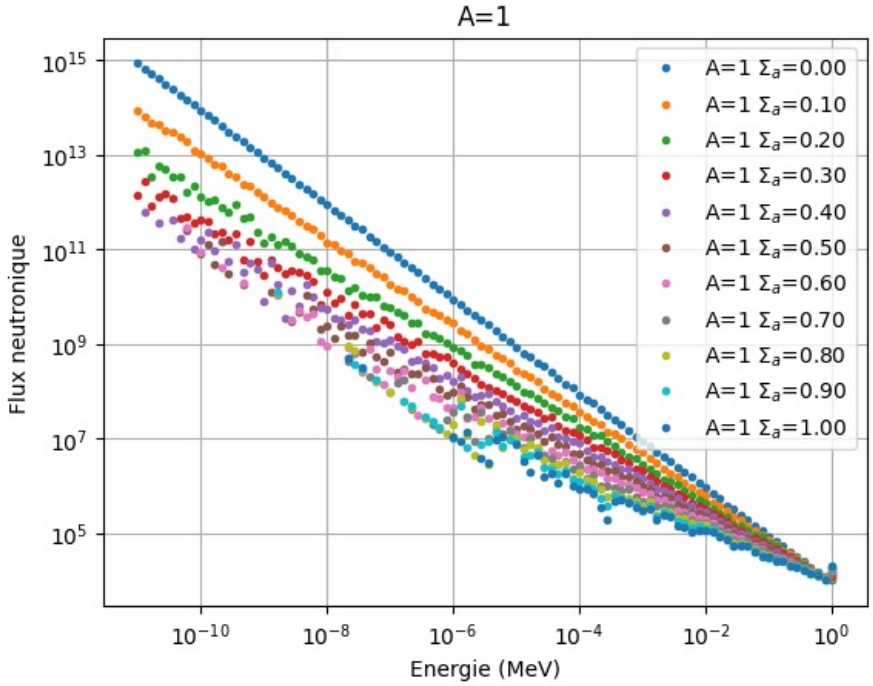
On ajoute de l'absorption, c'est à dire qu'a chaque pas, il y a une probabilité que la particule disparaisse.

In [3]:
```
import multiprocessing
from multiprocessing import Pool
import numpy as np
import matplotlib.pyplot as plt
processes=multiprocessing.cpu_count()


def evolution_neutron(args):
    """Simule la vie d'un neutron

    Args:
        args (tuple): E_source,E_lower_limit,alpha,intervalles
        E_source (float): Energie initiale du neutron
        E_lower_limit (float): Energie minimale du neutron
        alpha (float): Coefficient de ralentissement
        intervalles (np.array): Intervalles d'énergie
        Proba_absorption (float): Probabilité d'absorption par collision

    Returns:
        np.array: Score local des intervalles d'énergie
    """
    E_source, E_lower_limit, alpha, intervalles, Proba_absorption = args
    energie=E_source
    score_local = np.zeros(len(intervalles))

    score_local[-1] += 1  # Compter le neutron initial dans le dernier intervalle

    while energie>E_lower_limit:
        if rng.uniform(0,1) < Proba_absorption: # On absorbe le neutron avec la probabilité donnée
            break # Si il est absorbé, on arrête la simulation de ce neutron
        energie=rng.uniform(alpha*energie,energie)
        index_intervalle = np.searchsorted(intervalles, energie) - 1
        if 0 <= index_intervalle < len(score_local):
            score_local[index_intervalle] += 1
    return(score_local)


E_lower_limit = 1e-11
rng = np.random.default_rng(0)
N_neutrons = 100000
E_source = 1.0
Proba_absorption=0.5

N_source = 1e4
```

```python
nb_intervalle=100
intervalles=np.logspace(np.log10(E_lower_limit),np.log10(E_source),nb_intervalle) # intervalles en logarithme
intervalles_MC_abs=intervalles


LA=[1,16]
L_sigma_a=list(np.arange(0,1.1,0.1))

L_phi_MC_abs=[]
for A in LA:
    print("A =", A)
    # Boucle sur plusieurs sigma_a ce qui permet de simuler plusieurs proba d'absorption
    for sigma_a in L_sigma_a:
        alpha=((A-1)/(A+1))**2

        sigma_s=1.0
        sigma_t=sigma_s+sigma_a

        Proba_absorption = sigma_a / sigma_t

        with Pool(processes) as pool:
            results = list(tqdm(pool.imap(evolution_neutron, [(E_source, E_lower_limit, alpha, intervalles, Prob


        score_total = np.sum(results, axis=0)
        score_total[:-1] /= np.diff(intervalles)
        score_total*= sigma_t*N_source/N_neutrons


        plt.loglog(intervalles,score_total,'.',label='A={} $ \Sigma_a $={:.2f}'.format(A, sigma_a))
        # plt.plot(intervalles[:-1],score_total,'o-',label='A={} P_abs={:.1f}'.format(A, Proba_absorption))
        # plt.semilogx(intervalles[:-1],score_total,'o-',label='A={}'.format(A))
        L_phi_MC_abs.append(score_total)
    plt.title('A={}'.format(A))
    plt.legend()
    plt.grid()
    plt.xlabel('Energie (MeV)')
    plt.ylabel('Flux neutronique')
    plt.show(block=False)
```

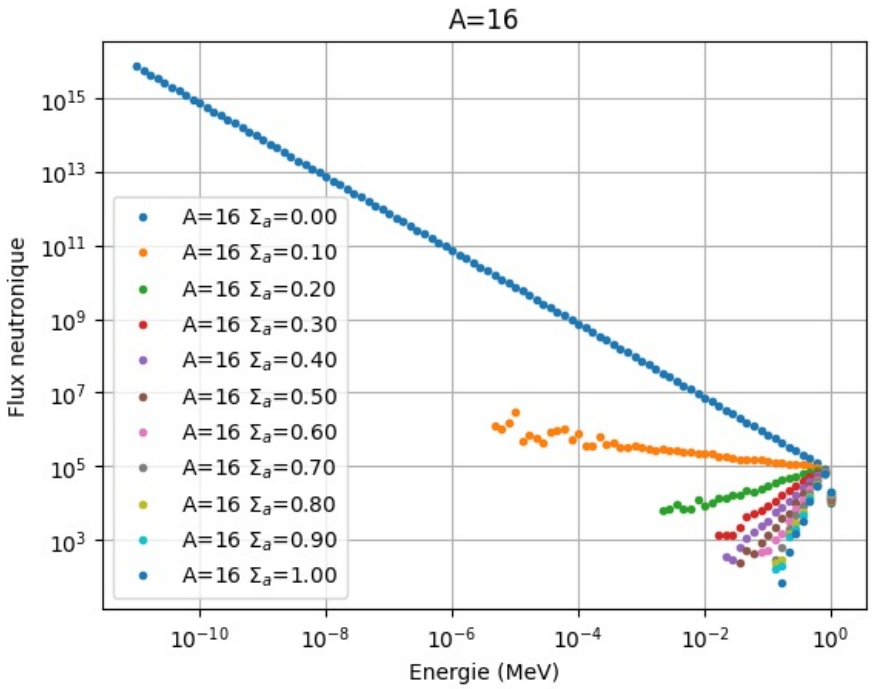
```
A = 1
```

```
<>:75: SyntaxWarning: invalid escape sequence '\S'
<>:75: SyntaxWarning: invalid escape sequence '\S'
/tmp/ipykernel_545418/2699198017.py:75: SyntaxWarning: invalid escape sequence '\S'
  plt.loglog(intervalles,score_total,'.',label='A={} $ \Sigma_a $={:.2f}'.format(A, sigma_a))
100%|██████████| 100000/100000 [00:12<00:00, 8034.54it/s]
100%|██████████| 100000/100000 [00:12<00:00, 8330.05it/s]
100%|██████████| 100000/100000 [00:12<00:00, 7962.58it/s]
100%|██████████| 100000/100000 [00:11<00:00, 8740.17it/s]
100%|██████████| 100000/100000 [00:11<00:00, 8762.85it/s]
100%|██████████| 100000/100000 [00:11<00:00, 8892.16it/s]
100%|██████████| 100000/100000 [00:11<00:00, 9077.13it/s]
100%|██████████| 100000/100000 [00:11<00:00, 8837.04it/s]
100%|██████████| 100000/100000 [00:10<00:00, 9193.08it/s]
100%|██████████| 100000/100000 [00:10<00:00, 9421.32it/s]
100%|██████████| 100000/100000 [00:07<00:00, 13668.56it/s]
```



A = 16

```
100%|██████████| 100000/100000 [00:29<00:00, 3405.87it/s]
100%|██████████| 100000/100000 [00:06<00:00, 15994.08it/s]
100%|██████████| 100000/100000 [00:11<00:00, 9004.91it/s]
100%|██████████| 100000/100000 [00:10<00:00, 9817.07it/s]
100%|██████████| 100000/100000 [00:07<00:00, 14269.26it/s]
100%|██████████| 100000/100000 [00:10<00:00, 9672.31it/s]
100%|██████████| 100000/100000 [00:05<00:00, 18708.18it/s]
100%|██████████| 100000/100000 [00:05<00:00, 18772.60it/s]
100%|██████████| 100000/100000 [00:05<00:00, 19618.99it/s]
100%|██████████| 100000/100000 [00:05<00:00, 18914.66it/s]
100%|██████████| 100000/100000 [00:05<00:00, 19102.39it/s]
```



Comme les particules sont absobé, on a moins de particules qui vont aux énergies très faibles donc on n'arrive pas à déterminer la courbe jusqu'au bout. On remarque que l'absorption fait diminuer le flux (ce qui est logique). L'absorption a plus d'impact si A est grand car on va sur des énergies entre αE et E et si $A=1$, $\alpha=0$ et donc on peut aller entre 0 et E.

Analytique

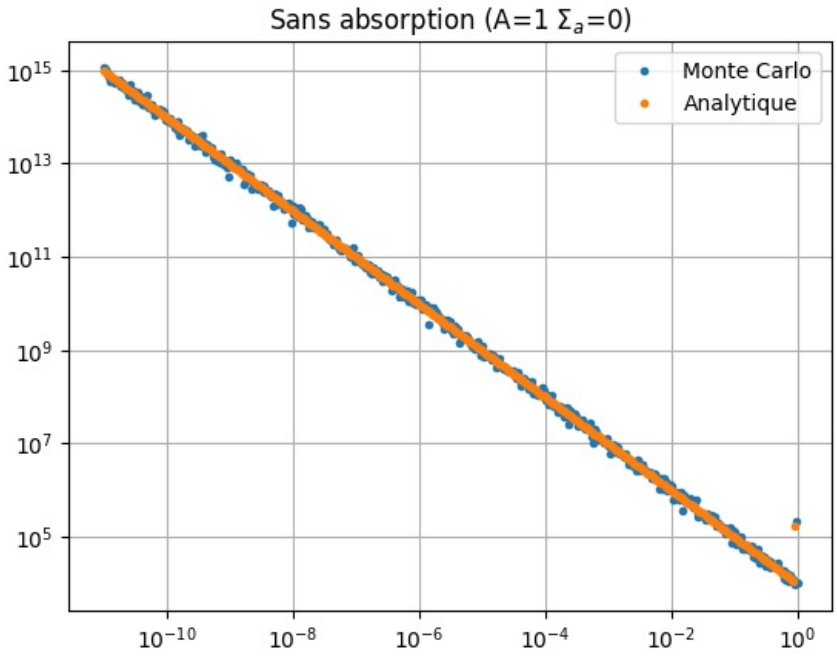
Nous allons maintenant étudier les fonctions analytiques et les comparés à la méthodes de Monte-Carlo.

Analytique A=1 et $\Sigma_a=0$

In [8]:
```
import numpy as np
import matplotlib.pyplot as plt

E_lower_limit = 1e-11
E_source = 1.0
Proba_absorption=0.5
N_source=1e4

nb_intervalle=400
intervalles=np.logspace(np.log10(E_lower_limit),np.log10(E_source),nb_intervalle) # intervalles en logarithme

score = np.zeros(len(intervalles) - 1)

score[-1] = 1e4
score/=np.diff(intervalles)

A=1
alpha=((A-1)/(A+1))**2
sigma_a=0
sigma_s=1.0
sigma_t=sigma_s/ (sigma_s+sigma_a)

i=len(score)-2
```

```
# On fait le calcul analytique avec les formules du cours
while i>=0:
    i_min=i+1

    energie=(intervalles[i+1]+intervalles[i])/2

    score[i]= N_source/sigma_s /energie
    i-=1

# On affiche la solution de MC et celle analytique (on a besoin d'avoir recharger la cellule du MC pour affiche
plt.loglog(intervalles_MC,L_phi_MC[0],'.',label="Monte Carlo")
plt.loglog(intervalles[:-1],score,'.',label="Analytique")
plt.title('Sans absorption (A={} $\Sigma_a$={})'.format(A, sigma_a))
plt.legend()
plt.grid()
plt.show()
```

```
<>:37: SyntaxWarning: invalid escape sequence '\S'
<>:37: SyntaxWarning: invalid escape sequence '\S'
/tmp/ipykernel_545418/1613758324.py:37: SyntaxWarning: invalid escape sequence '\S'
  plt.title('Sans absorption (A={} $\Sigma_a$={})'.format(A, sigma_a))
```



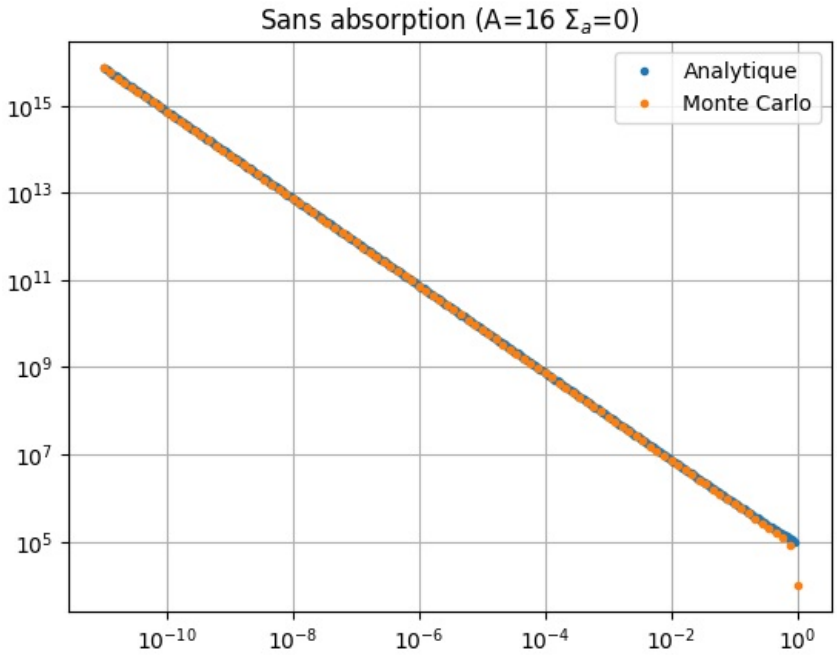
Analytique A=16 et $\Sigma_a=0$ avec la formule exacte

In [9]:

```
import numpy as np
import matplotlib.pyplot as plt

E_lower_limit = 1e-11
E_source = 1.0
N_source = 1e5

nb_intervalle=400
intervalles=np.logspace(np.log10(E_lower_limit),np.log10(E_source),nb_intervalle) # intervalles en logarithme

score = np.zeros(len(intervalles) - 1)

score[-1] = 1

A=16
alpha=((A-1)/(A+1))**2
sigma_a=0
sigma_s=1.0
sigma_t=sigma_s/ (sigma_s+sigma_a)

i=len(score)-2
while i>=0:

    energie=(intervalles[i+1]+intervalles[i])/2/alpha

    ksi=1-alpha/(1-alpha)*(intervalles[i+1]-intervalles[i])
    score[i]= 1/sigma_s /energie/ksi
    i-=1

score*=N_source
plt.loglog(intervalles[:-1],score,'.',label="Analytique")
```

```
plt.loglog(intervalles_MC_abs,L_phi_MC_abs[11],'.',label="Monte Carlo")
plt.title('Sans absorption (A={} $\Sigma_a$={})'.format(A, sigma_a))
plt.legend()
plt.grid()
plt.show()
```

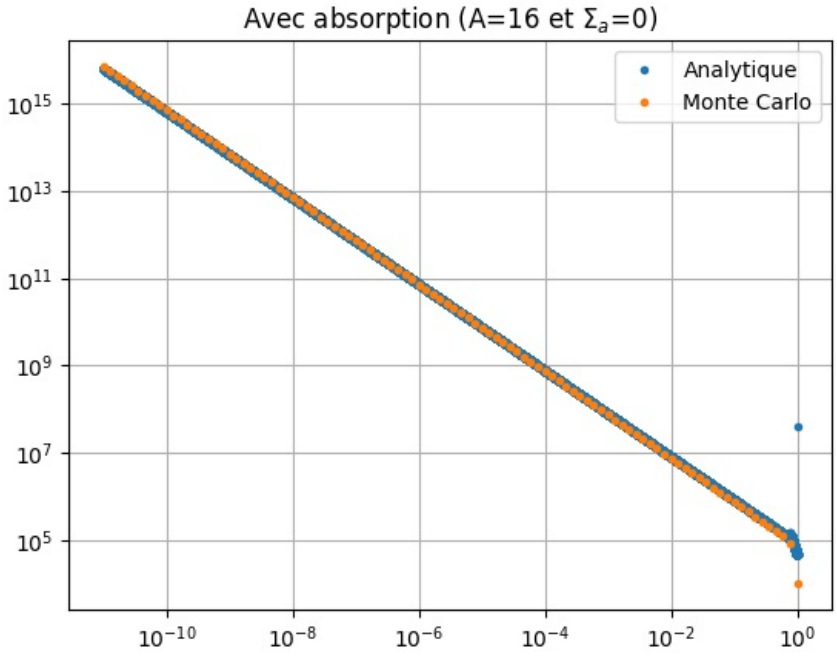
```
<>:33: SyntaxWarning: invalid escape sequence '\S'
<>:33: SyntaxWarning: invalid escape sequence '\S'
/tmp/ipykernel_545418/209692416.py:33: SyntaxWarning: invalid escape sequence '\S'
  plt.title('Sans absorption (A={} $\Sigma_a$={})'.format(A, sigma_a))
```



Analytique A=16 et $\Sigma_a=0$ avec l'approximation de l'intégrale du cas 4

In [10]:
```
E_lower_limit = 1e-11
E_source = 1.0
N_source = 1e4

nb_intervalle=100000
intervalles=np.logspace(np.log10(E_lower_limit),np.log10(E_source),nb_intervalle) # intervalles en logarithme

score = np.zeros(len(intervalles) - 1)

score[-1] = 1e4
score/=np.diff(intervalles)

A=16
alpha=((A-1)/(A+1))**2
sigma_a=0
sigma_s=1.0
sigma_t=sigma_s+sigma_a


i=len(score)-2
while i>=0:
    i_min=i+1

    energie=(intervalles[i+1]+intervalles[i])/2/alpha

    i_max= np.searchsorted(intervalles, energie)-1
    score[i]= np.sum(score[i_min:i_max]/intervalles[i_min:i_max]*np.diff(intervalles)[i_min:i_max])/(1-alpha)*s
    # print(i_max-i_min)
    i-=1
# score*=N_source
plt.loglog(intervalles[:-1],score,'.',label="Analytique")
plt.loglog(intervalles_MC_abs,L_phi_MC_abs[11],'.',label="Monte Carlo")
plt.title('Avec absorption (A={} et $\Sigma_a$={})'.format(A, sigma_a))
plt.legend()
plt.grid()
plt.show()
```

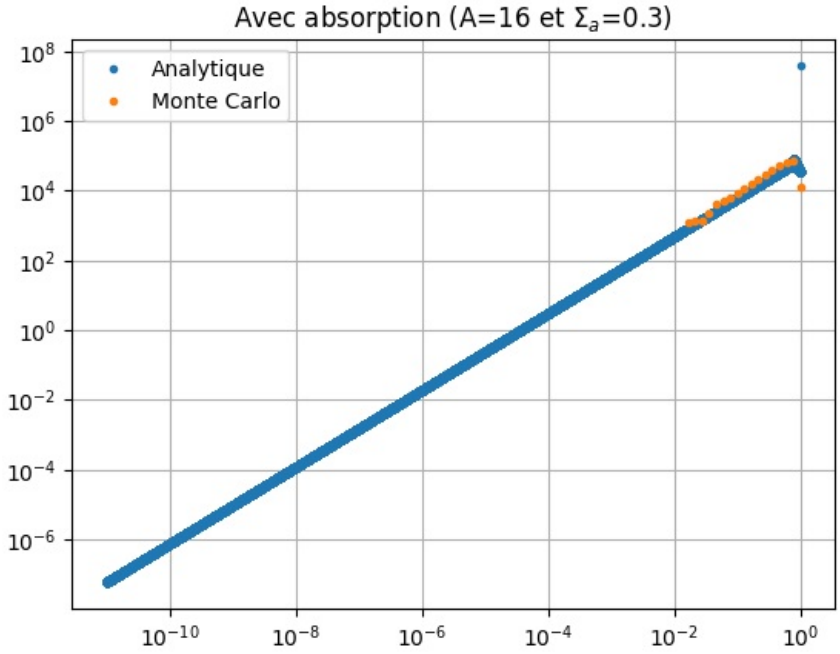
```
<>:33: SyntaxWarning: invalid escape sequence '\S'
<>:33: SyntaxWarning: invalid escape sequence '\S'
/tmp/ipykernel_545418/2564433712.py:33: SyntaxWarning: invalid escape sequence '\S'
  plt.title('Avec absorption (A={} et $\Sigma_a$={})'.format(A, sigma_a))
```



Analytique A=16 et $\Sigma_a=0.3$

In [35]:
```
import numpy as np
import matplotlib.pyplot as plt

E_lower_limit = 1e-11
E_source = 1.0
N_source = 1e4

nb_intervalle=100000
intervalles=np.logspace(np.log10(E_lower_limit),np.log10(E_source),nb_intervalle) # intervalles en logarithme

score = np.zeros(len(intervalles) - 1)

score[-1] = 1e4
score/=np.diff(intervalles)

A=16
alpha=((A-1)/(A+1))**2
sigma_a=0.3
sigma_s=1.0
sigma_t=sigma_s+sigma_a


i=len(score)-2
while i>=0:
    i_min=i+1

    energie=(intervalles[i+1]+intervalles[i])/2/alpha

    i_max= np.searchsorted(intervalles, energie)-1
    score[i]= np.sum(score[i_min:i_max]/intervalles[i_min:i_max]*np.diff(intervalles)[i_min:i_max])/(1-alpha)*s
    # print(i_max-i_min)
    i-=1
# score*=N_source
plt.loglog(intervalles[:-1],score,'.',label="Analytique")
plt.loglog(intervalles_MC_abs,L_phi_MC_abs[11+3],'.',label="Monte Carlo")
plt.title('Avec absorption (A={} et $\Sigma_a$={})'.format(A, sigma_a))
plt.legend()
plt.grid()
plt.show()
```

```
<>:36: SyntaxWarning: invalid escape sequence '\S'
<>:36: SyntaxWarning: invalid escape sequence '\S'
/tmp/ipykernel_494902/948714840.py:36: SyntaxWarning: invalid escape sequence '\S'
  plt.title('Avec absorption (A={} et $\Sigma_a$={})'.format(A, sigma_a))
```



Dans tous les cas analytiques, nous retrouvons les courbes issus de Monte-Carlo mais avec une meilleurs précision et un temps de calcul plus faible. Malgré que le cas 4 ne soit pas exactement une solution analytique, on retrouve la solution de MC et on arrive à aller bien plus loin dans les énergies.