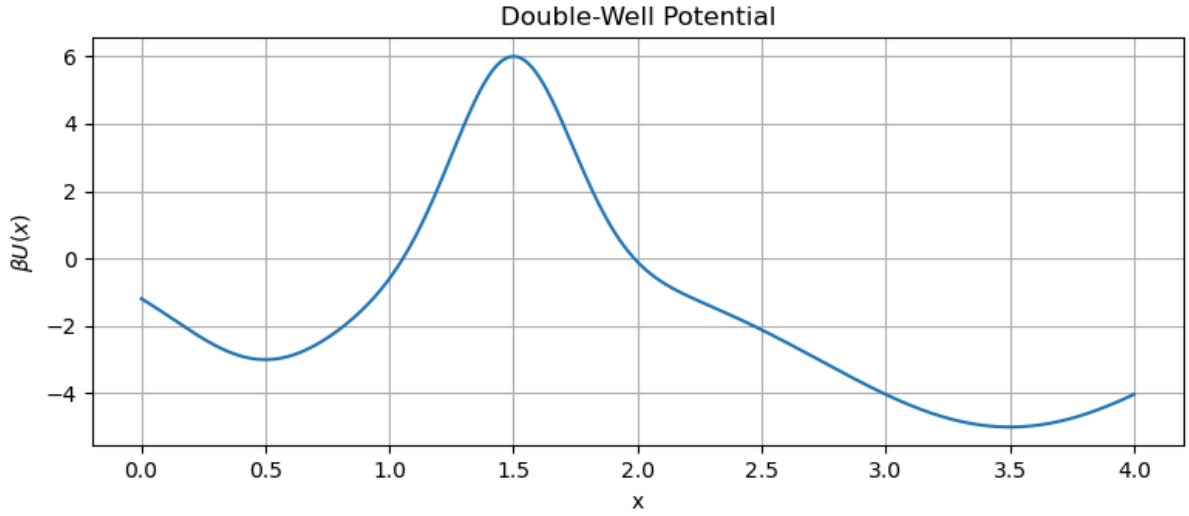
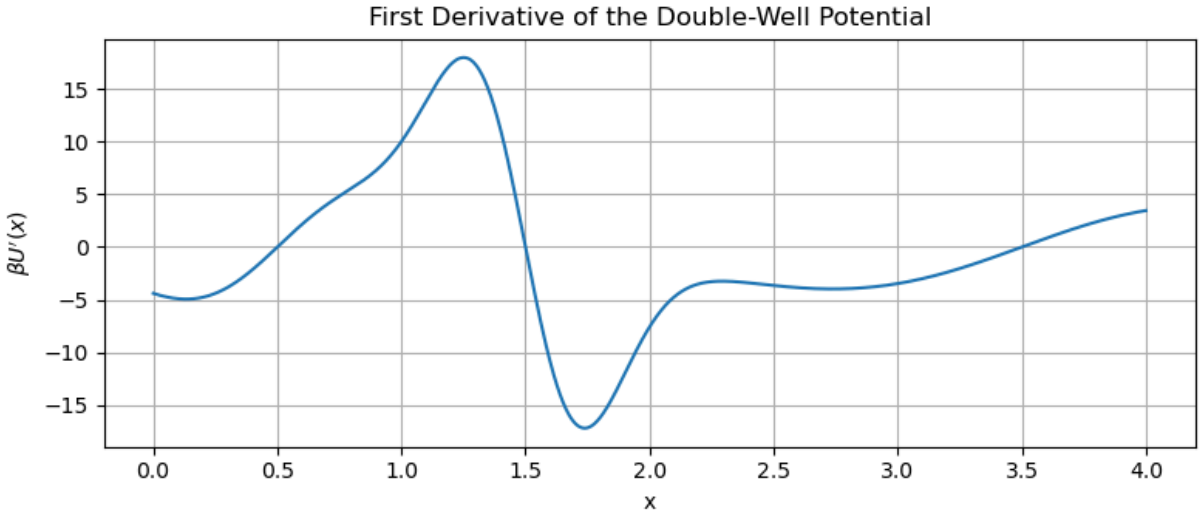
In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.sparse.linalg import spsolve
from scipy.integrate import quad
from scipy.optimize import fsolve
from scipy.interpolate import interp1d, PchipInterpolator
from scipy.optimize import fsolve
```

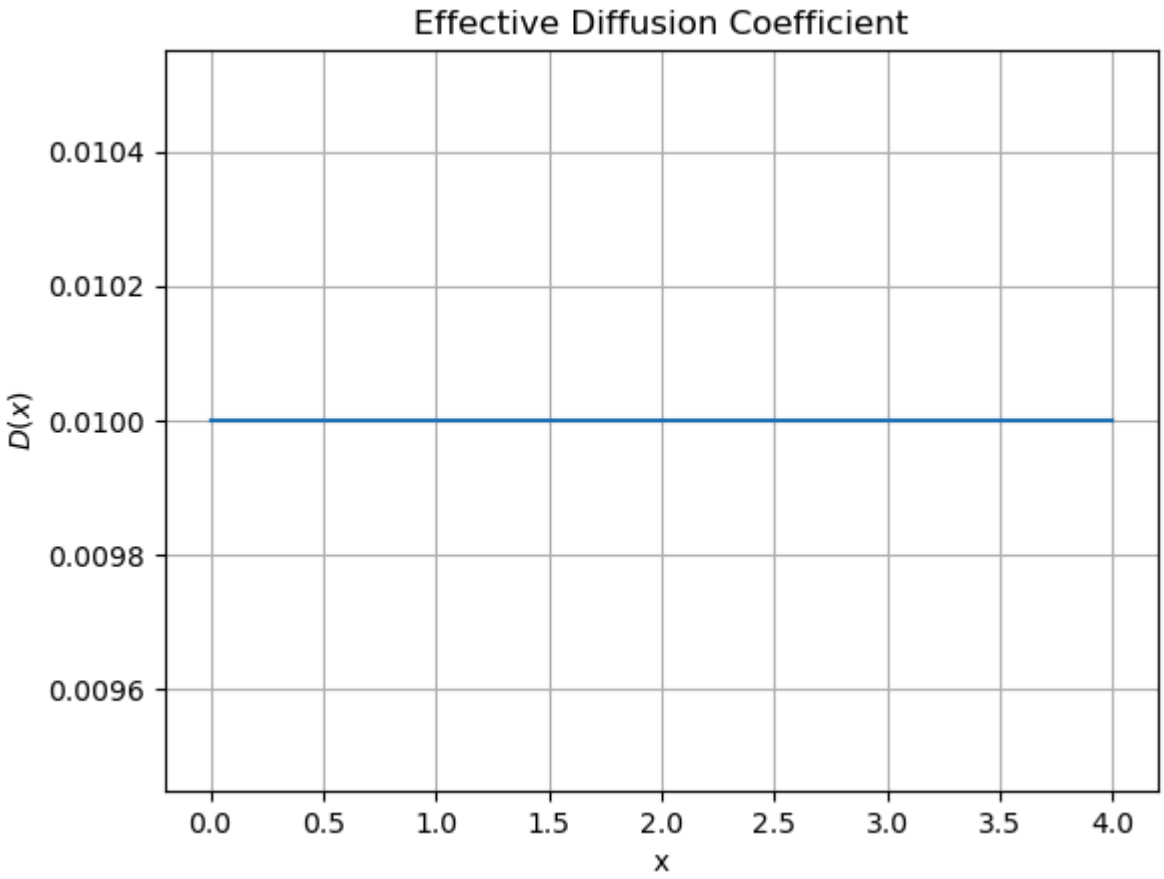
In [2]:
```
def double_well_potential(x1, x2, xm, V1, V2, Vm):
    # System of equations to determine A1, A2, Am, w1, w2, wm
    def equations(vars):
        A1, A2, Am, w1, w2, wm = vars
        eq1 = A1 * np.exp(-((x1 - x1) / w1) ** 2) + A2 * np.exp(-((x1 - x2)
        eq2 = A1 * np.exp(-((x2 - x1) / w1) ** 2) + A2 * np.exp(-((x2 - x2)
        eq3 = A1 * np.exp(-((xm - x1) / w1) ** 2) + A2 * np.exp(-((xm - x2)
        eq4 = -2 * A1 * (xm - x1) / (w1 ** 2) * np.exp(-((xm - x1) / w1) **
        eq5 = -2 * A1 * (x1 - x1) / (w1 ** 2) * np.exp(-((x1 - x1) / w1) **
        eq6 = -2 * A1 * (x2 - x1) / (w1 ** 2) * np.exp(-((x2 - x1) / w1) **
        return [eq1, eq2, eq3, eq4, eq5, eq6]

    # Initial guess for A1, A2, Am, w1, w2, wm
    initial_guess = [V1, V2, Vm, 0.5, 1.0, 0.6]

    # Solve the system of equations
    A1, A2, Am, w1, w2, wm = fsolve(equations, initial_guess)
    print(f"A1={A1}, A2={A2}, Am={Am}, w1={w1}, wm={wm}, w2={w2}")

    # Define the potential function
    def V(x):
        return A1 * np.exp(-((x - x1) / w1) ** 2) + A2 * np.exp(-((x - x2) /

    # Define the first derivative of the potential function
    def dV_dx(x):
        term1 = -2 * A1 * (x - x1) / (w1 ** 2) * np.exp(-((x - x1) / w1) **
        term2 = -2 * A2 * (x - x2) / (w2 ** 2) * np.exp(-((x - x2) / w2) **
        term3 = -2 * Am * (x - xm) / (wm ** 2) * np.exp(-((x - xm) / wm) **
        return term1 + term2 + term3

    return V, dV_dx

# Parameters
x1 = 0.5    # Location of the first minimum
xm = 1.5     # Location of the maximum (barrier)
x2 = 3.5     # Location of the second minimum
V1 = -3.0    # Value at the first minimum
V2 = -5.0  # Value at the second minimum
Vm = 6.0   # Value at the maximum (barrier height)


# Calculate the potential and its derivative
beta_U, beta_Up = double_well_potential(x1, x2, xm, V1, V2, Vm)

# x range for plotting
x = np.linspace(0, 4, 1000)
```

```
potential = beta_U(x)
dV_dx = beta_Up(x)

# Plotting the potential
plt.figure(figsize=(8, 7))
plt.subplot(2, 1, 1)
plt.plot(x, potential)
plt.xlabel('x')
plt.ylabel("$ \\beta U(x)$")
plt.title('Double-Well Potential')
plt.grid(True)

# Plotting the first derivative of the potential
plt.subplot(2, 1, 2)
plt.plot(x, dV_dx)
plt.xlabel('x')
plt.ylabel("$ \\beta U'(x)$")
plt.title("First Derivative of the Double-Well Potential")
plt.grid(True)

plt.tight_layout()
plt.show()

D0 = 0.01
def D(x):
    # return D0*x**(2/3)
    return D0*x**0

plt.plot(x, D(x))
plt.xlabel('x')
plt.ylabel("$ D(x)$")
plt.title('Effective Diffusion Coefficient')
plt.grid(True)
plt.show()
```

```
A1=-2.9984306623906316, A2=-5.0000000000002, Am=6.235107362393493, w1=0.5204
21938809253, wm=0.32906358123352797, w2=1.0784000138556546
```


Double-Well Potential
$\beta U(x)$
6
4
2
0
−2
−4
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
x


First Derivative of the Double-Well Potential
$\beta U'(x)$
15
10
5
0
−5
−10
−15
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
x



In [3]:
```
a = 0.5   # location of reflecting boundary
b = 3.5  # location of absorbing boundary

h = 0.01
N = int((b-a)/h+1)

x_arr = np.linspace(a, b, N)
```

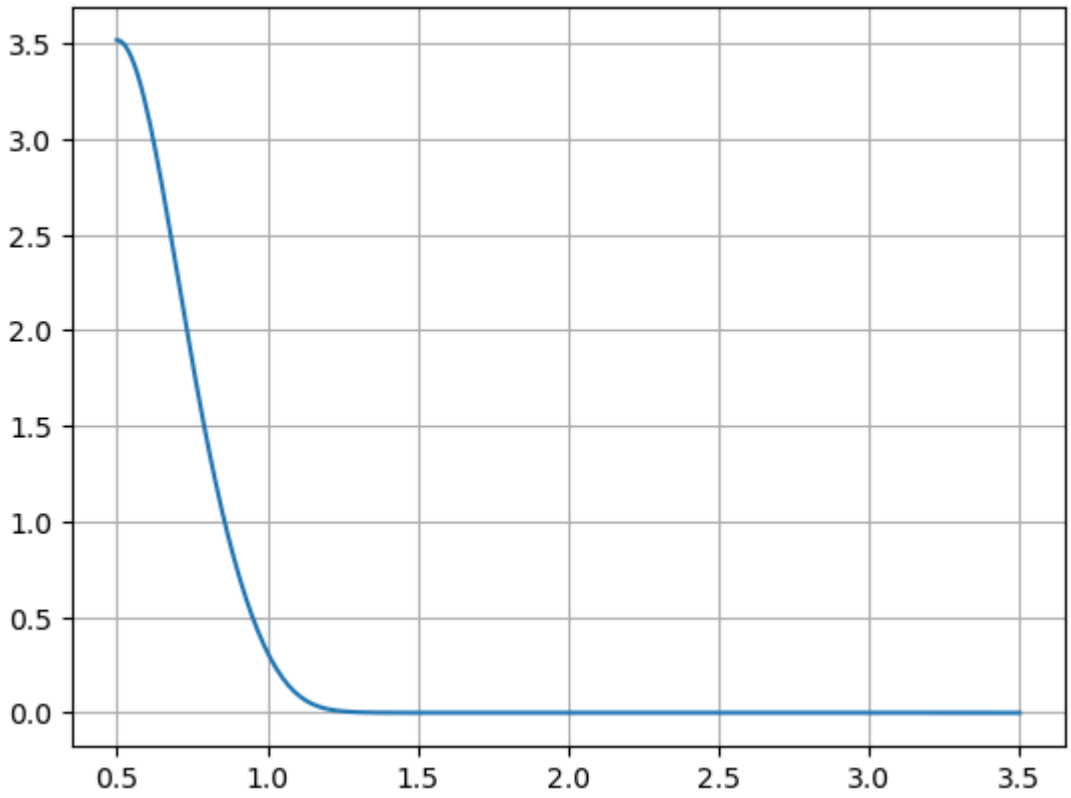
Numerical solve ODE with injecting boundary condition to find steady state solution

In [4]:
```
def kappa(x):
    return D(x)*np.exp(-beta_U(x))

x_minus_half = x_arr - h/2.0
x_plus_half = x_arr + h/2.0
u_arr = np.zeros(x_arr.size)

A = np.zeros((N-1, N-1))
np.fill_diagonal(A, -(kappa(x_minus_half)+kappa(x_plus_half)))
np.fill_diagonal(A[1:, ], kappa(x_minus_half[1:-1])) # subdiagonal
np.fill_diagonal(A[:, 1:], kappa(x_plus_half[:-2]))  # superdiagonal
A[0, 1] += kappa(x_minus_half[0])

f_vect = np.zeros(N-1)
f_vect[0] = 1
```

```
u_arr[0:-1] = spsolve(A, f_vect)
Pst_arr = u_arr*np.exp(-beta_U(x_arr))
Pst_arr /= (np.sum(Pst_arr)*h)

plt.plot(x_arr, Pst_arr)
plt.grid()

inject_rate = D(x_arr[-1])*np.exp(-beta_U(x_arr[-1])+beta_U(x_arr[-2]))*Pst_
1/inject_rate
```

```
/tmp/ipykernel_789248/1491311078.py:17: SparseEfficiencyWarning: spsolve req
uires A be CSC or CSR matrix format
  u_arr[0:-1] = spsolve(A, f_vect)
```

Out[4]: 112932.29344885287



In [ ]:

Numerically Nest integrate for Steady-State Flux and Probability distribution function

In [5]:
```
# Define the inner function to integrate as a function of y
def inner_integrand(y):
    return 1.0/D(y)*np.exp(beta_U(y))

# Define the inner integral as a function of x
def inner_integral(x):
    y_lower = x
    y_upper = b
    result, error = quad(inner_integrand, y_lower, y_upper)
```

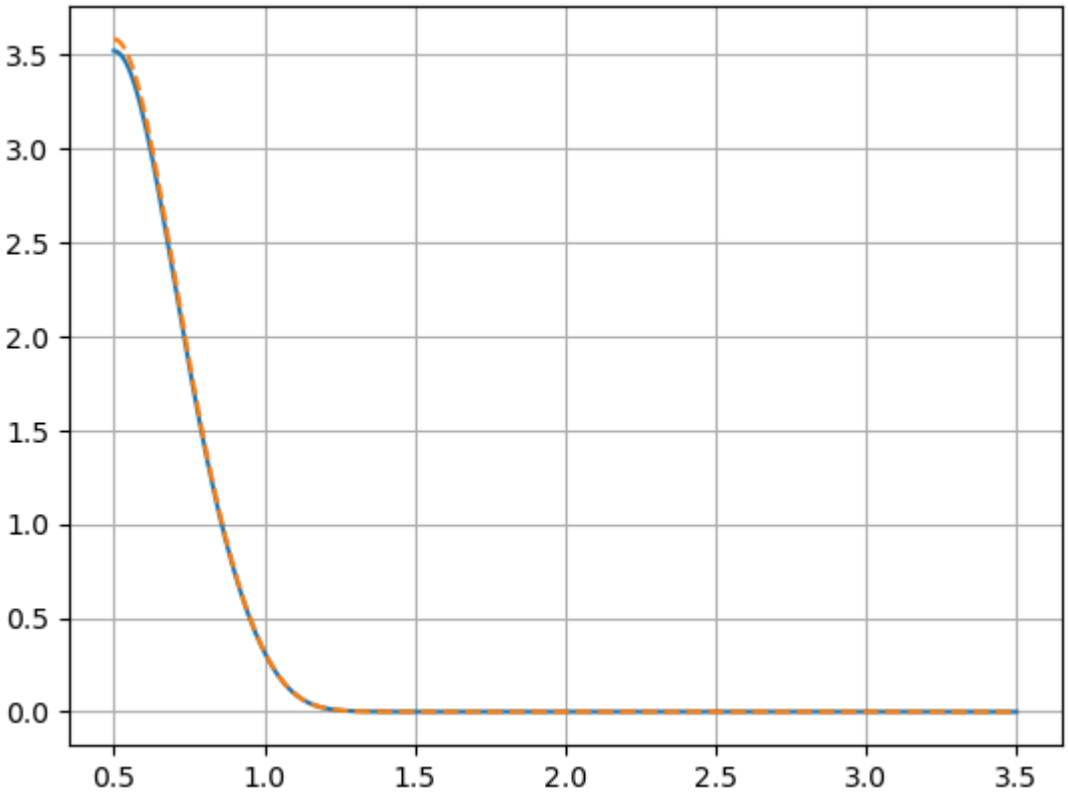
```
    return result

# Define the outer integral
x_lower = a
x_upper = b

# Define the outer function to integrate (also as a function of x)
def outer_integrand(x):
    return np.exp(-beta_U(x))*inner_integral(x)

# Perform the outer integration
invert_st_flux, error = quad(outer_integrand, x_lower, x_upper)
st_flux = 1.0/invert_st_flux
print(invert_st_flux)
```

```
55478.78943520375
```

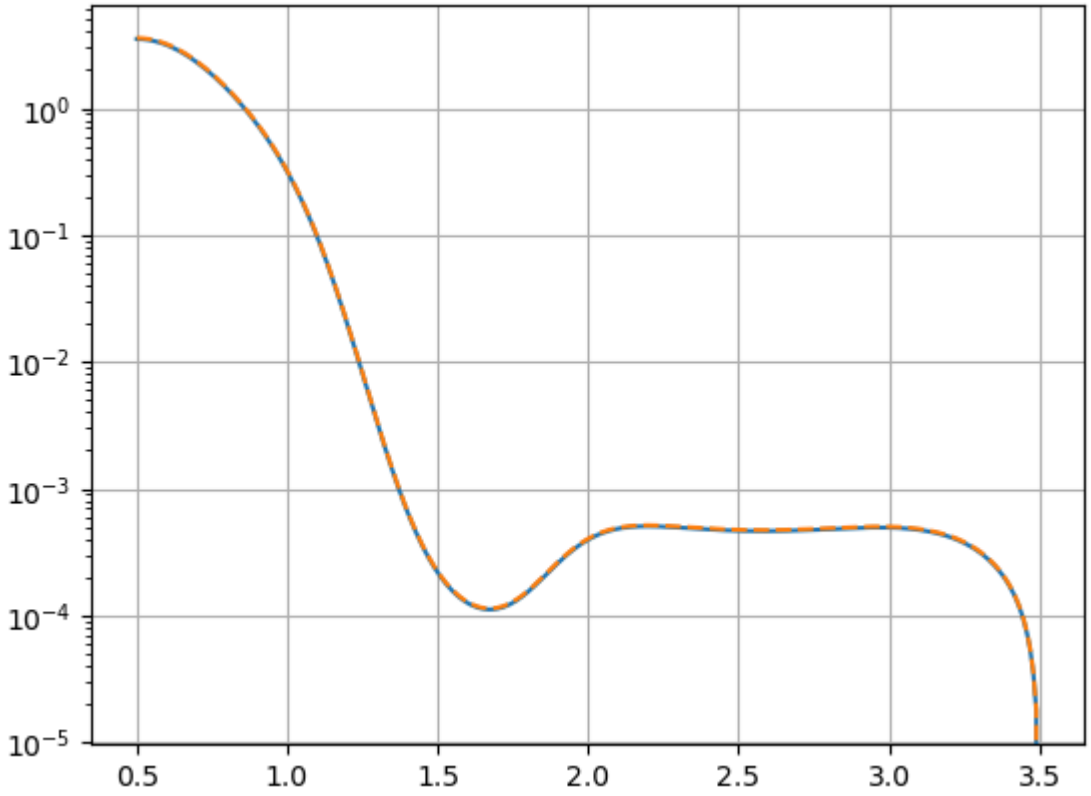
In [6]:

```
def st_P_func(x):
    def integrand(y):
        return 1.0/D(y)*np.exp(beta_U(y))
    # Perform the integration
    y_lower = x
    y_upper = b
    result, error = quad(integrand, y_lower, y_upper)
    result *= st_flux*np.exp(-beta_U(x))
    return result

st_P_arr = np.zeros(x_arr.size)
for i in np.arange(x_arr.size):
    st_P_arr[i] = st_P_func(x_arr[i])

plt.plot(x_arr, Pst_arr)
plt.plot(x_arr, st_P_arr, '--')

plt.grid()

print(Pst_arr[:5], st_P_arr[:5])
```

```
[3.5182883  3.51435165 3.50260409 3.48313637 3.45610469] [3.58128827 3.57728
112 3.56532321 3.54550689 3.51799117]
```



In [7]:
```
plt.semilogy(x_arr, Pst_arr)
plt.semilogy(x_arr, st_P_arr, '--')

plt.grid()
```



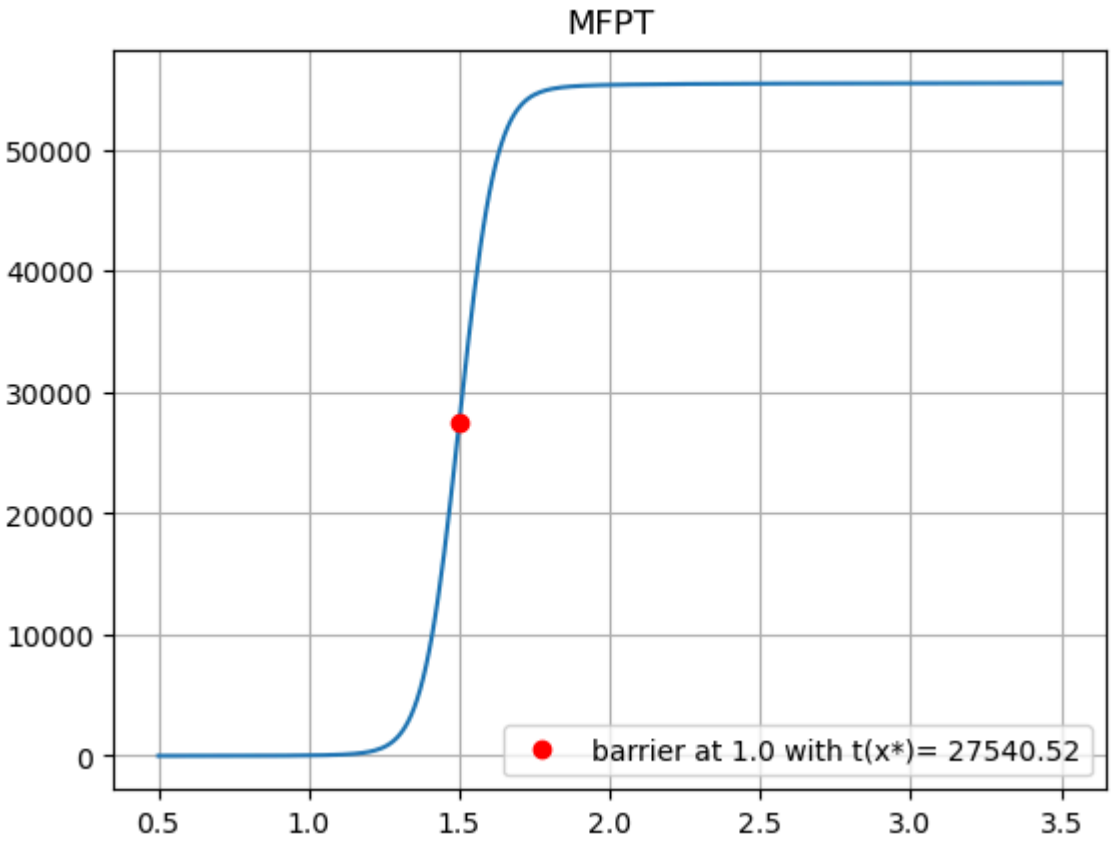
In [ ]:

Numerically Nest integrate for MFPT

In [8]:
```
x0 = a   # Regura's method, overlap the starting point and reflecting bounda

# Define the inner function to integrate as a function of z
def inner_integrand(z):
    return np.exp(-beta_U(z))

# Define the inner integral as a function of y
def inner_integral(y):
    z_lower = a
    z_upper = y
    result, error = quad(inner_integrand, z_lower, z_upper)
    return result

# Define the outer integral
y_lower = x0
y_upper = b

# Define the outer function to integrate (also as a function of y)
def outer_integrand(y):
    return np.exp(beta_U(y))*inner_integral(y)/D(y)

# Perform the outer integration (from x0 to b)
result, error = quad(outer_integrand, y_lower, y_upper)

mfpt_arr = np.zeros(x_arr.size)
for i in np.arange(x_arr.size):
    mfpt_arr[i], _ = quad(outer_integrand, y_lower, x_arr[i])
```

In [9]:
```
idx_barrier = np.where((x_arr > 1.5-h/2) & (x_arr < 1.5 + h/2))
t_star = mfpt_arr[idx_barrier][0]
print(f"{2*t_star} = 2*t(x*)")
print(f"{mfpt_arr[-1]} = t(b)")
print(f"{invert_st_flux} = 1/J")
print(f"{1/inject_rate/2} = 1/inject_rate")

plt.plot(x_arr, mfpt_arr)
plt.plot(1.5, t_star , 'ro', label=f"barrier at 1.0 with t(x*)={t_star: .2f}
plt.title('MFPT')
plt.legend()
plt.grid()
```

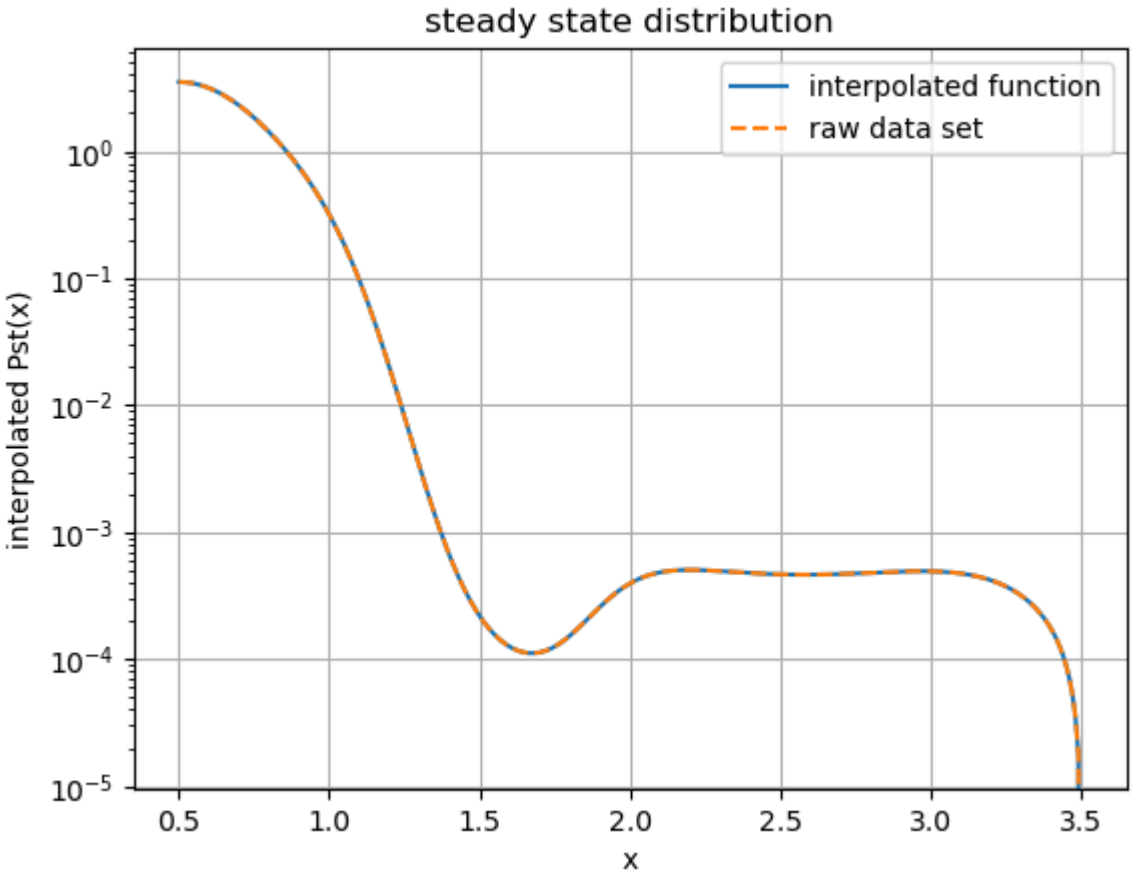
```
55081.04319401654 = 2*t(x*)
55478.78943520375 = t(b)
55478.78943520375 = 1/J
56466.146724426435 = 1/inject_rate
```



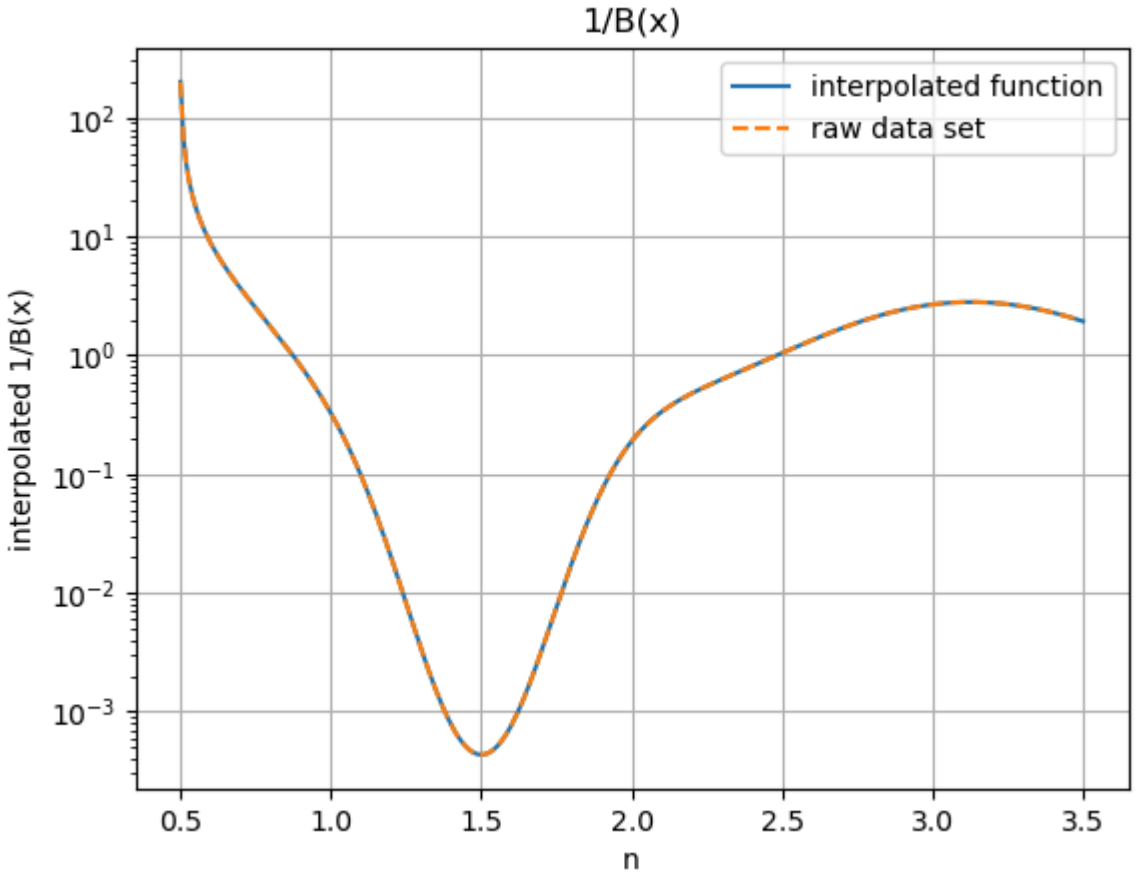
In [ ]:

Reconstruct Free Energy from Numerical Solution of Pst_arr and Numerical Integration of mfpt

In [10]:
```
interp_Pst_func = interp1d(x_arr, Pst_arr, kind='cubic', fill_value="extrapc
# interp_Pst_func = PchipInterpolator(b_arr, Pst_arr)

plt.semilogy(x_arr, interp_Pst_func(x_arr), label="interpolated function")
plt.semilogy(x_arr, Pst_arr, '--', label="raw data set")
# Plot formatting
plt.xlabel('x')
plt.ylabel('interpolated Pst(x)')
plt.title('steady state distribution')
plt.legend()
plt.grid()
```



In [11]:
```
# Except for the absorbing boundary Pst(b) = 0, and also avoid D(x)=0
integral_Pst_arr = np.zeros(N-1)
Bx_arr = np.zeros(N-1)

for i in range(N-1):
    integral_Pst_arr[i], _ = quad(interp_Pst_func, x_arr[i], x_arr[-1])
    Bx_arr[i] = -1.0/Pst_arr[i]*(integral_Pst_arr[i]-(mfpt_arr[-1]-mfpt_arr[

interp_invertBx_func = interp1d(x_arr[:-1], 1.0/Bx_arr, kind='cubic', fill_v

plt.semilogy(x_arr, interp_invertBx_func(x_arr), label="interpolated functic
plt.semilogy(x_arr[:-1], 1.0/Bx_arr, '--', label="raw data set")
# Plot formatting
plt.xlabel('n')
plt.ylabel('interpolated 1/B(x)')
plt.title('1/B(x)')
plt.legend()
plt.grid()
```

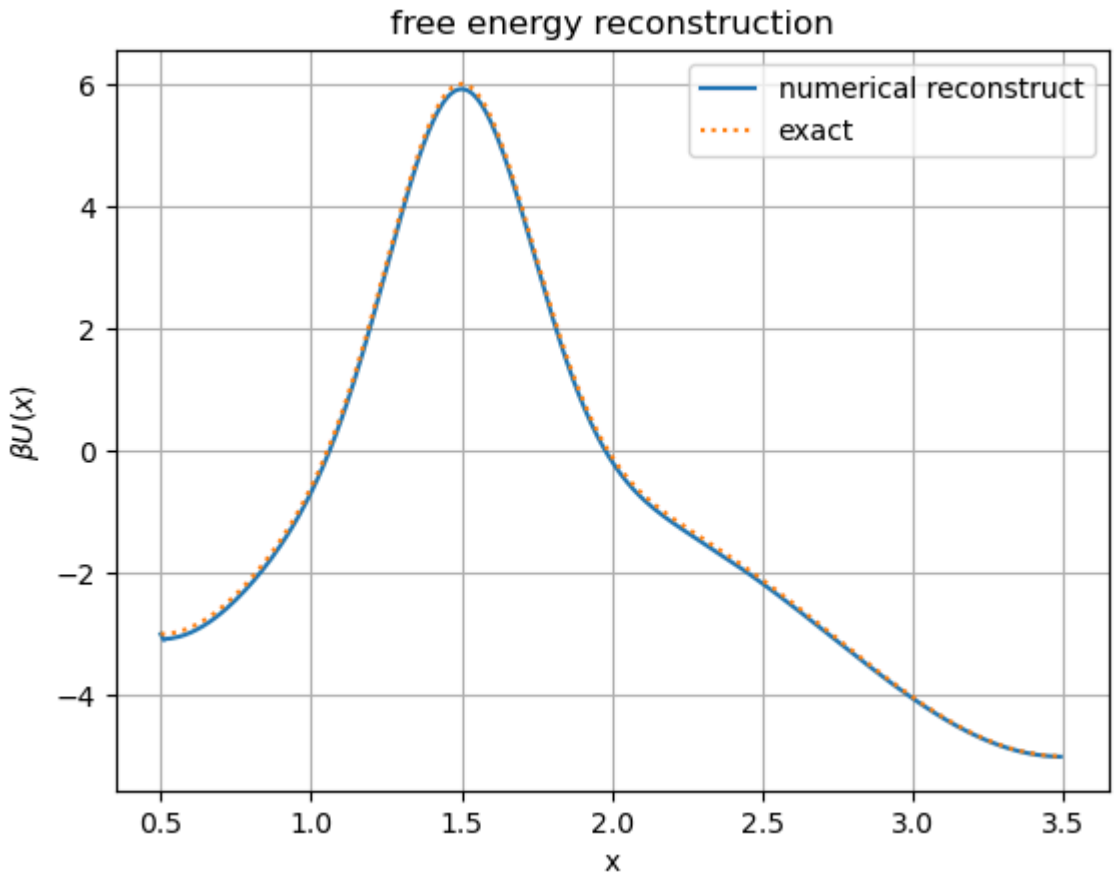


In [12]:
```
Bx_arr[:10]
```

Out[12]:
```
array([0.005     , 0.01501303, 0.02508193, 0.03525176, 0.0455682 ,
       0.05607776, 0.06682802, 0.07786791, 0.08924796, 0.10102059])
```

In [13]:
```
integral_invertBx_arr = np.zeros(N-1)
beta_Urec1_arr = np.zeros(N-1)

for i in range(x_arr.size-1):
    # Here x0 is b_arr[0]
    integral_invertBx_arr[i], _ = quad(interp_invertBx_func, x_arr[0], x_arr
    beta_Urec1_arr[i] = beta_U(x_arr[0])+np.log(Bx_arr[i]/Bx_arr[0])-integra

print(_)

plt.plot(x_arr[:-1], beta_Urec1_arr, label="numerical reconstruct")
plt.plot(x_arr, beta_U(x_arr), ":", label="exact")

# Plot formatting
plt.xlabel('x')
plt.ylabel('$ \\beta U(x) $')
plt.title('free energy reconstruction')
plt.legend()
plt.grid()
```

```
7.233454141622772e-08
```



In [14]:
```
print(beta_Urec1_arr[0], beta_U(x_arr[0]))
beta_Urec1_arr[:20]
```

```
-3.0000000000018456 -3.0000000000018456
```

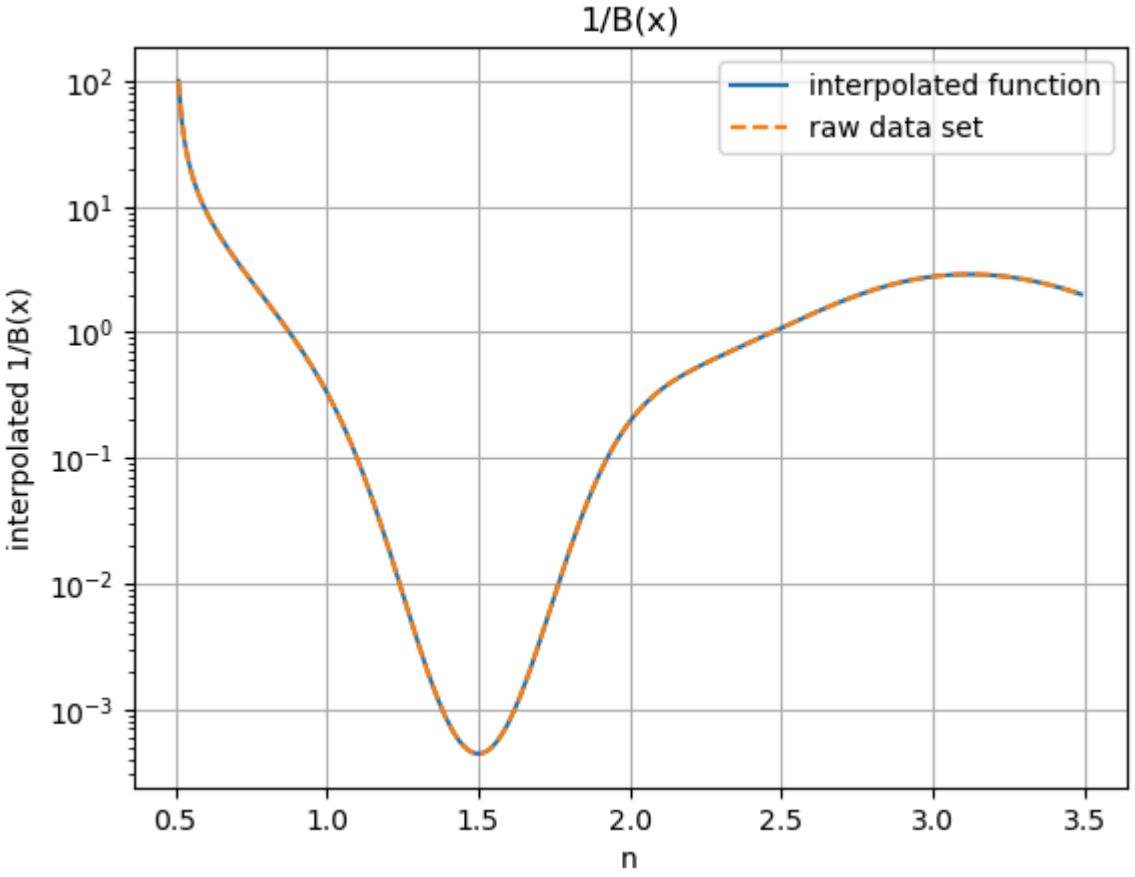
Out[14]:
```
array([-3.        , -3.0977337 , -3.07497171, -3.07458831, -3.06538317,
       -3.05575949, -3.04347431, -3.02914846, -3.01264249, -2.99402583,
       -2.97331624, -2.95054899, -2.92575786, -2.89897971, -2.87025299,
       -2.83961716, -2.80711271, -2.77278039, -2.73666085, -2.69879408])
```

In [ ]:
```
```

Reconstruct Free Energy from Numerical Integration of mfpt and Pst (st_P_func)

In [15]:
```
# Except for the absorbing boundary Pst(b) = 0, D(x)=0, also avoid at reflec
Bx_arr = np.zeros(N-2)
integral_Pst_arr = np.zeros(N-2)
for i in range(N-2):
    integral_Pst_arr[i], _ = quad(st_P_func, x_arr[1+i], x_arr[-1])
    Bx_arr[i] = -1.0/st_P_func(x_arr[1+i])*(integral_Pst_arr[i]-st_flux*(mfp

interp_invertBx_func = interp1d(x_arr[1:-1], 1.0/Bx_arr, kind='cubic') # fil

plt.semilogy(x_arr[1:-1], interp_invertBx_func(x_arr[1:-1]), label="interpol
plt.semilogy(x_arr[1:-1], 1.0/Bx_arr, '--', label="raw data set")
# Plot formatting
plt.xlabel('n')
```

```
plt.ylabel('interpolated 1/B(x)')
plt.title('1/B(x)')
plt.legend()
plt.grid()
```



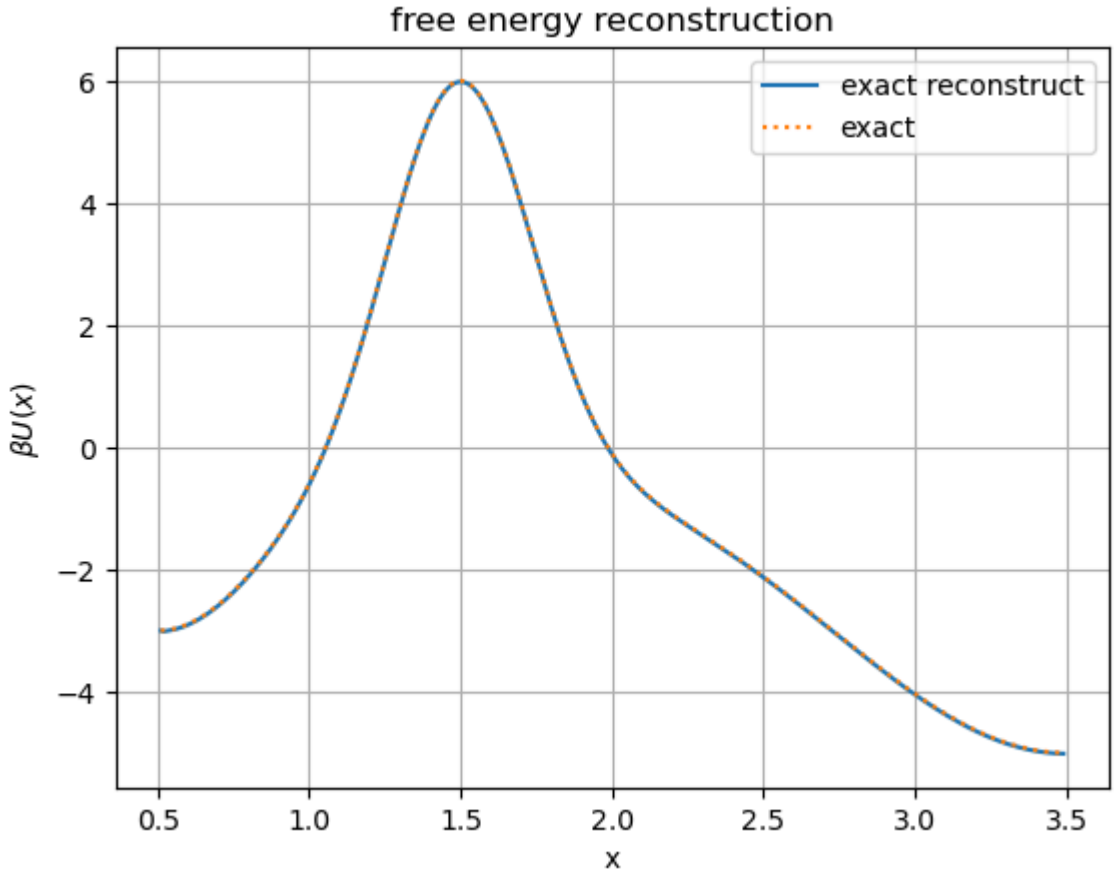
In [16]:
```
Bx_arr[:10]
```

Out[16]:
```
array([0.01000743, 0.02005954, 0.03020131, 0.04047825, 0.05093668,
       0.06162391, 0.07258855, 0.08388076, 0.09555251, 0.10765791])
```

In [17]:
```
integral_invertBx_arr = np.zeros(N-2)
beta_Urec2_arr = np.zeros(N-2)

for i in range(N-2):
    # Here x0 is x_arr[1]
    integral_invertBx_arr[i], _ = quad(interp_invertBx_func, x_arr[1], x_arr
    beta_Urec2_arr[i] = beta_U(x_arr[1])+np.log(Bx_arr[i]/Bx_arr[0])-integra

print(_)

plt.plot(x_arr[1:-1], beta_Urec2_arr, label="exact reconstruct")
plt.plot(x_arr[1:-1], beta_U(x_arr[1:-1]), ':', label="exact")

# Plot formatting
plt.xlabel('x')
plt.ylabel('$ \\beta U(x) $')
plt.title('free energy reconstruction')
```

```
plt.legend()
plt.grid()
```

```
3.26602241456593e-08
```



In [18]:
```
print(f"reconstructed: \n {beta_Urec2_arr[:20]}, \n exat data: \n {beta_U(x_
```

```
reconstructed:
 [-2.9988855  -3.01206044 -3.00247252 -2.99572302 -2.9854352  -2.97332585
 -2.95895219 -2.94245886 -2.92383886 -2.90313029 -2.88036291 -2.85557196
 -2.82879398 -2.80006731 -2.76943161 -2.73692725 -2.70259503 -2.66647558
 -2.6286089  -2.58903382],
 exat data:
 [-3.         -2.9988855  -2.9955422  -2.9899737  -2.9821878  -2.97219644
 -2.96001549 -2.9456646  -2.92916698 -2.9105491  -2.88984052 -2.86707349
 -2.84228268 -2.81550479 -2.78677821 -2.75614254 -2.72363822 -2.68930603
 -2.6531866  -2.61531993]
```

In [ ]:
```
```

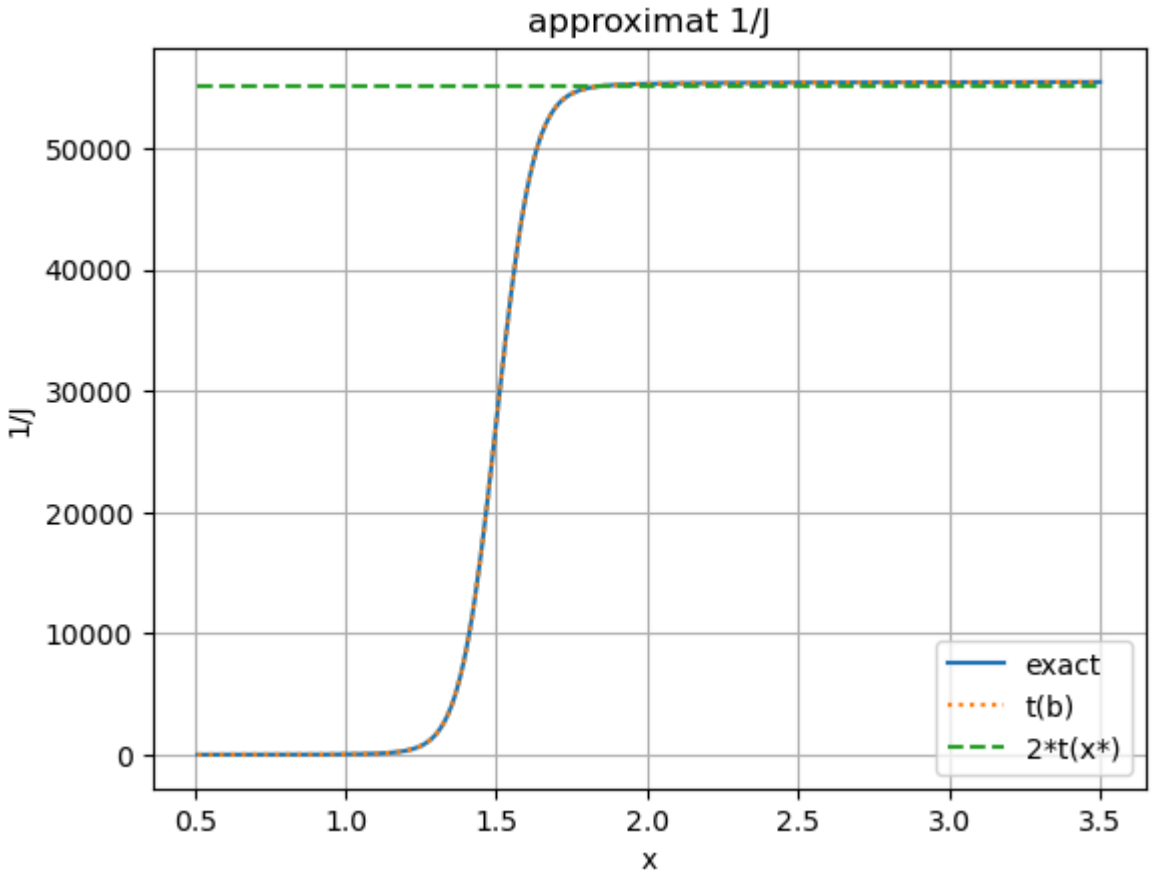
Verification Hill Relation: plot the $1/flux$, $\tau(b)$, $and\ 2\tau(x*)$ v.s different location of absorbing boundary

In [19]:
```
def exact_flux(a, b):
    # Define the inner function to integrate as a function of y
    def inner_integrand(y):
        return 1.0/D(y)*np.exp(beta_U(y))
```

```
    # Define the inner integral as a function of x
    def inner_integral(x):
        y_lower = x
        y_upper = b
        result, error = quad(inner_integrand, y_lower, y_upper)
        return result

    # Define the outer integral
    x_lower = a
    x_upper = b

    # Define the outer function to integrate (also as a function of x)
    def outer_integrand(x):
        return np.exp(-beta_U(x))*inner_integral(x)

    # Perform the outer integration
    invert_Jst, error = quad(outer_integrand, x_lower, x_upper)
    return 1/invert_Jst
```

In [20]:
```
# for valid value, exclude the refleting boundary
Jst_arr = np.zeros(N-1)

for i in np.arange(N-1):
    Jst_arr[i] = exact_flux(x_arr[0], x_arr[1+i])
```

In [21]:
```
plt.plot(x_arr[1:], 1/Jst_arr, label="exact")
plt.plot(x_arr[1:], mfpt_arr[1:], ':', label="t(b)")
plt.plot(x_arr[1:], 2*t_star*np.ones(x_arr[1:].size), '--', label="2*t(x*)")

# Plot formatting
plt.xlabel('x')
plt.ylabel('1/J')
plt.title('approximat 1/J')
plt.legend()
plt.grid()
```



In [22]:
```
print(f"1/J(b;x0):\n {1/Jst_arr[:20]}, \n t(b;x0): \n {mfpt_arr[1:21]}")
```

```
1/J(b;x0):
 [0.00500186 0.02002976 0.04515082 0.08047742 0.12616781 0.18242703
 0.24950807 0.3277133  0.41739609 0.51896285 0.63287527 0.75965299
 0.89987648 1.0541905  1.22330782 1.40801353 1.6091698  1.82772134
 2.06470142 2.32123875],
 t(b;x0):
 [0.00500186 0.02002976 0.04515082 0.08047742 0.12616781 0.18242703
 0.24950807 0.3277133  0.41739609 0.51896285 0.63287527 0.75965299
 0.89987648 1.0541905  1.22330782 1.40801353 1.6091698  1.82772134
 2.06470142 2.32123875]
```

In [ ]:
```
```

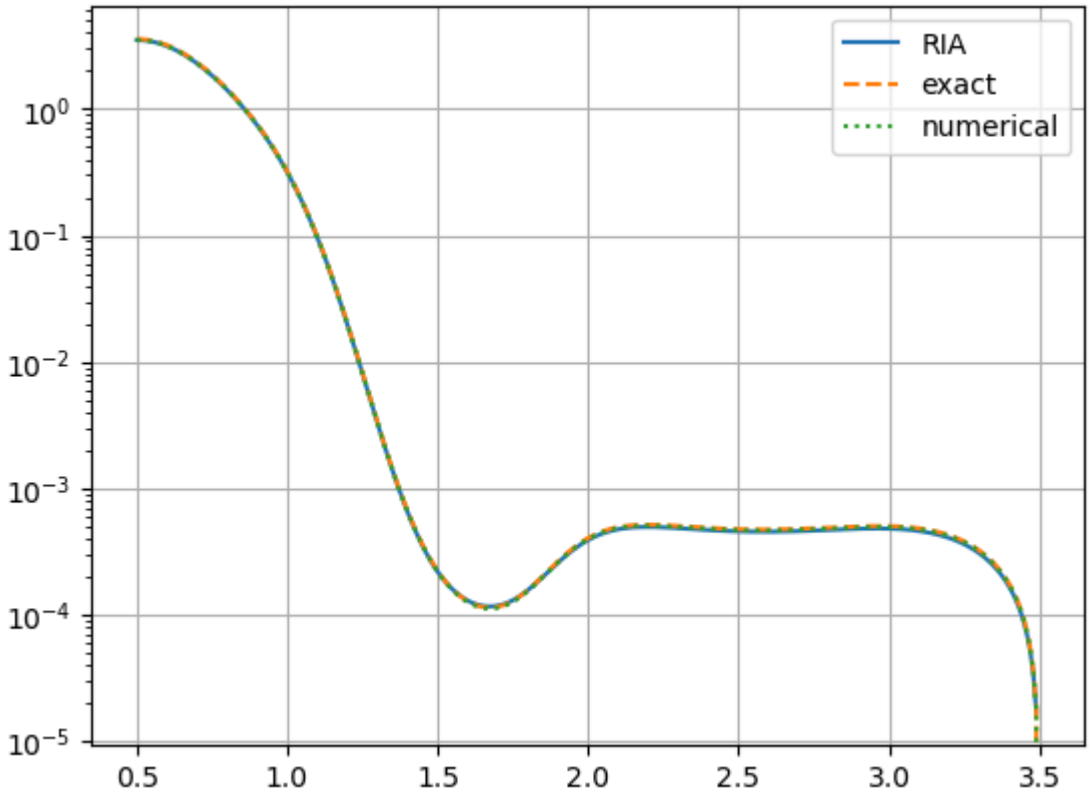
Transfer Matrix with Recycling Boundary Condition

In [23]:
```
from transfer_matrix_reptile import TransferMatrix_ReInAb

ria_trans = TransferMatrix_ReInAb(h, x_arr, beta_U, 0)
```

In [24]:
```
# plt.plot(x_arr[:-1], ria_trans.steady_state, label="RIA")
# plt.plot(x_arr, st_P_arr, '--', label="exact")
# plt.plot(x_arr, Pst_arr, ':', label="numerical")
plt.semilogy(x_arr[:-1], ria_trans.steady_state, label="RIA")
plt.semilogy(x_arr, st_P_arr, '--', label="exact")
plt.semilogy(x_arr, Pst_arr, ':', label="numerical")
```

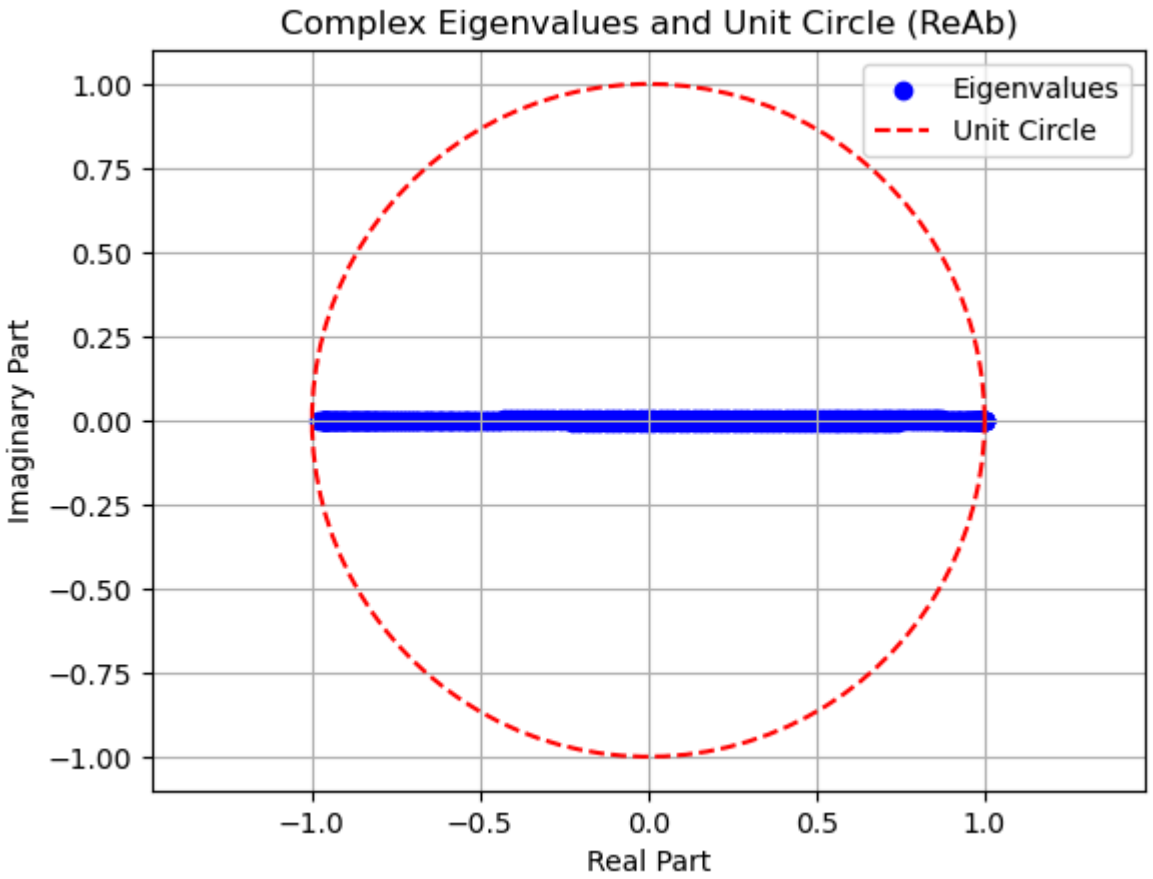
```
plt.legend()
plt.grid()
```

```
/home/yjiang23/anaconda3/lib/python3.11/site-packages/matplotlib/cbook.py:16
99: ComplexWarning: Casting complex values to real discards the imaginary pa
rt
  return math.isfinite(val)
/home/yjiang23/anaconda3/lib/python3.11/site-packages/matplotlib/cbook.py:13
45: ComplexWarning: Casting complex values to real discards the imaginary pa
rt
  return np.asarray(x, float)
```



In [25]:
```
ria_trans.plot_eigenvalues()
print(ria_trans.eig6_w)
```

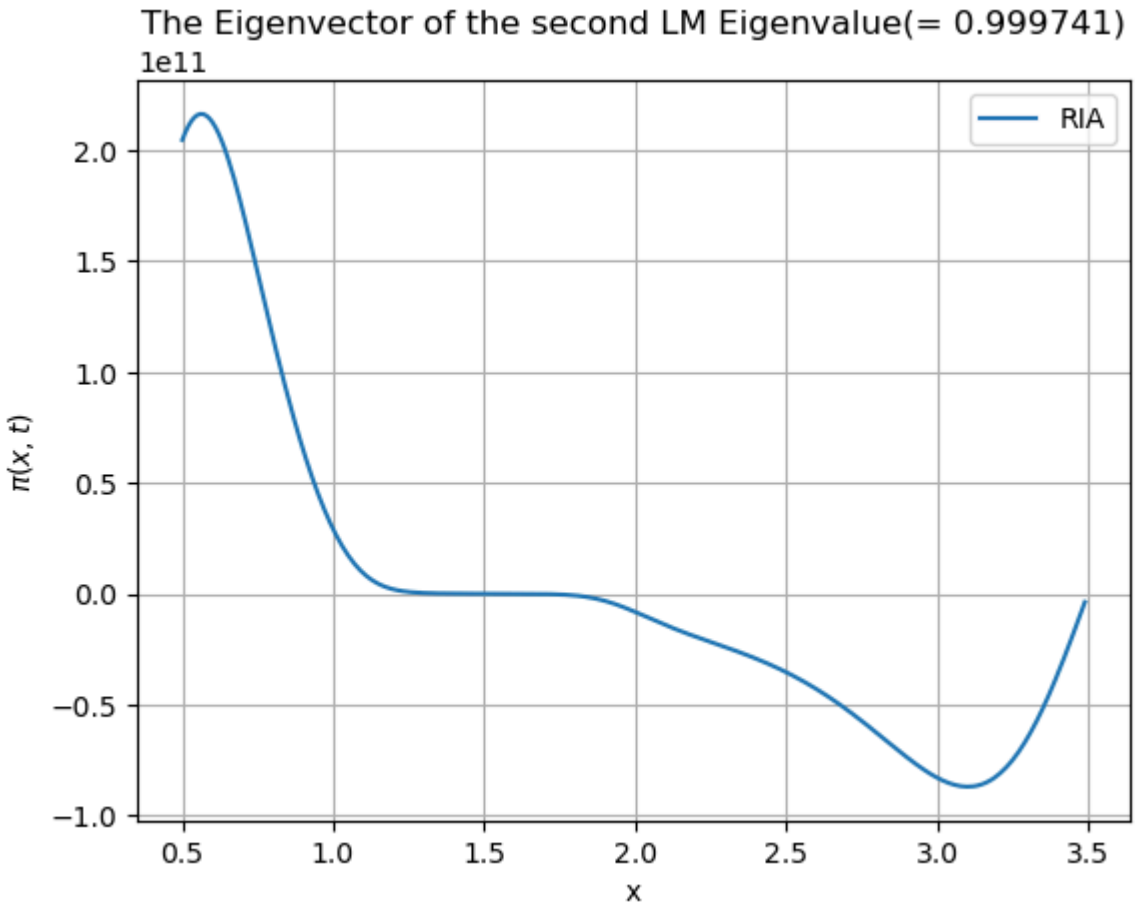
```
/home/yjiang23/Desktop/research/MFPT/MFPT_reconst_experiment/transfer_matrix
_reptile.py:547: RuntimeWarning: k >= N - 1 for N * N square matrix. Attempt
ing to use scipy.linalg.eig instead.
  eigenvalues, eigenvectors_mat = eigs(self.trans_mat, k=num_eigenvalues)
```




```
[1.         +0.j 0.99974063+0.j 0.99938798+0.j 0.99859263+0.j
 0.99807524+0.j 0.99749394+0.j]
```

In [26]:

```
plt.plot(x_arr[:-1], 1.0/(h*np.sum(ria_trans.eig6_v[:, 1]))*ria_trans.eig6_v
# Plot formatting
plt.xlabel('x')
plt.ylabel('$ \pi(x,t) $')
plt.title(f'The Eigenvector of the second LM Eigenvalue(= {ria_trans.eig6_w[
plt.legend()
plt.grid(True)
```



In [ ]:

Extract Steady State Distribution and MFPT from Simulation

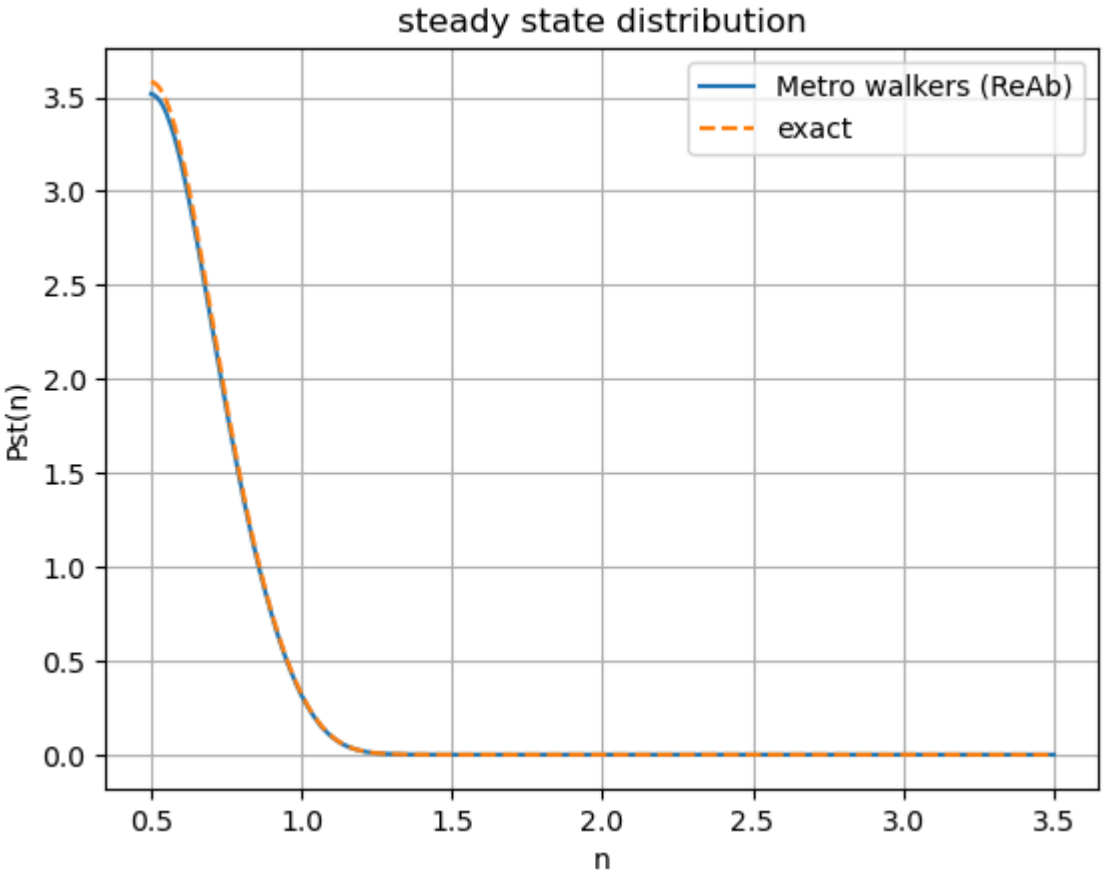
In [27]:
```
from mfpt_Pst_RW_simulate import simulate_ReAb, simulate_ReAb_accelerate

num_particles = 30
init_position_arr = a*np.ones(num_particles, dtype=float)
simu_x_arr = np.linspace(a, b, 301)
hx = (b-a)/(301-1)
ht = hx**2/(2*D0)
n_arr = np.arange(a, b+0.01, 0.01)
n_arr = np.round(n_arr, decimals=5)
hx == 0.01
```

Out[27]: True

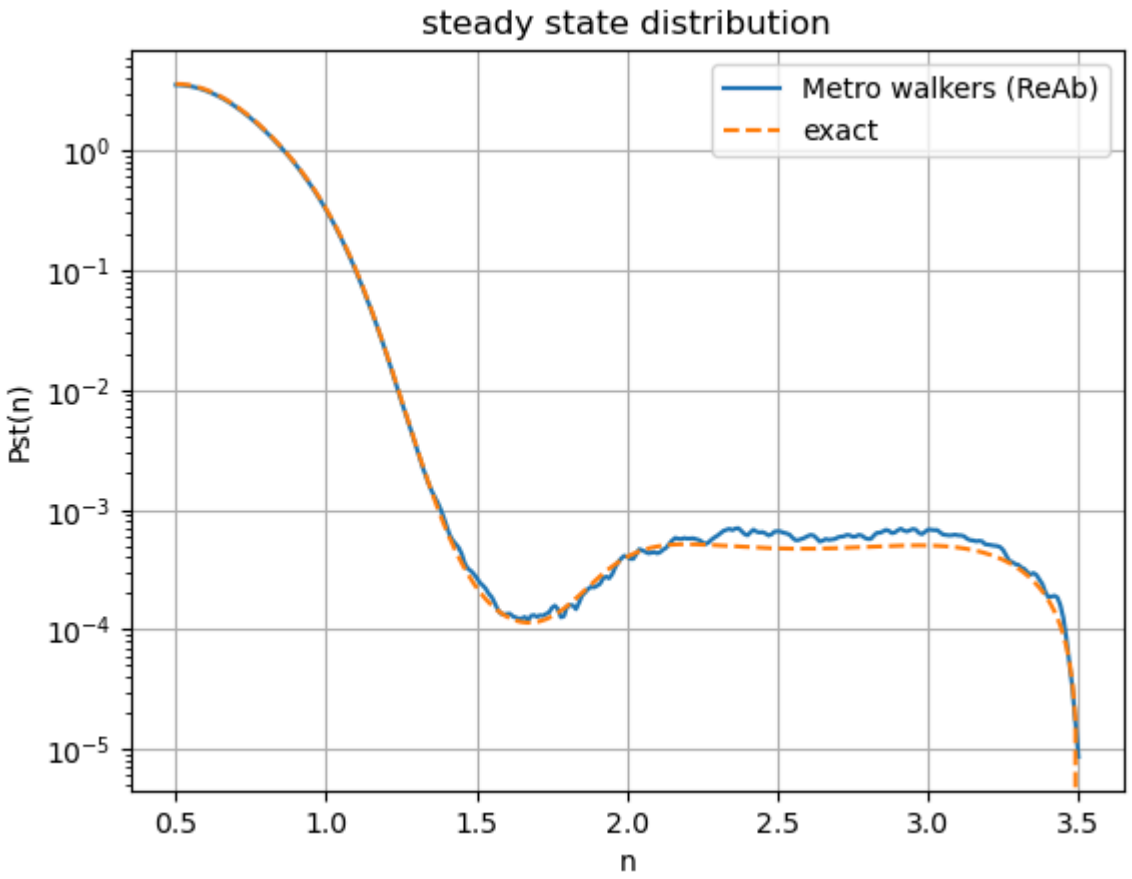
In [28]:
```
count_n, ti_n = simulate_ReAb_accelerate(init_position_arr=init_position_arr
```

In [29]:
```
Pst_n = count_n/(hx*np.sum(count_n))
plt.plot(n_arr, Pst_n, label="Metro walkers (ReAb)")
plt.plot(x_arr, st_P_arr, '--', label="exact")
# plt.plot(x_arr[:-1], ria_trans.steady_state, label="RIA")
# plt.plot(x_arr[:-1], 1.0/(h*np.sum(ria_trans.eig6_v[:, 1]))*ria_trans.eig6
```

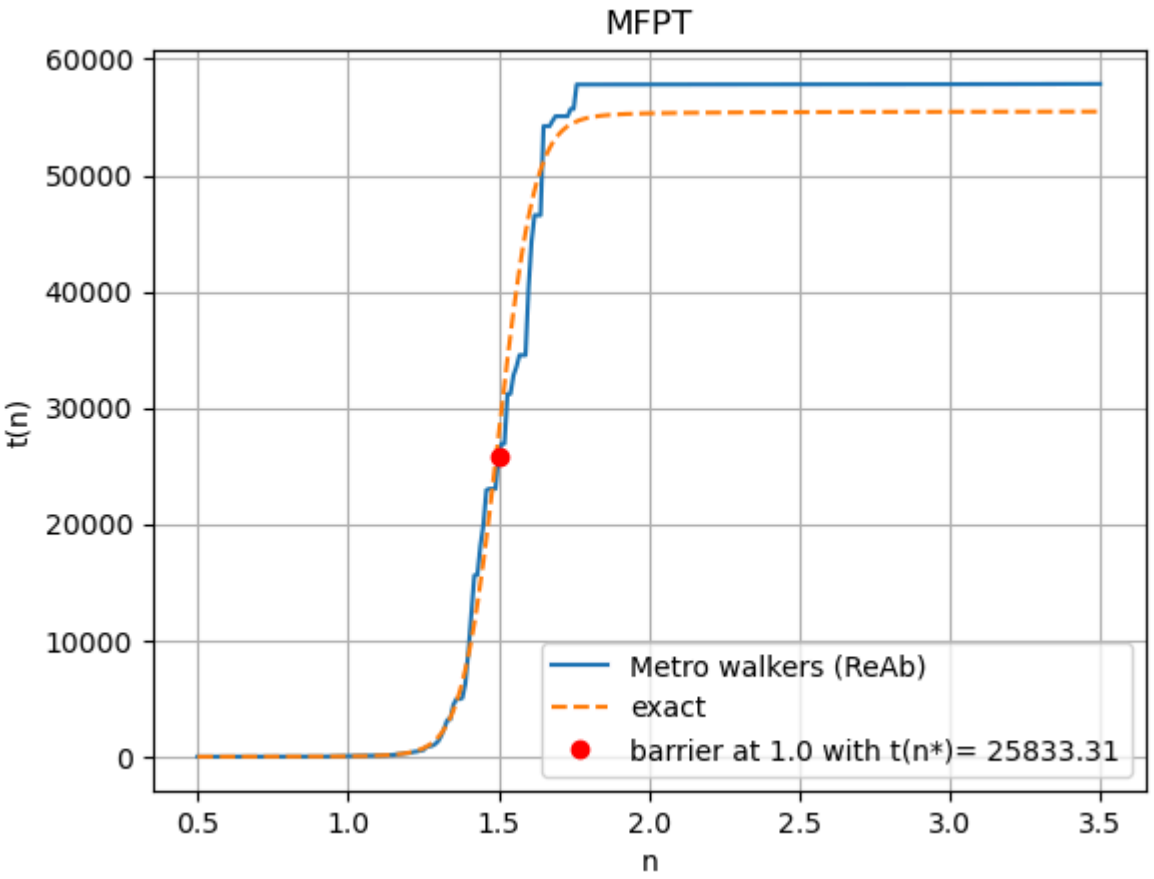
```
# Plot formatting
plt.xlabel('n')
plt.ylabel('Pst(n)')
plt.title('steady state distribution')
plt.legend()
plt.grid()
```



In [30]:
```
plt.semilogy(n_arr, Pst_n, label="Metro walkers (ReAb)")
# plt.semilogy(x_arr[:-1], ria_trans.steady_state, label="RIA")
# plt.semilogy(x_arr[:-1], 1.0/(h*np.sum(ria_trans.eig6_v[:, 1]))*ria_trans.
plt.semilogy(x_arr, st_P_arr, '--', label="exact")

# Plot formatting
plt.xlabel('n')
plt.ylabel('Pst(n)')
plt.title('steady state distribution')
plt.legend()
plt.grid()
```



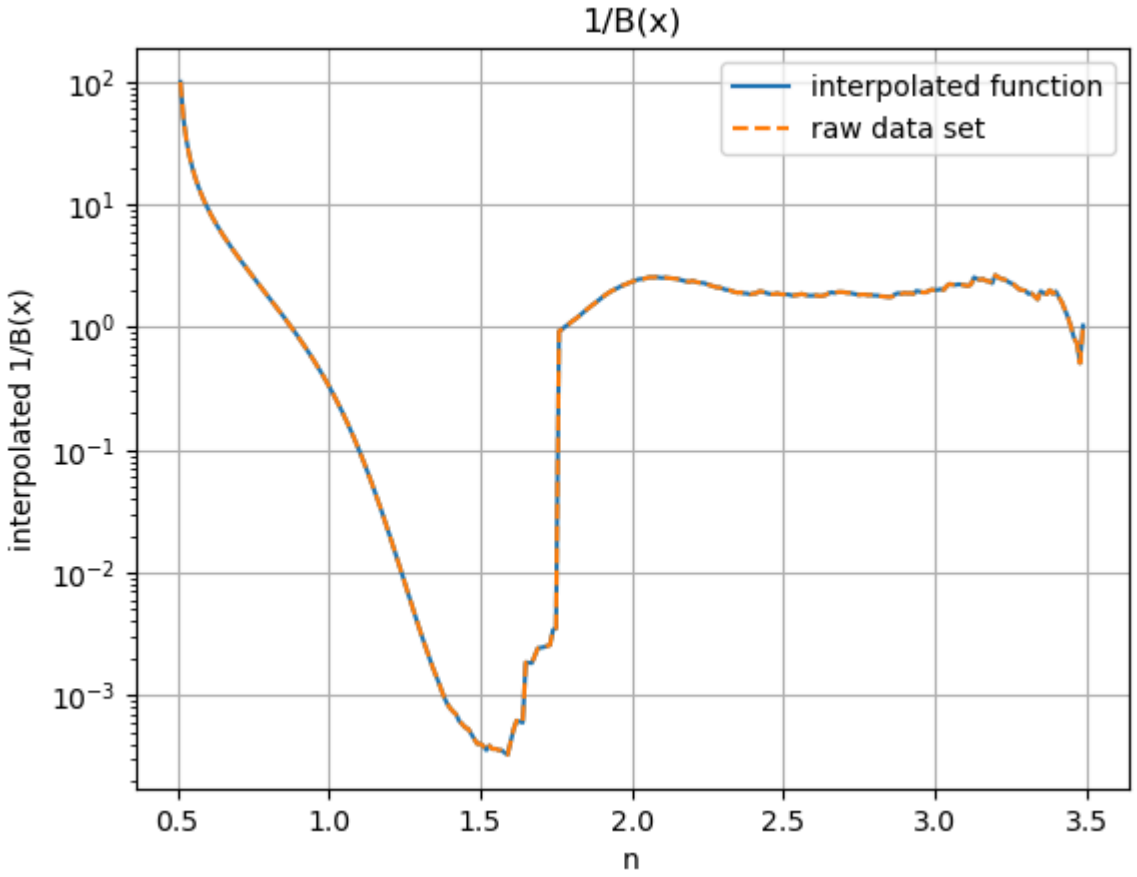
In [44]:
```
mfpt_simu_arr = np.mean(ti_n, axis=0)
idx_barrier = np.where(n_arr == 1.5)[0][0]
t_star = mfpt_simu_arr[idx_barrier]

plt.plot(n_arr, mfpt_simu_arr, label="Metro walkers (ReAb)")
plt.plot(x_arr, mfpt_arr, '--', label="exact")
plt.plot(1.5, t_star , 'ro', label=f'barrier at 1.0 with t(n*)={t_star: .2f}
# Plot formatting
plt.xlabel('n')
plt.ylabel('t(n)')
plt.title('MFPT')
plt.legend()
plt.grid()
```



Reconstruct Free Energy Using mfpt (mfpt_simu_arr) and Pst (Pst_n) extracted from Simulation

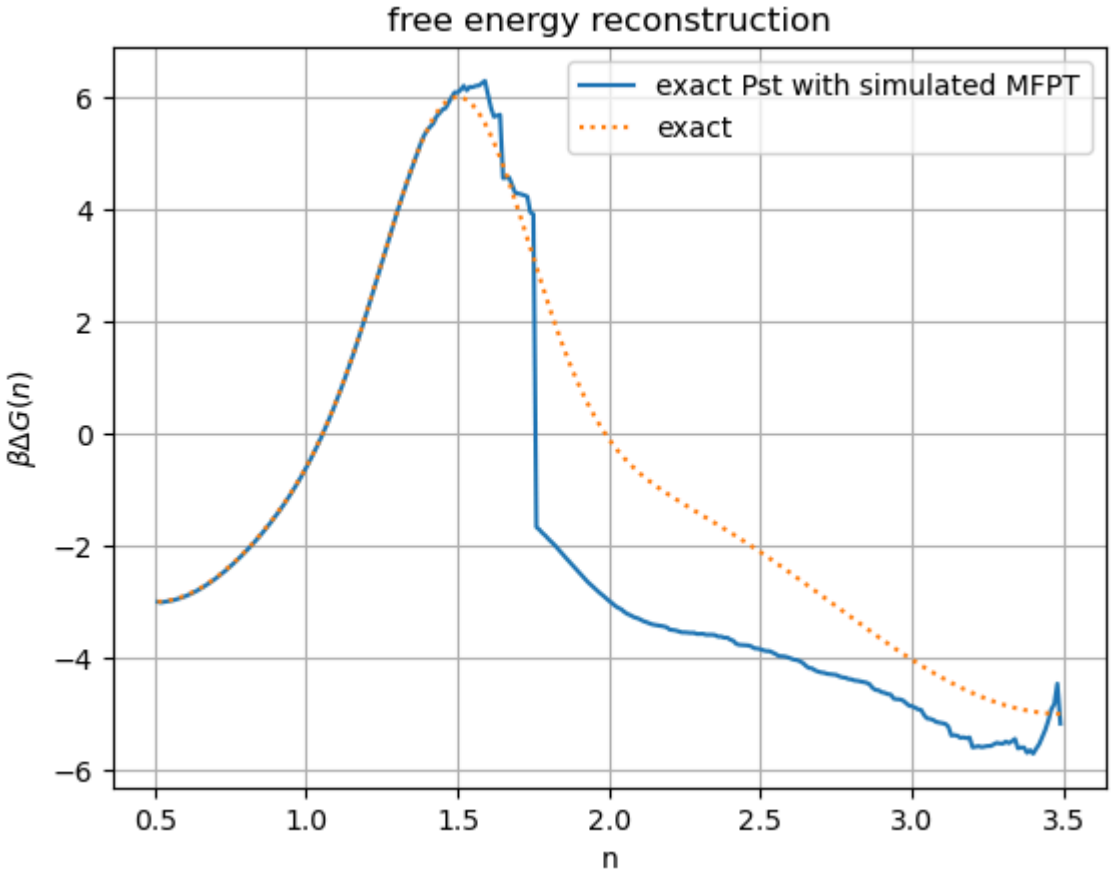
(1) Exact Steady State Distribution with simulated MFPT

In [32]:
```
# Except for the absorbing boundary Pst(b) = 0, D(x)=0, also avoid at reflec
Bx_arr = np.zeros(N-2)
integral_Pst_arr = np.zeros(N-2)
for i in range(N-2):
    integral_Pst_arr[i], _ = quad(st_P_func, x_arr[1+i], x_arr[-1])
    Bx_arr[i] = -1.0/st_P_func(x_arr[1+i])*(integral_Pst_arr[i]-(mfpt_simu_a

interp_invertBx_func = interp1d(x_arr[1:-1], 1.0/Bx_arr, kind='cubic') # fil

plt.semilogy(x_arr[1:-1], interp_invertBx_func(x_arr[1:-1]), label="interpol
plt.semilogy(x_arr[1:-1], 1.0/Bx_arr, '--', label="raw data set")
# Plot formatting
plt.xlabel('n')
plt.ylabel('interpolated 1/B(x)')
plt.title('1/B(x)')
plt.legend()
plt.grid()
```



In [33]:
```
integral_invertBx_arr = np.zeros(N-2)
beta_Grec2_arr = np.zeros(N-2)

for i in range(N-2):
    # Here x0 is x_arr[1]
    integral_invertBx_arr[i], _ = quad(interp_invertBx_func, x_arr[1], x_arr
    beta_Grec2_arr[i] = beta_U(x_arr[1])+np.log(Bx_arr[i]/Bx_arr[0])-integra

print(_)

plt.plot(x_arr[1:-1], beta_Grec2_arr, label="exact Pst with simulated MFPT")
plt.plot(x_arr[1:-1], beta_U(x_arr[1:-1]), ':', label="exact")

# Plot formatting
plt.xlabel('n')
plt.ylabel('$ \\beta \Delta G(n) $')
plt.title('free energy reconstruction')
plt.legend()
plt.grid()
```

```
/tmp/ipykernel_789248/3282593085.py:6: IntegrationWarning: The maximum numbe
r of subdivisions (50) has been achieved.
  If increasing the limit yields no improvement it is advised to analyze
  the integrand in order to determine the difficulties.  If the position of
a
  local difficulty can be determined (singularity, discontinuity) one will
  probably gain from splitting up the interval and calling the integrator
  on the subranges.  Perhaps a special-purpose integrator should be used.
  integral_invertBx_arr[i], _ = quad(interp_invertBx_func, x_arr[1], x_arr[1
+i])
0.004272760156140976
```

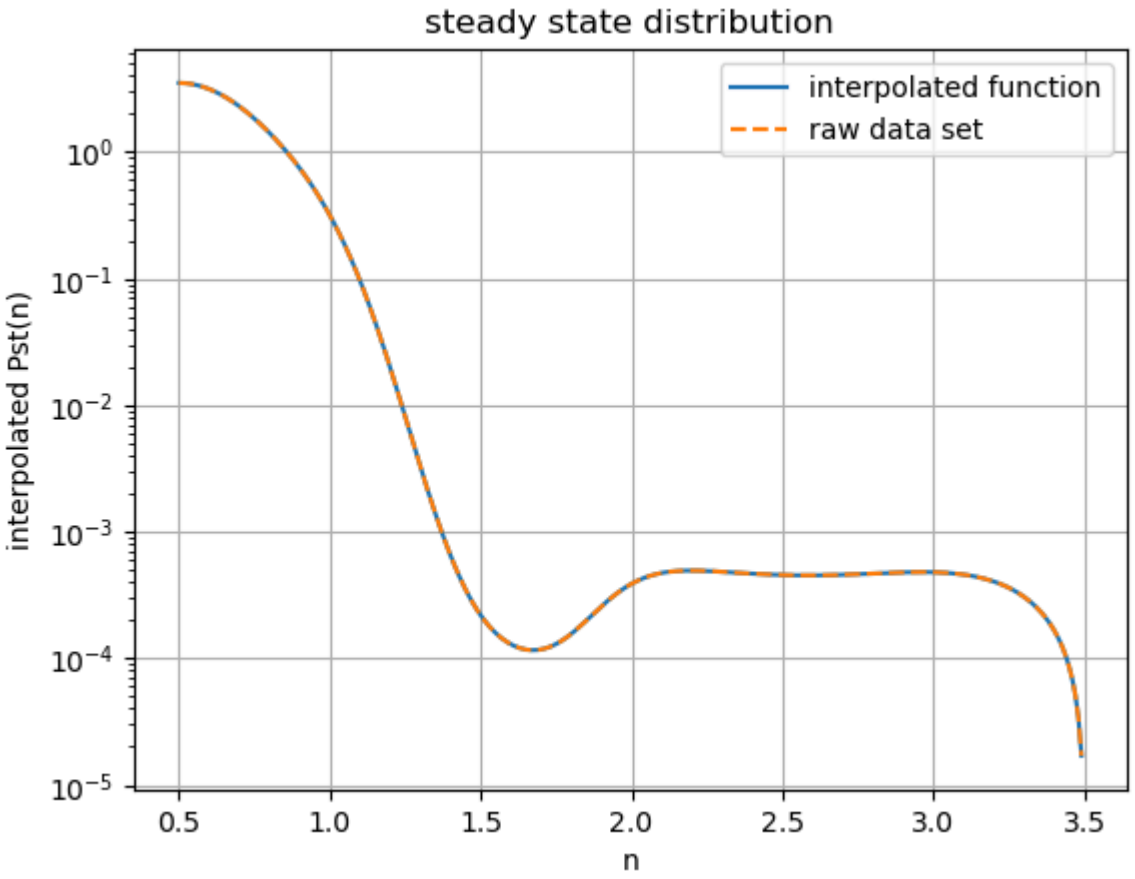


(2) Transfer Matrix Steady State Distribution with exact MFPT and then with simulated MFPT

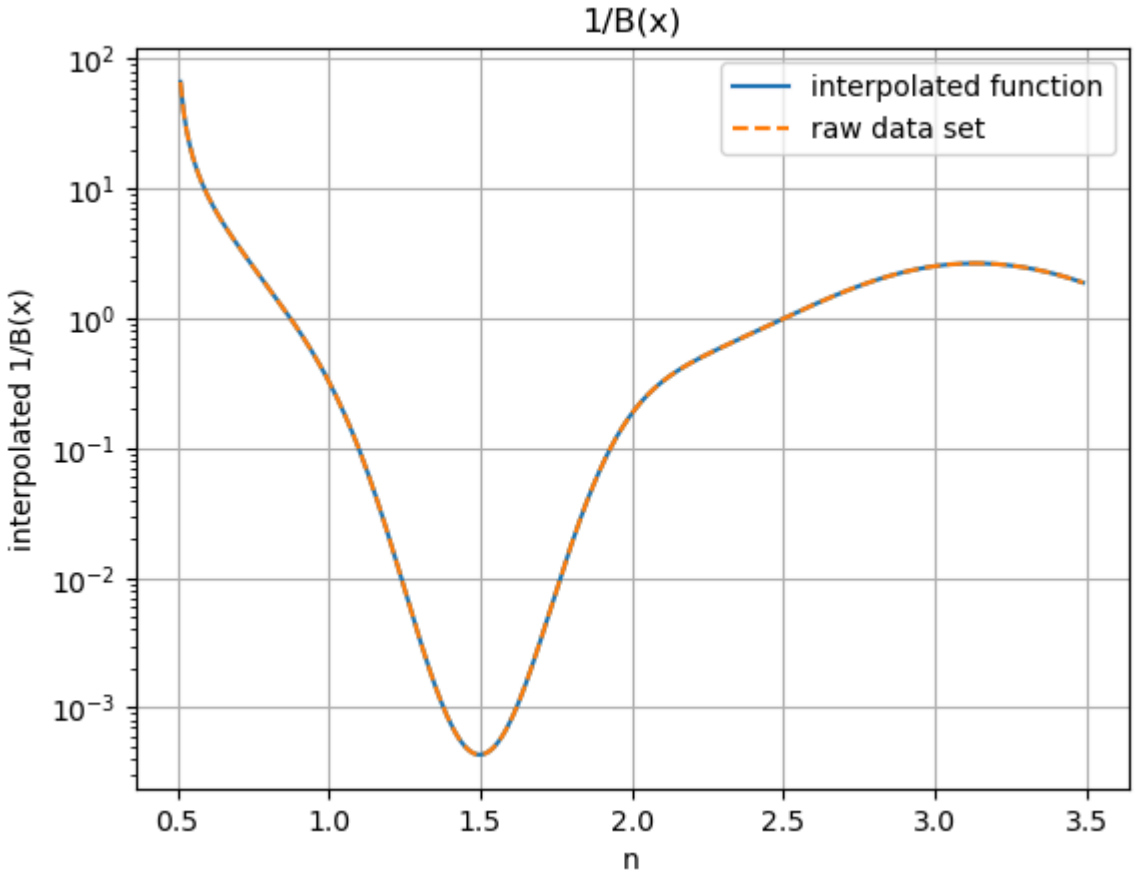
In [34]:
```
# interp_Pst_func = interp1d(x_arr[:-1], np.array(1.0/(h*np.sum(ria_trans.ei
interp_Pst_func = interp1d(x_arr[:-1], np.array(ria_trans.steady_state, dtyp
# interp_Pst_func = PchipInterpolator(b_arr, Pst_arr)

plt.semilogy(x_arr[:-1], interp_Pst_func(x_arr[:-1]), label="interpolated fu
# plt.semilogy(x_arr[:-1], np.array(1.0/(h*np.sum(ria_trans.eig6_v[:, 1]))*r
plt.semilogy(x_arr[:-1], ria_trans.steady_state, '--', label="raw data set")

# Plot formatting
plt.xlabel('n')
plt.ylabel('interpolated Pst(n)')
plt.title('steady state distribution')
```

```
plt.legend()
plt.grid()
```

```
/tmp/ipykernel_789248/3766657854.py:2: ComplexWarning: Casting complex value
s to real discards the imaginary part
  interp_Pst_func = interp1d(x_arr[:-1], np.array(ria_trans.steady_state, dt
ype=float), kind='cubic', fill_value="extrapolate")
```

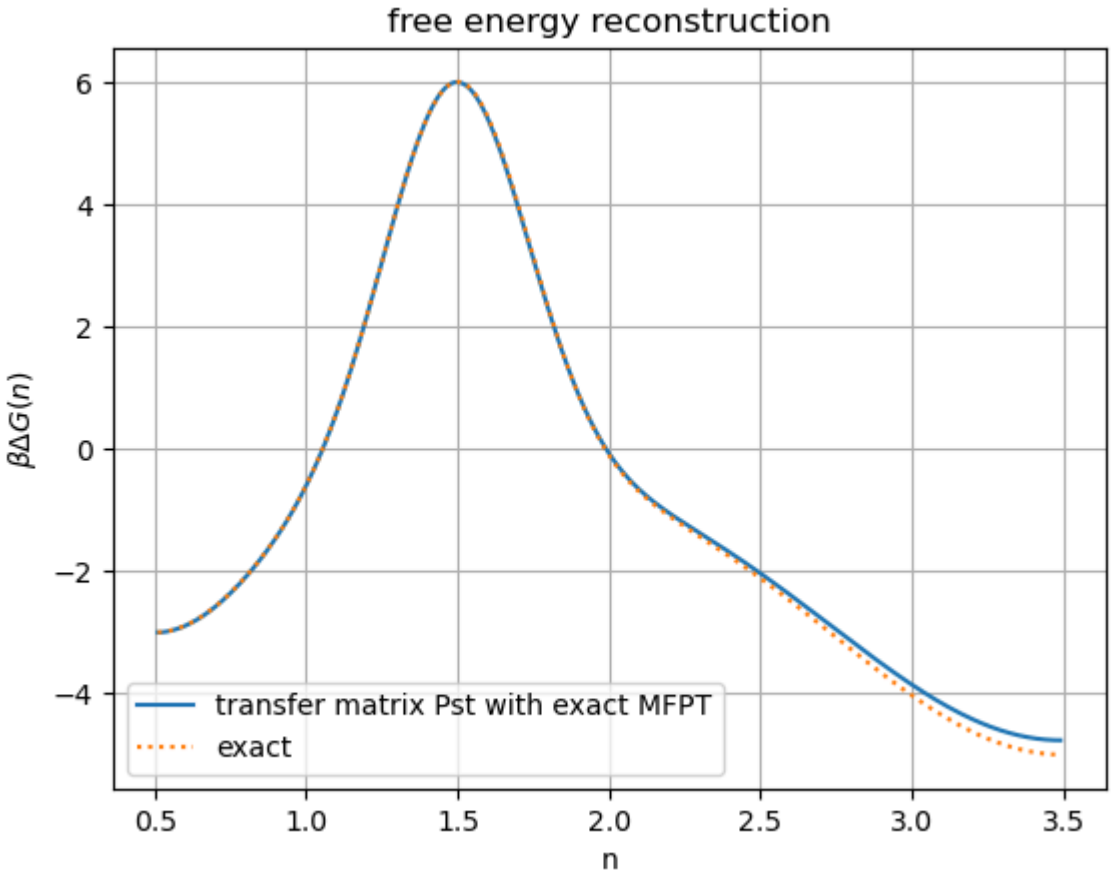


In [35]:
```
# Except for the absorbing boundary Pst(b) = 0, D(x)=0, also avoid at reflec
Bx_arr = np.zeros(N-2)
integral_Pst_arr = np.zeros(N-2)
for i in range(N-2):
    integral_Pst_arr[i], _ = quad(interp_Pst_func, x_arr[1+i], x_arr[-1])
    Bx_arr[i] = -1.0/interp_Pst_func(x_arr[1+i])*(integral_Pst_arr[i]-(mfpt_

interp_invertBx_func = interp1d(x_arr[1:-1], 1.0/Bx_arr, kind='cubic') # fil

plt.semilogy(x_arr[1:-1], interp_invertBx_func(x_arr[1:-1]), label="interpol
plt.semilogy(x_arr[1:-1], 1.0/Bx_arr, '--', label="raw data set")
# Plot formatting
plt.xlabel('n')
plt.ylabel('interpolated 1/B(x)')
plt.title('1/B(x)')
plt.legend()
plt.grid()
```

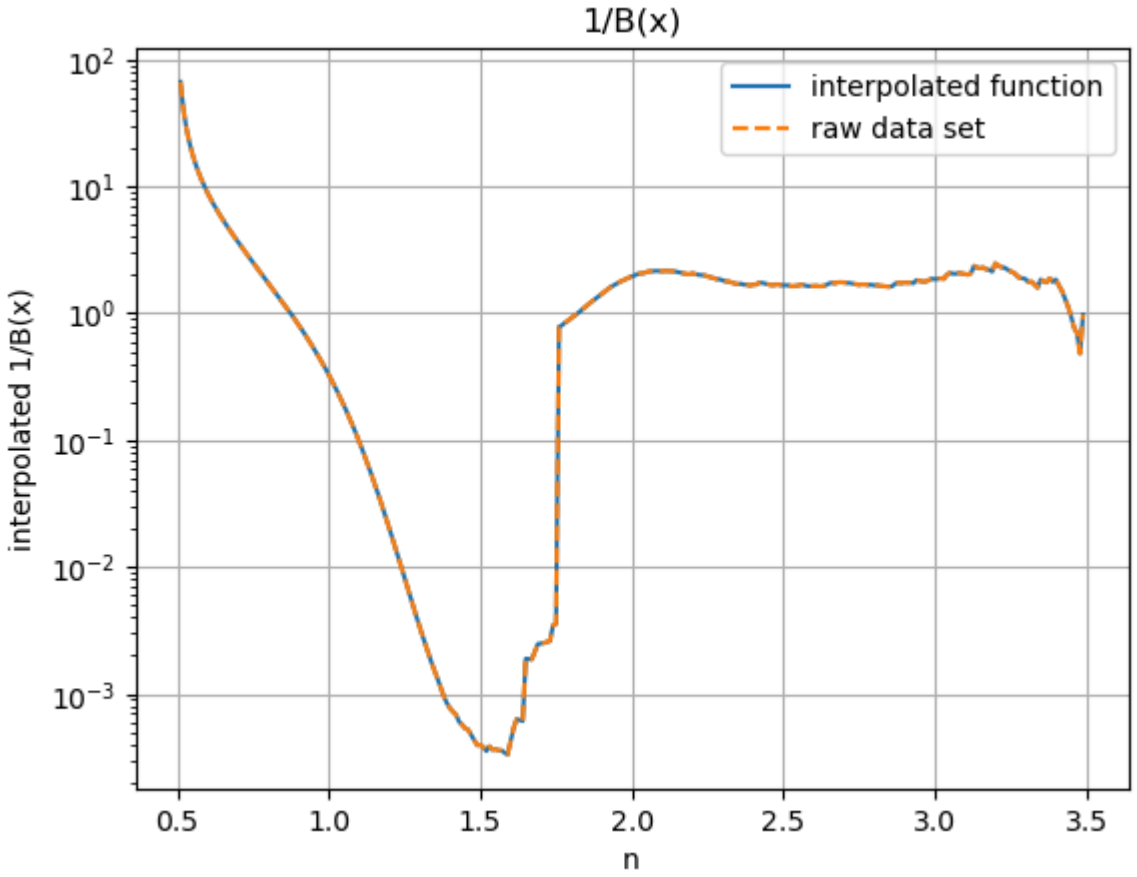


In [36]:
```
integral_invertBx_arr = np.zeros(N-2)
beta_Grec2_arr = np.zeros(N-2)

for i in range(N-2):
    # Here x0 is x_arr[1]
    integral_invertBx_arr[i], _ = quad(interp_invertBx_func, x_arr[1], x_arr
    beta_Grec2_arr[i] = beta_U(x_arr[1])+np.log(Bx_arr[i]/Bx_arr[0])-integra

print(_)

plt.plot(x_arr[1:-1], beta_Grec2_arr, label="transfer matrix Pst with exact
plt.plot(x_arr[1:-1], beta_U(x_arr[1:-1]), ':', label="exact")

# Plot formatting
plt.xlabel('n')
plt.ylabel('$ \\beta \Delta G(n) $')
plt.title('free energy reconstruction')
plt.legend()
plt.grid()
```

```
7.050729470223247e-08
```

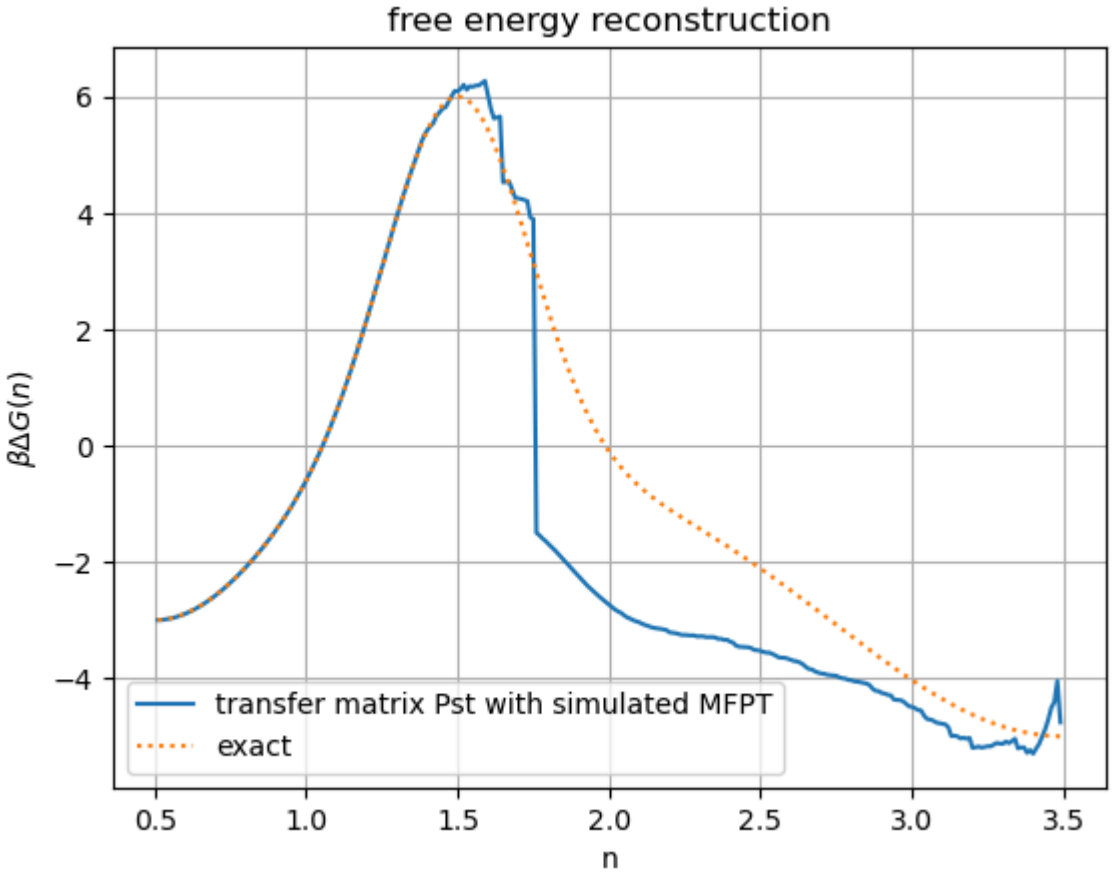


In [37]:
```
# Except for the absorbing boundary Pst(b) = 0, D(x)=0, also avoid at reflec
Bx_arr = np.zeros(N-2)
integral_Pst_arr = np.zeros(N-2)
for i in range(N-2):
    integral_Pst_arr[i], _ = quad(interp_Pst_func, x_arr[1+i], x_arr[-1])
    Bx_arr[i] = -1.0/interp_Pst_func(x_arr[1+i])*(integral_Pst_arr[i]-(mfpt_

interp_invertBx_func = interp1d(x_arr[1:-1], 1.0/Bx_arr, kind='cubic') # fil

plt.semilogy(x_arr[1:-1], interp_invertBx_func(x_arr[1:-1]), label="interpol
plt.semilogy(x_arr[1:-1], 1.0/Bx_arr, '--', label="raw data set")
# Plot formatting
plt.xlabel('n')
plt.ylabel('interpolated 1/B(x)')
plt.title('1/B(x)')
plt.legend()
plt.grid()
```



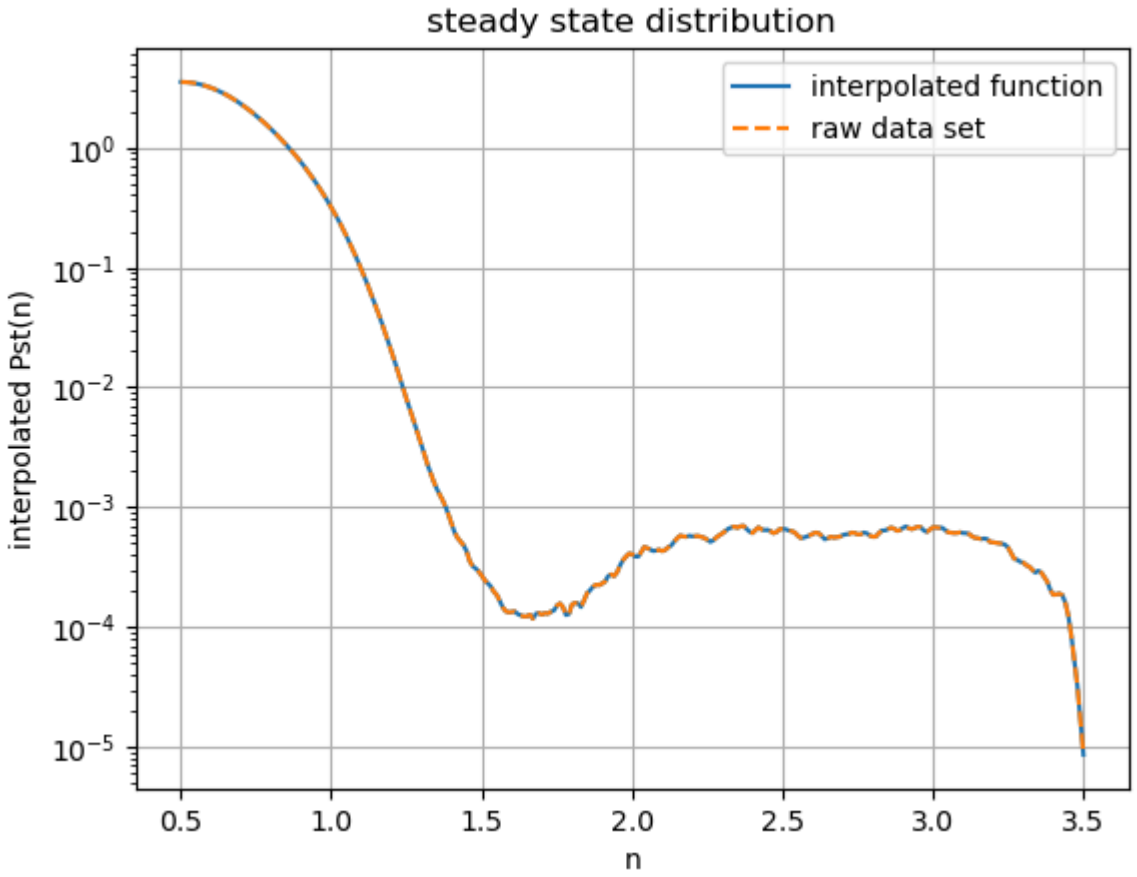
In [38]:
```
integral_invertBx_arr = np.zeros(N-2)
beta_Grec2_arr = np.zeros(N-2)

for i in range(N-2):
    # Here x0 is x_arr[1]
    integral_invertBx_arr[i], _ = quad(interp_invertBx_func, x_arr[1], x_arr
    beta_Grec2_arr[i] = beta_U(x_arr[1])+np.log(Bx_arr[i]/Bx_arr[0])-integra

print(_)

plt.plot(x_arr[1:-1], beta_Grec2_arr, label="transfer matrix Pst with simula
plt.plot(x_arr[1:-1], beta_U(x_arr[1:-1]), ':', label="exact")

# Plot formatting
plt.xlabel('n')
plt.ylabel('$ \\beta \Delta G(n) $')
plt.title('free energy reconstruction')
plt.legend()
plt.grid()
```

```
/tmp/ipykernel_789248/683931227.py:6: IntegrationWarning: The maximum number
of subdivisions (50) has been achieved.
  If increasing the limit yields no improvement it is advised to analyze
  the integrand in order to determine the difficulties.  If the position of
a
  local difficulty can be determined (singularity, discontinuity) one will
  probably gain from splitting up the interval and calling the integrator
  on the subranges.  Perhaps a special-purpose integrator should be used.
  integral_invertBx_arr[i], _ = quad(interp_invertBx_func, x_arr[1], x_arr[1
+i])
0.004042818371613879
```



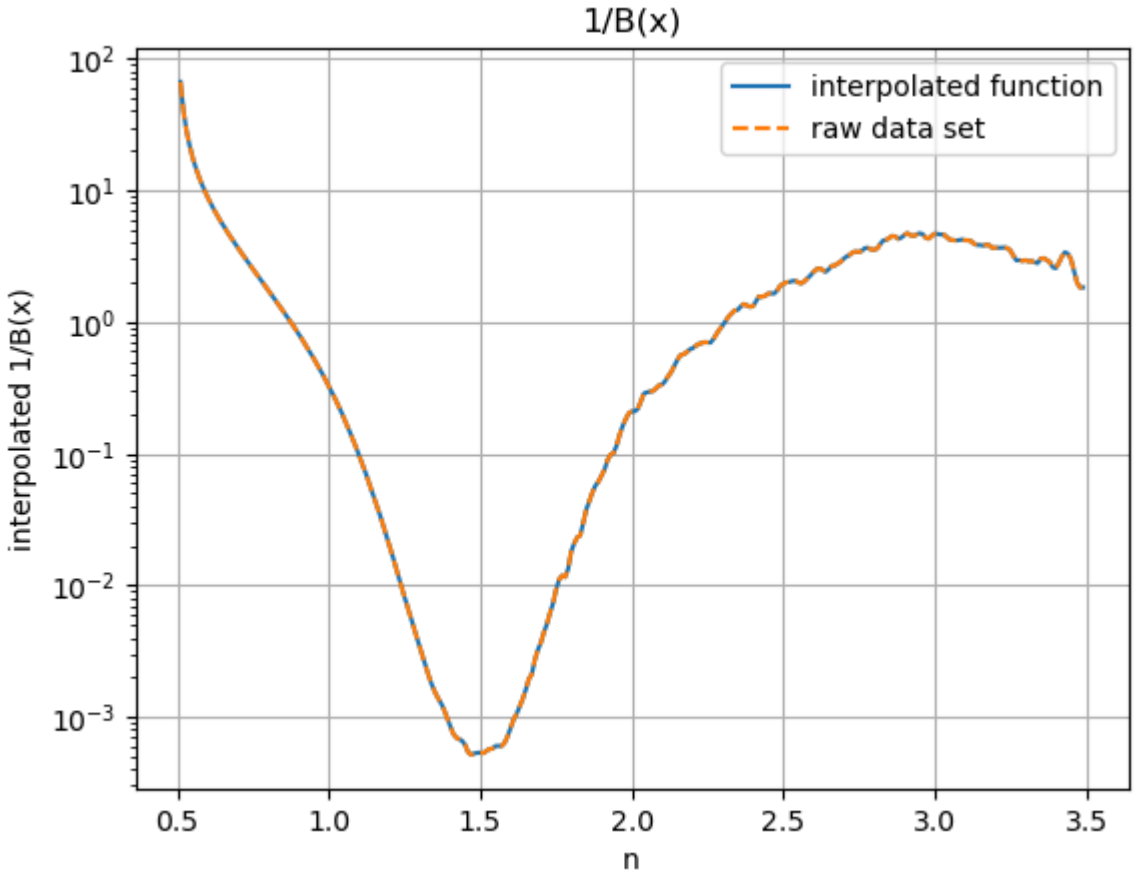
(3) Simulated Steady State Distribution with exact MFPT and then with simulated MFPT

In [39]:
```
interp_simu_Pst_func = interp1d(simu_x_arr, Pst_n, kind='cubic', fill_value=
# interp_Pst_func = PchipInterpolator(b_arr, Pst_arr)

plt.semilogy(simu_x_arr, interp_simu_Pst_func(simu_x_arr), label="interpolat
plt.semilogy(simu_x_arr, Pst_n, '--', label="raw data set")
# Plot formatting
plt.xlabel('n')
plt.ylabel('interpolated Pst(n)')
plt.title('steady state distribution')
plt.legend()
plt.grid()
```



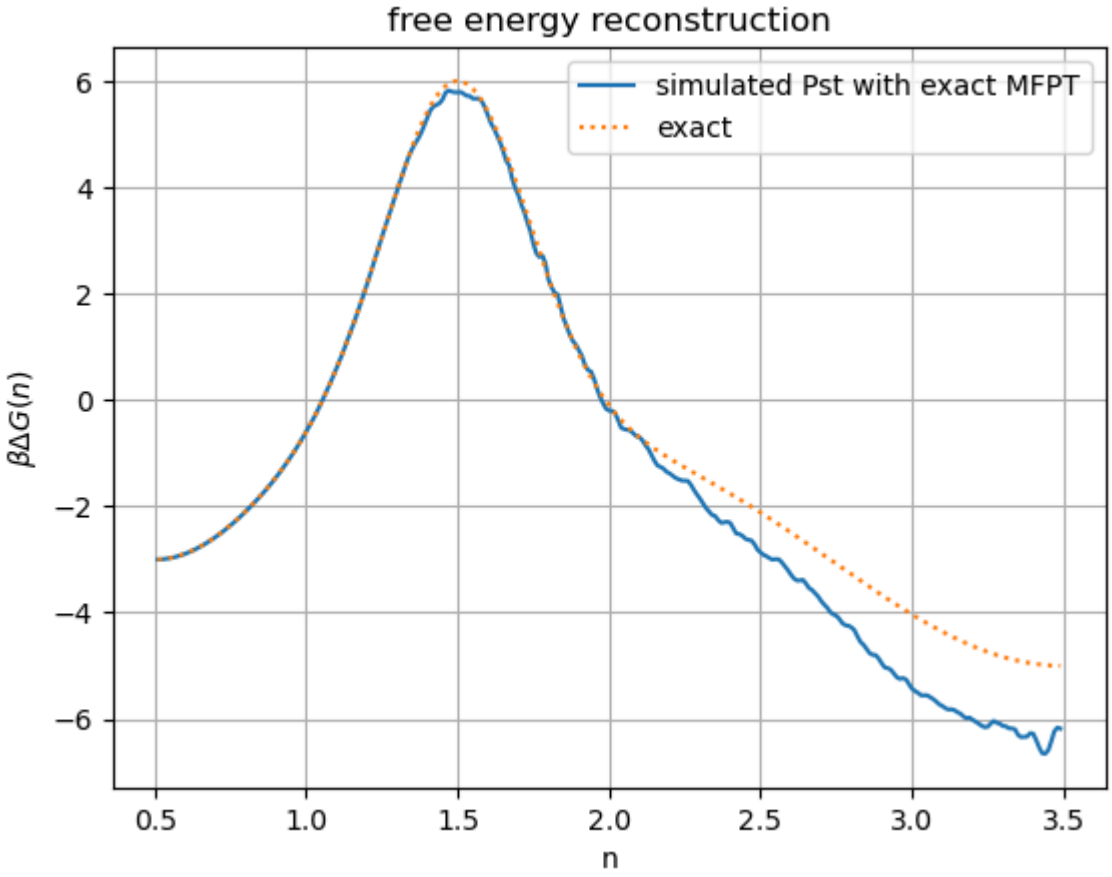
In [40]:
```
# Except for the absorbing boundary Pst(b) = 0, D(x)=0, also avoid at reflec
simu_Bx_arr = np.zeros(N-2)
integral_Pst_arr = np.zeros(N-2)
for i in range(N-2):
    integral_Pst_arr[i], _ = quad(interp_simu_Pst_func, simu_x_arr[1+i], sim
    simu_Bx_arr[i] = -1.0/Pst_n[1+i]*(integral_Pst_arr[i]-(mfpt_arr[-1]-mfpt

interp_simu_invertBx_func = interp1d(x_arr[1:-1], 1.0/simu_Bx_arr, kind='cub

plt.semilogy(simu_x_arr[1:-1], interp_simu_invertBx_func(x_arr[1:-1]), label
plt.semilogy(simu_x_arr[1:-1], 1.0/simu_Bx_arr, '--', label="raw data set")
# Plot formatting
plt.xlabel('n')
plt.ylabel('interpolated 1/B(x)')
plt.title('1/B(x)')
plt.legend()
plt.grid()
```

```
/tmp/ipykernel_789248/3951444635.py:5: IntegrationWarning: The maximum numbe
r of subdivisions (50) has been achieved.
  If increasing the limit yields no improvement it is advised to analyze
  the integrand in order to determine the difficulties.  If the position of
a
  local difficulty can be determined (singularity, discontinuity) one will
  probably gain from splitting up the interval and calling the integrator
  on the subranges.  Perhaps a special-purpose integrator should be used.
  integral_Pst_arr[i], _ = quad(interp_simu_Pst_func, simu_x_arr[1+i], simu_
x_arr[-1])
```



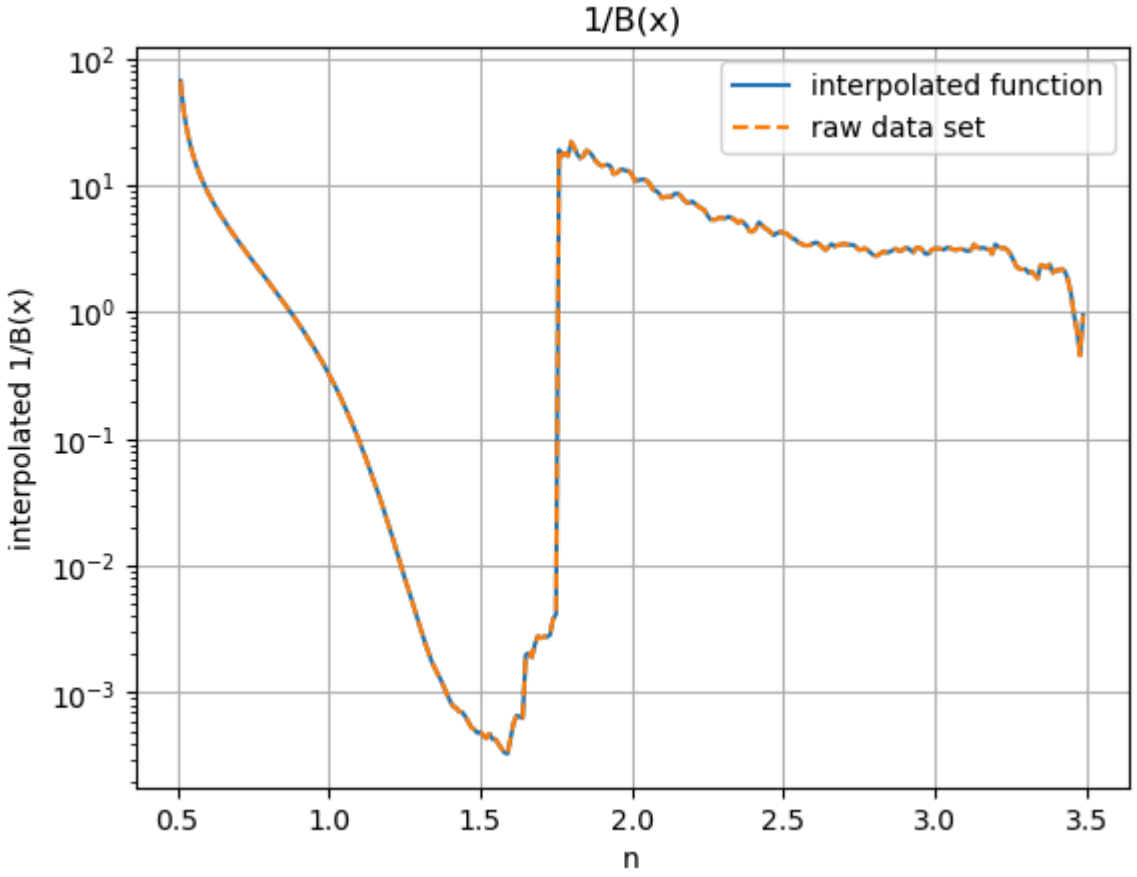
In [41]:
```
integral_invertBx_arr = np.zeros(N-2)
beta_Grec2_arr = np.zeros(N-2)

for i in range(N-2):
    # Here x0 is x_arr[1]
    integral_invertBx_arr[i], _ = quad(interp_simu_invertBx_func, simu_x_arr
    beta_Grec2_arr[i] = beta_U(simu_x_arr[1])+np.log(simu_Bx_arr[i]/simu_Bx_

print(_)

plt.plot(simu_x_arr[1:-1], beta_Grec2_arr, label="simulated Pst with exact M
plt.plot(simu_x_arr[1:-1], beta_U(x_arr[1:-1]), ':', label="exact")

# Plot formatting
plt.xlabel('n')
plt.ylabel('$ \\beta \Delta G(n) $')
plt.title('free energy reconstruction')
plt.legend()
plt.grid()
```

```
/tmp/ipykernel_789248/2109196022.py:6: IntegrationWarning: The maximum numbe
r of subdivisions (50) has been achieved.
  If increasing the limit yields no improvement it is advised to analyze
  the integrand in order to determine the difficulties.  If the position of
a
  local difficulty can be determined (singularity, discontinuity) one will
  probably gain from splitting up the interval and calling the integrator
  on the subranges.  Perhaps a special-purpose integrator should be used.
  integral_invertBx_arr[i], _ = quad(interp_simu_invertBx_func, simu_x_arr
[1], simu_x_arr[1+i])
0.0011142226443684564
```



In [42]:
```
# Except for the absorbing boundary Pst(b) = 0, D(x)=0, also avoid at reflec
simu_Bx_arr = np.zeros(N-2)
integral_Pst_arr = np.zeros(N-2)
for i in range(N-2):
    integral_Pst_arr[i], _ = quad(interp_simu_Pst_func, simu_x_arr[1+i], sim
    simu_Bx_arr[i] = -1.0/Pst_n[1+i]*(integral_Pst_arr[i]-(mfpt_simu_arr[-1]

interp_simu_invertBx_func = interp1d(simu_x_arr[1:-1], 1.0/simu_Bx_arr, kinc

plt.semilogy(simu_x_arr[1:-1], interp_simu_invertBx_func(x_arr[1:-1]), label
plt.semilogy(simu_x_arr[1:-1], 1.0/simu_Bx_arr, '--', label="raw data set")
# Plot formatting
plt.xlabel('n')
plt.ylabel('interpolated 1/B(x)')
plt.title('1/B(x)')
plt.legend()
plt.grid()
```

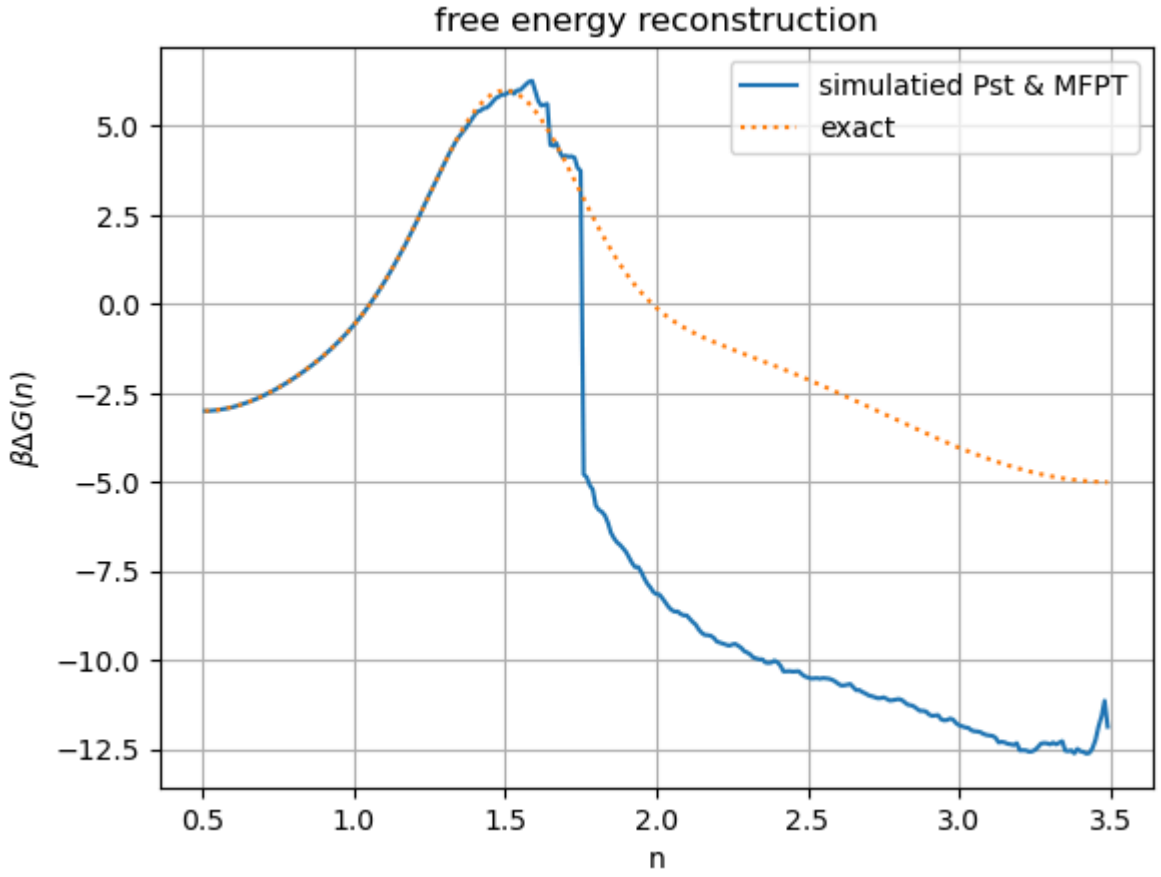
```
/tmp/ipykernel_789248/1053423012.py:5: IntegrationWarning: The maximum numbe
r of subdivisions (50) has been achieved.
  If increasing the limit yields no improvement it is advised to analyze
  the integrand in order to determine the difficulties.  If the position of
a
  local difficulty can be determined (singularity, discontinuity) one will
  probably gain from splitting up the interval and calling the integrator
  on the subranges.  Perhaps a special-purpose integrator should be used.
  integral_Pst_arr[i], _ = quad(interp_simu_Pst_func, simu_x_arr[1+i], simu_
x_arr[-1])
```



In [43]:
```
integral_invertBx_arr = np.zeros(N-2)
beta_Grec2_arr = np.zeros(N-2)

for i in range(N-2):
    # Here x0 is x_arr[1]
    integral_invertBx_arr[i], _ = quad(interp_simu_invertBx_func, simu_x_arr
    beta_Grec2_arr[i] = beta_U(simu_x_arr[1])+np.log(simu_Bx_arr[i]/simu_Bx_

print(_)

plt.plot(simu_x_arr[1:-1], beta_Grec2_arr, label="simulatied Pst & MFPT")
plt.plot(simu_x_arr[1:-1], beta_U(x_arr[1:-1]), ':', label="exact")

# Plot formatting
plt.xlabel('n')
plt.ylabel('$ \\beta \Delta G(n) $')
plt.title('free energy reconstruction')
```

```
plt.legend()
plt.grid()
```

```
/tmp/ipykernel_789248/3105494345.py:6: IntegrationWarning: The maximum numbe
r of subdivisions (50) has been achieved.
  If increasing the limit yields no improvement it is advised to analyze
  the integrand in order to determine the difficulties.  If the position of
a
  local difficulty can be determined (singularity, discontinuity) one will
  probably gain from splitting up the interval and calling the integrator
  on the subranges.  Perhaps a special-purpose integrator should be used.
  integral_invertBx_arr[i], _ = quad(interp_simu_invertBx_func, simu_x_arr
[1], simu_x_arr[1+i])
0.024111570733485937
```



In [ ]: