Activated rates and the saddle-node transition

(Sethna, "Entropy, Order Parameters, and Complexity", ex. 12.26)

The hints below will help you solve parts (i) and (j) of this question, where we will numerically evaluate the slowest decaying mode and barrier crossing time by computing eigenstates of the "quantum" Hamiltonian.

In the Mathematica hints, we shall solve the differential equations directly. In the Python hints, we shall instead construct a Hamiltonian matrix by discretizing space into segments of length dx, and then finding the lowest eigenvalue of that Hamiltonian.

Import packages

In [ ]:
```
# Sometimes gives interactive new windows
# Must show() after plot, figure() before new plot
# %matplotlib

# Adds static figures to notebook: good for printing
%matplotlib inline

# Interactive windows inside notebook! Must include plt.figure() between plots
# %matplotlib notebook

# Better than from numpy import *, but need np.sin(), np.array(), plt.plot(), etc.
import numpy as np
import matplotlib.pyplot as plt
from scipy.sparse import diags
from scipy.linalg import eig_banded
from scipy import special
```

Define functions to compute the exact scaling function $\mathcal{T}(\alpha)$ and escape time τ. For simplicity, in the numerical portion of the exercise we shall assume $\eta = g = a = 1$ throughout.

In [ ]:
```
def Texact(alpha):
    """
    Returns the exact scaling function T(alpha)
    """
    return 2**(1/3) * np.pi**2 * (special.airy(-2**(2/3) * alpha)[0]**2
                                  + special.airy(-2**(2/3) * alpha)[2]**2)

def tauExact(alpha, g=1):
    """
    Returns the exact escape time for a cubic potential
    """
    return g**(-2/3) * Texact(alpha)
```

Define functions to evaluate the potential $V(x)$ and effective quantum potential $V_{\mathrm{eff}}(\mathrm{x})$ and compute a discretized Hamiltonian matrix H. Again, we assume $\eta = g = a = 1$. We will use this matrix to compute the slowest decaying mode for parts (i) and (j).

In [ ]:

```
def V(x, eps0, a=1, eta=1):
    """
    Returns the cubic potential -eta ax^3/3 - eta eps0 x
    """
    return  ...


def Veff(x, eps0, g=1, a=1):
    """
    Returns the effective quantum potential evaluated at x.
    Calculate Veff by pluging the cubic potential into the
    effective potential you found in part (g). You should find
    (ax^2 + eps0)^2/(2g^2) + ax (notice this is independent of eta)
    """
    return ...


def hamiltonian(eps0, g=1, a=1, x0=5, dx=0.001):
    """
    Returns a discretized hamiltonian matrix for numerical eigendecomposition
    +/-x0 - finite boundary conditions
    dx - discrete grid size
    """
    N = round(2*x0/dx) #total number of elements on grid (boundary at +/- x0)
    x = np.arange(-x0,x0,dx) #make discrete grid

    ## Finite difference Kinetic Energy
    #d^2/dx^2 is a matrix with -2 on the diagonal and 1 on the super/subdiagonal divide
    #In the first list put the elements: -2, 1, and 1. In the second list put the corre
    #0 = diagonal, 1 = superdiagonal, -1 = subdiagonal
    T = -1/2*g**2*diags([..., ..., ...], [...,..., ...], shape=(N, N))/(...)**2

    ## Potential Energy
    #Effective quantum potential on discrete grid
    Veff_x = Veff(x, eps0, g, a)
    #Matrix potential (Veff_x on diagonal)
    Veff_mat = diags([...], [0])

    H = T + Veff_mat

    return H.toarray()

def eigenSystem(eps0, g=1, a=1, x0=5, dx=0.001, Nstates=1):
    """
    Returns the Nstates lowest energy eigenvalues and associated eigenstates for
    the quantum system corresponding to the Fokker-Plank equation with a cubic potentia
    """
    H = hamiltonian(eps0, g, a, x0, dx)

    banded = np.array([np.diagonal(H), np.append(np.diagonal(H,1),0)])
    eigs = eig_banded(banded, select="i", select_range=[0,0], lower=True)
    ##eigh_tridiagonal(np.diagonal(H), np.diagonal(H,1), select="i", select_range=[0,0]
    # If you have scipy version 1.0 or newer, eigh_tridiagonal (add from scipy.linalg i
    # will give you slightly faster performance

    return eigs
```

(i) *For the cubic potential, numerically compute the eigenstate of the transformed diffusion equation with smallest eigenvalue for $\alpha = -2$. What does the eigenvalue predict for the lifetime? How nearly does it agree with τ from eqn (11)? Using the corresponding eigenstate, plot the slowest decaying mode $\rho_0(x) = \sqrt{\rho^*}\sigma_0$, normalized to one, along with the Boltzmann distribution $\rho^*(x)/Z$. How well can you match the two inside the well, by varying Z?*

Choose ϵ_0 so that $\alpha = -2$ (since we have set $\sigma = \gamma = 1$ the relationship between ϵ_0 and α is very simple)

In [ ]:
```
# Compute eigenvalue and eigenstate
x0 = 5
dx=0.001
eps0 = ...
alpha = ...

print(chr(945),"=", alpha)

val, vec = eigenSystem(...)
val = val[0]
vec = np.transpose(vec)[0]
```

In [ ]:
```
# Plot the potential, notice where the well and barrier are located
x = np.arange(...)

plt.plot(x, ...)
plt.xlabel("x",size=20)
plt.ylabel("V(x)",size=20)
plt.ylim([-10,10])

plt.show()
```

In [ ]:
```
# Compare eigenvalue approximation for escape time to analytical result

tauNumerical = ...
tauAnalytical = tauExact(...)

# Percent difference between numerical and analytical result
diff = 100*abs(...)/tauAnalytical

print(tauNumerical)
print(tauAnalytical)

print(str(round(diff,4))+"% difference")
```

(i) *... What does the eigenvalue predict for the lifetime? How nearly does it agree with τ from eqn (11)?*

Using the corresponding eigenstate, plot the slowest decaying mode $\rho_0(x) = \sqrt{\rho^*}\sigma_0$, normalized to one, along with the Boltzmann distribution $\rho^*(x)/Z$. ...

In [ ]:
```
# Compute slowest decaying mode

# Boltzmann distribution for cubic potential
boltz = np.array(np.exp(...))

# Compute slowest decaying mode from eigenstate 'vec'
mode = (vec*np.sqrt(...))

# Plot the unnormalized mode
plt.plot(x,mode)
plt.legend([r"$\rho_0$"],prop={'size': 15})
plt.xlabel("x",size=20)
plt.ylabel("Density",size=20)

plt.show()
```

In [ ]:
```
# Now normalize your slowest decaying mode
# We only know the slowest decaying mode on a discrete grid, so the norm is given by
# adding up the points
# Due to numerical errors your slowest decaying mode may blow up at one of the boundari
# numerical eigenstate doesn't exactly cancel the blow up from the Boltzmann distributi
# If this occurs, use cutoff to restrict the normalization calculation to [-xLim, xLim]
xLim = ...
# boltz, x, and mode are lists. 'cutoff' gives the range of list elements of interest
cutoff = round((x0-abs(xLim))/dx)+1
norm = sum(mode[cutoff:-cutoff])*dx

# Adjust Z manually so that the Boltzmann distribution best matches the slowest decayin
# inside the well or approximate Z by normalizing over the inside of the potential well
# If you choose the later option, you will want to restrict the normalization to the do
# Use your plot of the potential above to choose xMin and xMax
xMin = ...
xMax = ...

# Pick range corresponding to inside of the wellcutoff1 = round((x0-abs(xMin))/dx)
cutoff1 = round((x0-abs(xMin))/dx)
cutoff2 = round((x0-abs(xMax))/dx)

Z = sum(boltz[...:-...])*(...)


# Compare Boltzmann distribution to mode
plt.plot(x[cutoff:-cutoff],mode[cutoff:-cutoff]/norm)
plt.plot(x[cutoff:-cutoff],boltz[cutoff:-cutoff]/Z)
plt.ylim([0,1.5])
plt.legend([r"$\rho_0$",r"$\rho^*$"],prop={'size': 15})
plt.xlabel("x",size=20)
plt.ylabel("Density",size=20)


plt.show()
```

(i) *... How well can you match the two inside the well, by varying Z?*

(j) *Repeating the above steps, compute the slowest decaying mode for $\alpha = 0$. Does the eigenvalue agree with the prediction from τ? Why or why not? Plot the slowest decaying mode. Where is it peaked? Why do the particles that are slowest to escape sit there?*

In [ ]:
```
# Compute eigenvalue and eigenstate
# Choose eps0_SN so that alpha=0. Again, for simplicity use g=a=1

x0 = 5
dx= 0.001
eps0_SN = ...
alphaSN = ...


print(chr(945),"=",alphaSN)

valSN, vecSN = eigenSystem(...)
valSN = valSN[0]
vecSN = np.transpose(vecSN)[0]
```

In [ ]:
```
# Plot the potential at the saddle node bifurcation. Notice where the slope is minimum
x = np.arange(...)

plt.plot(x, ...)
plt.xlabel("x",size=20)
plt.ylabel("V(x)",size=20)
plt.ylim([-10,10])

plt.show()
```

In [ ]:
```
# Compare eigenvalue approximation for escape time to analytical result

tauNumericalSN = 1/(...)
tauAnalyticalSN = tauExact(...)

#percent difference between numerical and analytical result
diffSN = 100*abs(...)/tauAnalyticalSN

print(tauNumericalSN)
print(tauAnalyticalSN)

print(str(round(diffSN,4))+"% difference")
```

(j) *... Does the eigenvalue agree with the prediction from τ? Why or why not? ...*

In [ ]:
```
# Compute slowest decaying mode

# Boltzmann distribution for cubic potential
boltzSN = np.array(...)

# Compute slowest decaying mode from eigenstate 'vec'
modeSN = (...*np.sqrt(...))

# Plot the unormalized mode
plt.plot(x, modeSN)
plt.legend([r"$\rho_0$",r"$\rho_\mathrm{well}$"],prop={'size': 15})
plt.xlabel("x",size=20)
plt.ylabel("Density",size=20)

plt.show()
```

In [ ]:
```
# Now normalize your slowest decaying mode
# Due to numerical errors your slowest decaying mode may blow up at one of the boundari
# numerical eigenstate doesn't exactly cancel the blow up from the Boltzmann distributi
# If this occurs, use cutoff to restrict the normalization calculation to [-xLim, xLim]
xLim = ...
cutoff = round((x0-abs(xLim))/dx)+1
normSN = sum(...)*dx

plt.plot(x[cutoff:-cutoff],modeSN[cutoff:-cutoff]/normSN)
plt.legend([r"$\rho_0$",r"$\rho_\mathrm{well}$"],prop={'size': 15})
plt.xlabel("x",size=20)
plt.ylabel("Density",size=20)

plt.show()
```

(j) *... the slowest decaying mode. Where is it peaked? Why do the particles that are slowest to escape sit there?*

Optional: For a range of positive and negative α, plot the eigenvalue approximation to the scaling function $\mathcal{T}(\alpha) = (g\,a)^{2/3}\tau = \tau$ (when $a = g = 1$) and compare to the exact formula eqn (11).

In [ ]:
```
x0 = 5
dx = 0.005

#List of alpha
alphaList = np.array([... for eps0 in np.arange(...,...,0.1)])
eigApprox = [.../ eigenSystem(eps0, x0=x0, dx=dx)[0][0] for eps0 in np.arange(..., ...,

analytical = ...(alphaList)
```

In [ ]:
```
plt.plot(alphaList, analytical, '-b')
plt.plot(alphaList, eigApprox, 'ok')

plt.legend(["Analytical", "Eigenvalues"],prop={'size': 15})


plt.xlabel(r'$\alpha$', size=20)
plt.ylabel(r'$\mathcal{T}(\alpha)$', size=20)

plt.yscale('log')
```