In [1]:

```
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np
from math import *
from scipy.optimize import minimize
from scipy.optimize import fsolve
import scipy.sparse.linalg as spl
from scipy.integrate import simps
```

Problem 1

(a) Steepest Descent

In [2]:

```
# Rosenbrock function
def rosenbrock(x, y):
    return 100*((y-x*x)**2) + (1-x)**2
```

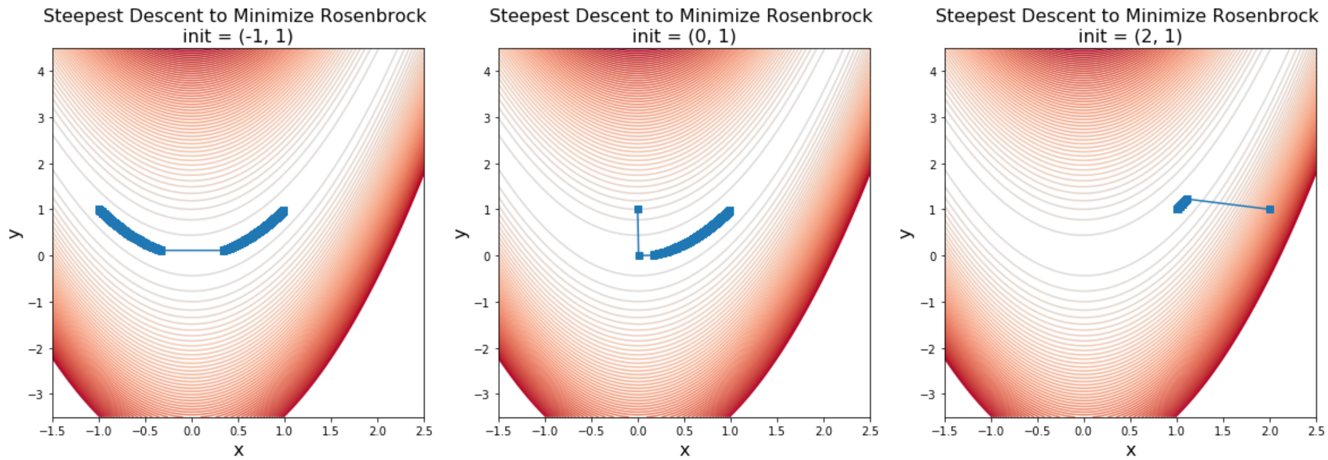
In [3]:

```
eta0 = 0.01
inits = [(-1, 1), (0, 1), (2, 1)]
x_paths_sd = []
y_paths_sd = []

for x0, y0 in inits: # loop thru the 3 starting points
    x_path = []
    y_path = []
    absolute_step_size = 999
    eta_c = eta0
    step_count = 0
    value = rosenbrock(x0, y0)

    x_c, y_c = x0, y0
    x_path.append(x_c)
    y_path.append(y_c)
    while absolute_step_size >= 1e-8 and step_count < 2000:
        # Calculate the direction of steepest descent (-fx, -fy)
        fx_minus = 400*x_c*(y_c-x_c*x_c) + 2*(1-x_c)
        fy_minus = -200*(y_c-x_c*x_c)

        # Minimize the 1D function of line search along (-fx, -fy)
        def line_search(eta):
            x_next = x_c + eta*fx_minus
            y_next = y_c + eta*fy_minus
            return rosenbrock(x_next, y_next)

        eta_best = minimize(line_search, eta_c).x
        step = (eta_best*fx_minus, eta_best*fy_minus)
        x_c += step[0]
        y_c += step[1]
        value = rosenbrock(x_c, y_c)[0]
        x_path.append(x_c[0])
        y_path.append(y_c[0])
        absolute_step_size = np.linalg.norm((eta_best*fx_minus, eta_best*fy_minus), ord=2)
        step_count += 1
    print('starting point= ({},{}), #_iterations= {}, min= {}, \nending point= ({},{})\n'.format(x0, y0, step_count, value, x_c[0],
    x_paths_sd.append(x_path)
    y_paths_sd.append(y_path)
```

```
starting point= (-1,1), #_iterations= 2000, min= 0.00017111059610313366,
ending point= (0.9869219360370596,0.9740422645641981)

starting point= (0,1), #_iterations= 2000, min= 0.00011511683580099871,
ending point= (0.9892825205281497,0.9786296590722668)

starting point= (2,1), #_iterations= 2000, min= 2.206668410399586e-10,
ending point= (1.0000148413424343,1.0000297462609378)
```

In [4]:

```
x = np.linspace(-1.5, 2.5, 200)
y = np.linspace(-3.5, 4.5, 200)
X, Y = np.meshgrid(x, y)
rosen_2d = rosenbrock(X, Y)
levels = np.linspace(-2000, 2000, 100)

fig, axs = plt.subplots(1, 3, figsize=(20, 6))
for i, ax in enumerate(axs):
    ax.contour(X, Y, rosen_2d, levels=levels, cmap=plt.cm.coolwarm) # Plot contour of the Rosenbrock function
    ax.plot(x_paths_sd[i], y_paths_sd[i], 's-') # Plot the path of steepest descent
    ax.set_xlabel('x', fontsize=16)
    ax.set_ylabel('y', fontsize=16)
    ax.set_xlim(-1.5, 2.5)
    ax.set_ylim(-3.5, 4.5)
    ax.set_title('Steepest Descent to Minimize Rosenbrock \ninit = {}'.format(inits[i]), fontsize=16)
```



(b) Newton's Method

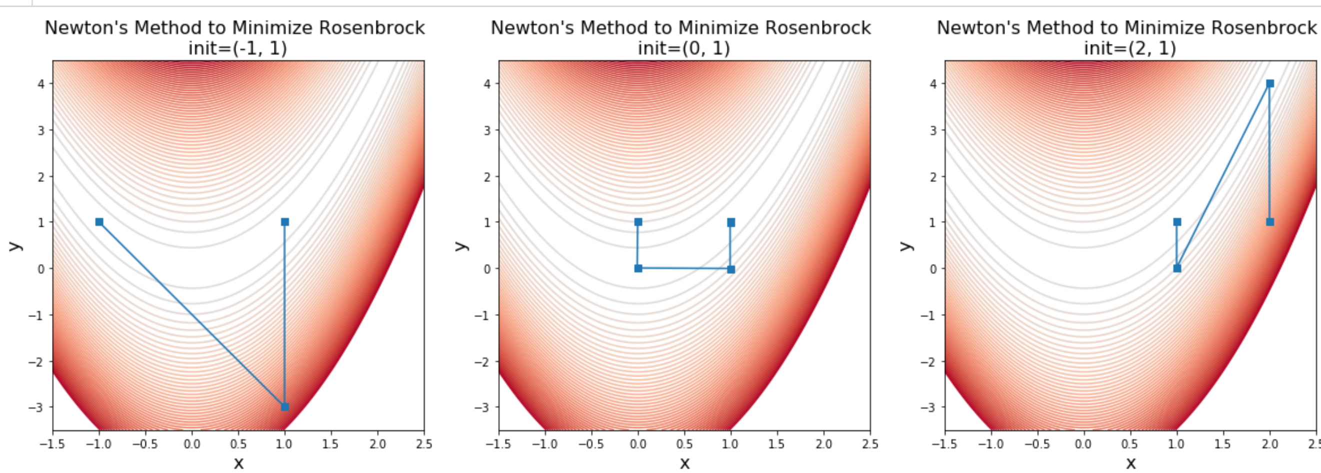
In [5]:

```
def Jacob(x, y): # Jacobian of rosenbrock
    J = np.zeros((2,))
    J[0] = -400*x*y + 400*(x**3) - 2 + 2*x
    J[1] = 200*y - 200*(x**2)
    return J

def Hessian(x, y): # Hessian of rosenbrock
    H = np.zeros((2, 2))
    H[0, 0] = -400*y + 1200*(x**2) + 2
    H[0, 1] = -400*x
    H[1, 0] = -400*x
    H[1, 1] = 200
    return H

inits = [(-1, 1), (0, 1), (2, 1)]
x_paths_newton = []
y_paths_newton = []
for x0, y0 in inits: # loop thru the 3 starting points
    x_path = []
    y_path = []
    absolute_step_size = 999
    step_count = 0
    value = rosenbrock(x0, y0)

    x_c, y_c = x0, y0
    x_path.append(x_c)
    y_path.append(y_c)
    J = Jacob(x0, y0)
    H = Hessian(x0, y0)
    delta = np.linalg.solve(H, -1*J)
    while absolute_step_size >= 1e-8 and step_count < 2000:
        x_c += delta[0]
        y_c += delta[1]
        value = rosenbrock(x_c, y_c)
        x_path.append(x_c)
        y_path.append(y_c)
        absolute_step_size = np.linalg.norm(delta, ord=2)
        step_count += 1

        J = Jacob(x_c, y_c)
        H = Hessian(x_c, y_c)
        delta = np.linalg.solve(H, -1*J)
    print('starting point= ({},{}), #_iterations= {}, min= {}, \nending point= ({},{})\n'.format(x0, y0, step_count, value, x_c, y_
    x_paths_newton.append(x_path)
    y_paths_newton.append(y_path)
```

```
starting point= (-1,1), #_iterations= 3, min= 0.0,
ending point= (1.0,1.0)

starting point= (0,1), #_iterations= 6, min= 8.077935669463161e-28,
ending point= (0.9999999999999716,0.9999999999999432)

starting point= (2,1), #_iterations= 6, min= 1.232595164407831e-30,
ending point= (1.0,0.9999999999999999)
```

In [6]:

```
x = np.linspace(-1.5, 2.5, 200)
y = np.linspace(-3.5, 4.5, 200)
X, Y = np.meshgrid(x, y)
rosen_2d = rosenbrock(X, Y)
levels = np.linspace(-2000, 2000, 100)

fig, axs = plt.subplots(1, 3, figsize=(20, 6))
for i, ax in enumerate(axs):
    ax.contour(X, Y, rosen_2d, levels=levels, cmap=plt.cm.coolwarm) # Plot contour of the Rosenbrock function
    ax.plot(x_paths_newton[i], y_paths_newton[i], 's-') # Plot the path of Newton's method
    ax.set_xlabel('x', fontsize=16)
    ax.set_ylabel('y', fontsize=16)
    ax.set_xlim(-1.5, 2.5)
    ax.set_ylim(-3.5, 4.5)
    ax.set_title('Newton\'s Method to Minimize Rosenbrock \ninit={}'.format(inits[i]), fontsize=16)
```



(c) BFGS

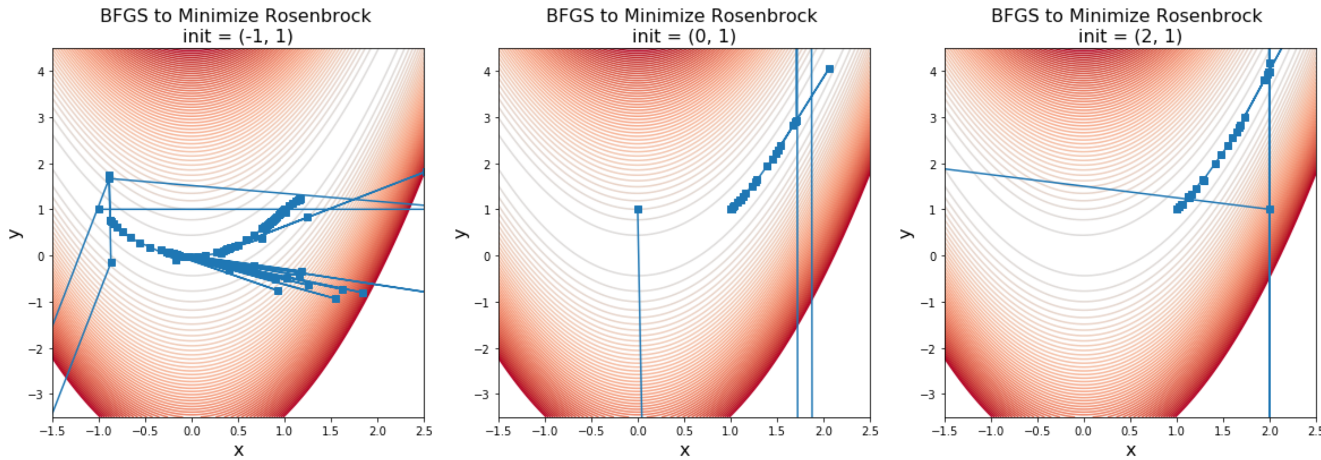
In [7]:

```
# Compute B to approximate the Hessian of f(x)
def compute_delta_B(delta_Jacob, delta, B):
    delta_Jacob_M = np.copy(delta_Jacob).reshape(2, 1)
    delta_M = np.copy(delta).reshape(2, 1)

    M1 = np.matmul(delta_Jacob_M, delta_Jacob_M.T)
    M1 = M1/(np.matmul(delta_Jacob_M.T, delta_M)[0][0])

    M2 = np.matmul(delta_M, delta_M.T)
    M2 = np.matmul(B, M2)
    M2 = np.matmul(M2, B)
    M2 = M2/(np.matmul(np.matmul(delta_M.T, B), delta_M)[0][0])

    return M1 - M2

inits = [(-1, 1), (0, 1), (2, 1)]
x_paths_bfgs = []
y_paths_bfgs = []
for x0, y0 in inits: # loop thru the 3 starting points
    x_path = []
    y_path = []
    absolute_step_size = 999
    step_count = 0
    value = rosenbrock(x0, y0)

    x_c, y_c = x0, y0
    x_path.append(x_c)
    y_path.append(y_c)
    J = Jacob(x0, y0)
    B = np.eye(2) # set B0 = I2
    delta = np.linalg.solve(B, -1*J)
    while absolute_step_size >= 1e-8 and step_count < 2000:
        J_last = J
        x_c += delta[0]
        y_c += delta[1]
        value = rosenbrock(x_c, y_c)
        x_path.append(x_c)
        y_path.append(y_c)
        absolute_step_size = np.linalg.norm(delta, ord=2)
        step_count += 1

        J = Jacob(x_c, y_c)
        delta_Jacob = J - J_last
        delta_B = compute_delta_B(delta_Jacob, delta, B)
        B += delta_B
        delta = np.linalg.solve(B, -1*J)
    print('starting point= ({},{}), #_iterations= {}, min= {}, \nending point= ({},{})\n'.format(x0, y0, step_count, value, x_c, y_
    x_paths_bfgs.append(x_path)
    y_paths_bfgs.append(y_path)
```

```
starting point= (-1,1), #_iterations= 124, min= 1.2818989709841442e-30,
ending point= (0.9999999999999998,0.9999999999999997)

starting point= (0,1), #_iterations= 38, min= 3.0814879110195774e-31,
ending point= (0.9999999999999994,0.9999999999999989)

starting point= (2,1), #_iterations= 45, min= 0.0,
ending point= (1.0,1.0)
```

In [8]:

```python
x = np.linspace(-1.5, 2.5, 200)
y = np.linspace(-3.5, 4.5, 200)
X, Y = np.meshgrid(x, y)
rosen_2d = rosenbrock(X, Y)
levels = np.linspace(-2000, 2000, 100)

fig, axs = plt.subplots(1, 3, figsize=(20, 6))
for i, ax in enumerate(axs):
    ax.contour(X, Y, rosen_2d, levels=levels, cmap=plt.cm.coolwarm) # Plot contour of the Rosenbrock function
    ax.plot(x_paths_bfgs[i], y_paths_bfgs[i], 's-') # Plot the path of BFGS
    ax.set_xlabel('x', fontsize=16)
    ax.set_ylabel('y', fontsize=16)
    ax.set_xlim(-1.5, 2.5)
    ax.set_ylim(-3.5, 4.5)
    ax.set_title('BFGS to Minimize Rosenbrock \ninit = {}'.format(inits[i]), fontsize=16)
```



Problem 2

(b)

In [9]:

```python
# jump rope params set up
R = 3
w = 1
L = 1
p = 1

# y(x)
def shape(b, x):
    ret = 0
    for i, bi in enumerate(b):
        ret += bi*sin(pi*(i+1)*x/L)
    return ret

# dy/dx
def d_shape_dx(b, x):
    ret = 0
    for i, bi in enumerate(b):
        ret += (i+1)*bi*cos(pi*(i+1)*x/L)
    ret = (pi/L)*ret
    return ret

# partial_lagrangian_lambda
def partial_lagr_lambda(b):
    # composite trapezoid rule with 251 equally-spaced points
    n = 250
    h = L/n
    ret = 0.5*sqrt(1+(d_shape_dx(b, 0)**2)) + 0.5*sqrt(1+(d_shape_dx(b, L)**2))
    for i in range(1, n):
        xi = i*h
        ret += sqrt(1+(d_shape_dx(b, xi)**2))
    ret *= h

    return (ret-R)

# partial_lagrangian_bk
def partial_lagr_bk(b, k, lam):
    # composite trapezoid rule with 251 equally-spaced points
    n = 250
    h = L/n
    ret1 = 0.5*(shape(b, L)**2 * pi/L * k *cos(pi*k) * d_shape_dx(b, L)/sqrt(1+(d_shape_dx(b, L)**2)) )
    for i in range(1, n):
        xi = i*h
        ret1 += 2*shape(b, xi) * sin(pi*k*xi/L) * sqrt(1+(d_shape_dx(b, xi)**2)) + \
            shape(b, xi)**2 * pi/L * k *cos(pi*k*xi/L) * d_shape_dx(b, xi)/sqrt(1+(d_shape_dx(b, xi)**2))
    ret1 *= p*w*w
    ret1 *= h

    ret2 = 0.5*(pi/L * k * d_shape_dx(b, 0)/sqrt(1+(d_shape_dx(b, 0)**2))) + \
        0.5*(pi/L * k * cos(pi*k)* d_shape_dx(b, L)/sqrt(1+(d_shape_dx(b, L)**2)))
    for i in range(1, n):
        xi = i*h
        ret2 += pi/L * k * cos(pi*k*xi/L)* d_shape_dx(b, xi)/sqrt(1+(d_shape_dx(b, xi)**2))
    ret2 *= lam
    ret2 *= h
    return (ret1+ret2)

def grad_lagrangian(p):
    b = p[:-1]
    lam = p[-1]
    equations = []
    for i, bi in enumerate(b):
        equations.append(partial_lagr_bk(b, i+1, lam))
    equations.append(partial_lagr_lambda(b))

    return equations
```

In [10]:

```python
# initial guess b1=1.3, all other components=0
inits1 = np.zeros((21,))
inits1[0] = 1.3
b_lam1 = fsolve(grad_lagrangian, inits1)
b_lam1
```

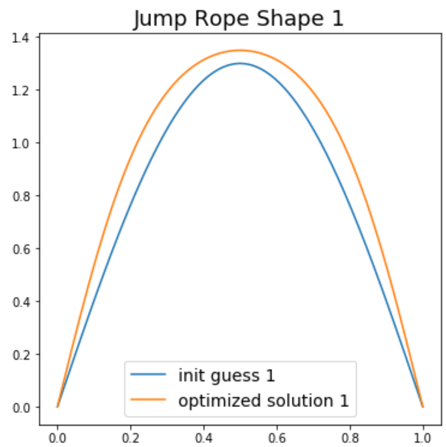
Out[10]:

```
array([  1.44289102e+00,  -8.10521914e-12,   1.00094880e-01,
        -3.08717722e-12,   7.50006855e-03,   1.23656531e-12,
         5.62211237e-04,   7.53751475e-13,   4.21439112e-05,
        -1.04484445e-13,   3.15914916e-06,   4.53378613e-13,
         2.36812546e-07,   3.72462709e-13,   1.77512092e-08,
        -2.75615094e-13,   1.32826918e-09,   7.50383908e-14,
         9.29831429e-11,   3.06261741e-13,  -2.20982903e+00])
```

In [11]:

```python
# plot the initial guess & the optimized solution of y(x)
X_lin = np.linspace(0, L, 251)
Y_init1 = np.array([shape(inits1[:-1], XX) for XX in X_lin])
Y_end1 = np.array([shape(b_lam1[:-1], XX) for XX in X_lin])

fig, ax = plt.subplots(1, 1, figsize=(6, 6))
ax.plot(X_lin, Y_init1, label='init guess 1')
ax.plot(X_lin, Y_end1, label='optimized solution 1')
ax.legend(fontsize=14)
ax.set_title('Jump Rope Shape 1', fontsize=18)
```

Out[11]: <matplotlib.text.Text at 0x11c244e48>



(c)

In [12]:

```python
# initial guess b2=0.7, all other components=0
inits2 = np.zeros((21,))
inits2[1] = 0.7
b_lam2 = fsolve(grad_lagrangian, inits2)
b_lam2
```

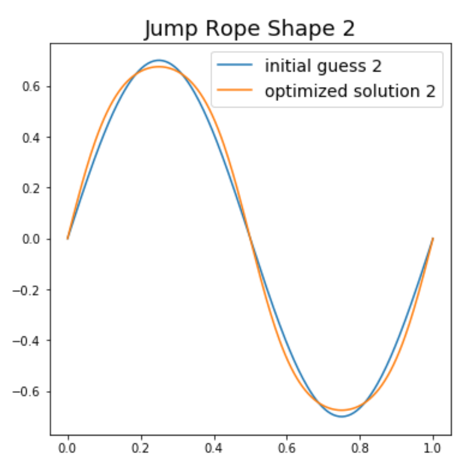
Out[12]:

```
array([ -6.78326864e-12,   7.21445497e-01,   7.53019760e-12,
         1.48672174e-12,  -3.62036242e-13,   5.00473784e-02,
         1.60638303e-12,   8.82583142e-13,   1.00921372e-12,
         3.74980724e-03,   7.71210472e-13,   4.79102307e-13,
         2.32528926e-13,   2.80369427e-04,   5.29007697e-13,
         2.60338184e-13,   1.35530816e-14,   1.94822263e-05,
         1.58498140e-13,   1.03973326e-13,  -5.52457258e-01])
```

In [13]:

```python
# plot the initial guess & the optimized solution of y(x)
X_lin = np.linspace(0, L, 251)
Y_init2 = np.array([shape(inits2[:-1], XX) for XX in X_lin])
Y_end2 = np.array([shape(b_lam2[:-1], XX) for XX in X_lin])

fig, ax = plt.subplots(1, 1, figsize=(6, 6))
ax.plot(X_lin, Y_init2, label='initial guess 2')
ax.plot(X_lin, Y_end2, label='optimized solution 2')
ax.legend(fontsize=14)
ax.set_title('Jump Rope Shape 2', fontsize=18)
```

Out[13]: <matplotlib.text.Text at 0x11bac7358>



Problem 3

(a)

In [14]:

```
# grid setup
n = 1920
dx = 24/n
c = 1/(dx*dx)

# define potential functions
def v0(x):
    return x*x/10

def v1(x):
    return abs(x)

def v2(x):
    return 12*((x/10)**4) - x*x/18 + x/8 + 13/10

def v3(x):
    return 8*abs(abs(abs(x)-1)-1)

v = [v0, v1, v2, v3]

vals_list = []
vecs_list = []
for vi in v:
    # build the matrix that represents discretized phi(x)
    A = np.zeros((n+1, n+1))
    A[0:n, 1:n+1] = (-c) * np.eye(n)
    B = np.zeros((n+1, n+1))
    B[1:n+1, 0:n] = (-c) * np.eye(n)
    M = (2*c)*np.eye(n+1) + A + B
    for i in range(n+1):
        xi = i*dx - 12
        M[i, i] += vi(xi)

    # compute the 5 lowest eigenvalues & corresponding eigenvectors
    vals, vecs = spl.eigs(M, k=5, which='SM')
    vals_list.append(vals)
    vecs_list.append(vecs)

```

In [15]:

```
len(vals_list), len(vecs_list)
```

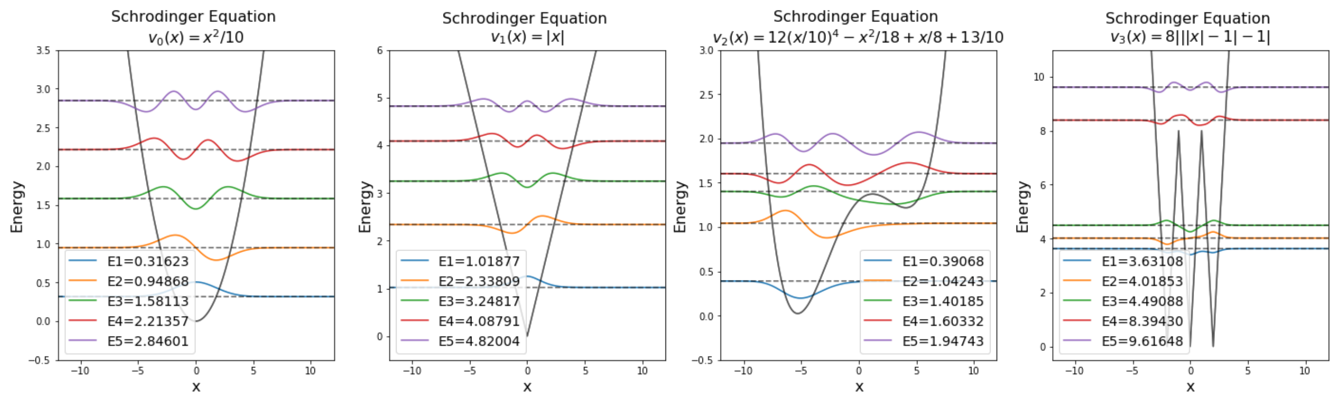
Out[15]: (4, 4)

In [16]:

```
# Record eigen_modes for each v(x)
X_lin = X_lin = np.linspace(-12, 12, n+1)
Y_vi = np.zeros((len(v), n+1))

for i, vi in enumerate(v):
    Y_vi[i] = np.array([vi(XX) for XX in X_lin])

# eigen_modes for each v(x)
eigen_modes = np.zeros((len(v), n+1, 5))
eigen_ylines = np.zeros((len(v), n+1, 5))
for j, (vals, vecs) in enumerate(zip(vals_list, vecs_list)):
    for i in range(5):
        Ei = vals[i].real
        vi = vecs[:, i].real
        yi = 3*vi + Ei
        eigen_modes[j][:, i] = yi # eigenmodes as 3*Psi(x)+E
        eigen_ylines[j][:, i] = np.array([Ei for XX in X_lin]) # eigenvalue E
```

In [17]:

```python
# Plot the eigenvalues & eigenvectors
fig, axes = plt.subplots(1, 4, figsize=(24, 6))
for j, ax in enumerate(axes):
    for i in range(5):
        ax.plot(X_lin, eigen_modes[j][:, i], label='E%d=%.5f' % (i+1, vals_list[j][i].real)) # 3*Psi(x)+E
        ax.plot(X_lin, Y_vi[j], 'k-', alpha=0.2) # the potential function
        ax.plot(X_lin, eigen_ylines[j][:, i], 'k--', alpha=0.6) # horizontal line y = Ei

    ax.set_xlabel('x', fontsize=16)
    ax.set_ylabel('Energy', fontsize=16)
    ax.set_xlim(-12, 12)
    ax.legend(fontsize=14)

axes[0].set_ylim(-0.5, 3.5)
axes[1].set_ylim(-0.5, 6)
axes[2].set_ylim(-0.5, 3)
axes[3].set_ylim(-0.5, 11)
axes[0].set_title('Schrodinger Equation \n$v_0(x) = x^2/10$', fontsize=16)
axes[1].set_title('Schrodinger Equation \n$v_1(x) = |x|$', fontsize=16)
axes[2].set_title('Schrodinger Equation \n$v_2(x) = 12(x/10)^4 - x^2/18 + x/8 + 13/10$', fontsize=16)
axes[3].set_title('Schrodinger Equation \n$v_3(x) = 8||x|-1|-1|$', fontsize=16)
plt.show()
```



(b)

In [18]:

```python
# Find the indices of the a=0 and b=6
ia = int(12/dx)
ib = int(18/dx)+1
x_all = np.linspace(-12, 12, n+1)
print(x_all[ia: ib].shape, x_all.shape)

# Compute probability using composite Simpson rule
for i in range(5):
    E = vals_list[2][i].real
    y = vecs_list[2][:, i].real # get the first five eigenvectors
    abs_y_sqr = np.abs(y)**2
    p = simps(abs_y_sqr[ia: ib], x_all[ia: ib])/simps(abs_y_sqr, x_all)
    print('E = %.5f, p = %.5f' % (E, p))
```

```
(481,) (1921,)
E = 0.39068, p = 0.00032
E = 1.04243, p = 0.03036
E = 1.40185, p = 0.78730
E = 1.60332, p = 0.39990
E = 1.94743, p = 0.53251
```

In [ ]: