AE588 Assignment 2

2.1

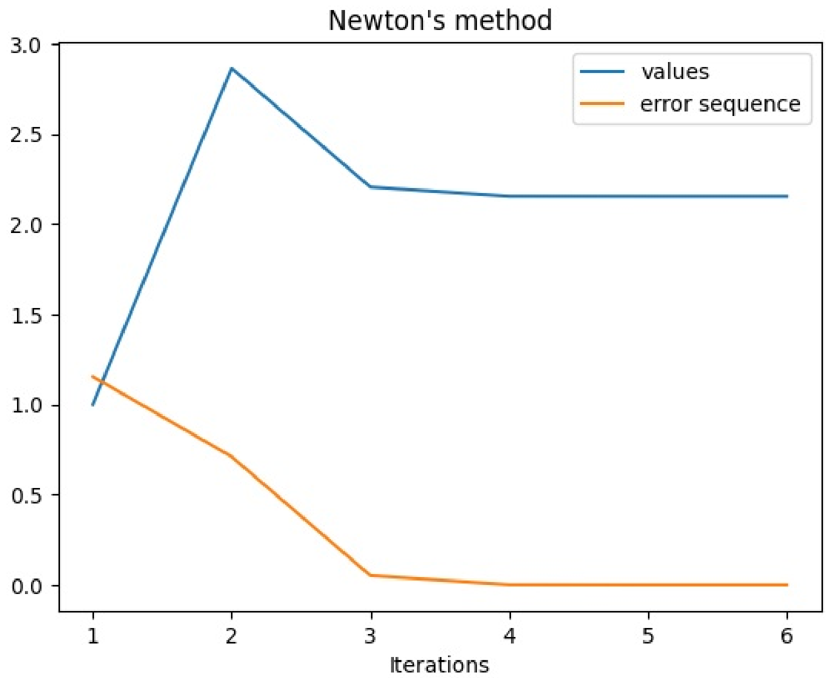
In [1]:
```
# import required modules

import numpy as np
import matplotlib.pyplot as plt

# define constants

ECC = 0.7  # e, eccentricity
MEAN_ANOM = np.pi/2.0  # M, mean_anomaly

# plotting fn for convergence


def plot_convergence(iters, values, diffs, title):
    error_const_list = []
    for value in values:
        error_const_list.append(abs(value-values[-1]))
    plt.plot(range(1, len(iters)+1), values, label="values")
    # plt.plot(range(1, len(iters)+1), [abs(diff)
    #          for diff in diffs], label="diffs")
    plt.plot(range(1, len(iters)+1), error_const_list, label="error sequence")
    plt.xlabel("Iterations")
    plt.legend(loc='best')
    plt.title(title)


# kepler's equation: E − e sin(E) = M;
def kepler_eq(eccentric_anomaly: float, eccentricity: float, mean_anomaly: float) -> float:
    return eccentric_anomaly - eccentricity*np.sin(eccentric_anomaly) - mean_anomaly

# kepler's equation in residual form: E − e sin(E) - M = 0


def kepler_eq_res(eccentric_anomaly: float, eccentricity: float, mean_anomaly: float) -> float:
    return eccentric_anomaly - eccentricity*np.sin(eccentric_anomaly) - mean_anomaly
```

2.1.a) Implement Newton's method

In [2]:
```
# newton stuff


# kepler's equation's residual differentiated wrt to E (eccentric anomaly):  1 - e cos(E)
def kepler_eq_res_de(eccentricity: float, eccentric_anomaly: float) -> float:
    return 1.0 - eccentricity*np.cos(eccentric_anomaly)

# newton's method


def newton_iter(func, func_diff, eccentric_anomaly: float, eccentricity: float, mean_anomaly: float, diff_stop:
    iter_list = []
    value_list = []
    diff_list = []
    diff = 1
    iters = 0
    while (abs(diff) > diff_stop and iters < max_iters):
        iters += 1
        eccentric_anomaly_next = eccentric_anomaly - \
            func(eccentric_anomaly, eccentricity, mean_anomaly) / \
            func_diff(eccentricity, eccentric_anomaly)
        diff = eccentric_anomaly - eccentric_anomaly_next
        iter_list.append(iters)
        value_list.append(eccentric_anomaly)
        diff_list.append(diff)
        eccentric_anomaly = eccentric_anomaly_next
    return iter_list, value_list, diff_list
```

In [3]:
```
# do newton's method
NEWTON_DIFF = 0.000000001
NEWTON_MAX_ITERS = 100

eccentric_anomaly = 1  # initial guess
iters, values, diffs = newton_iter(kepler_eq_res, kepler_eq_res_de,
                                   eccentric_anomaly, ECC, MEAN_ANOM, NEWTON_DIFF, NEWTON_MAX_ITERS)
```

```
plot_convergence(iters, values, diffs, "Newton's method")
```

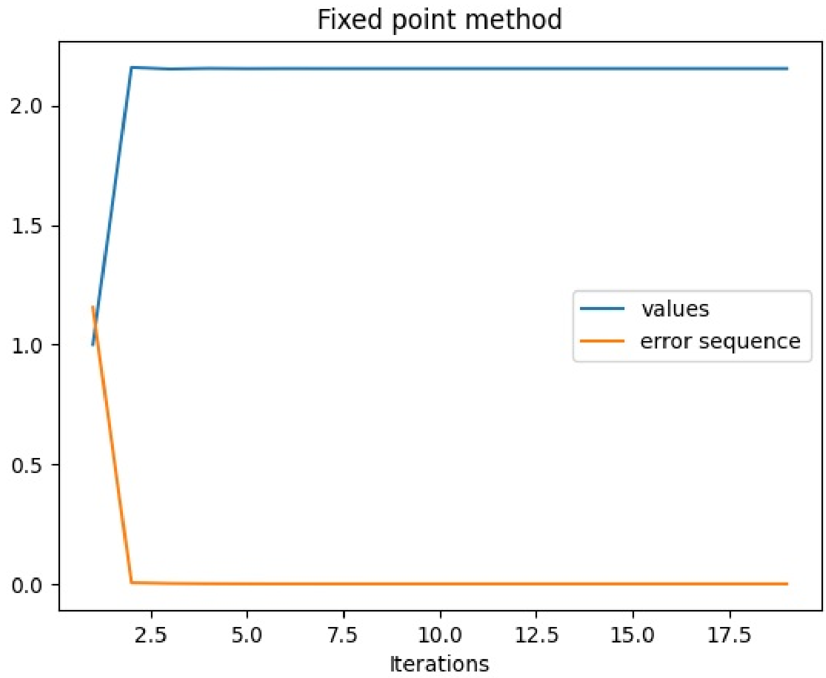


- Newton's method converges in 6 iterations
- the maximum precision here is limited by the precision of Pi as defined in np.pi

2.1.b) Fixed point implementation

In [4]:
```
# fixed point function
def fixed_pt_fn(eccentric_anomaly: float, eccentricity: float, mean_anomaly: float) -> float:
    return mean_anomaly + eccentricity*np.sin(eccentric_anomaly)

# fixed point iteration


def fixed_pt_iter(func, eccentric_anomaly: float, eccentricity: float, mean_anomaly: float, diff_stop: float, ma
    iter_list = []
    value_list = []
    diff_list = []
    diff = 1
    iters = 0
    while (abs(diff) > diff_stop and iters < max_iters):
        iters += 1
        eccentric_anomaly_next = func(
            eccentric_anomaly, eccentricity, mean_anomaly)
        diff = eccentric_anomaly - eccentric_anomaly_next
        iter_list.append(iters)
        value_list.append(eccentric_anomaly)
        diff_list.append(diff)
        eccentric_anomaly = eccentric_anomaly_next
    return iter_list, value_list, diff_list
```

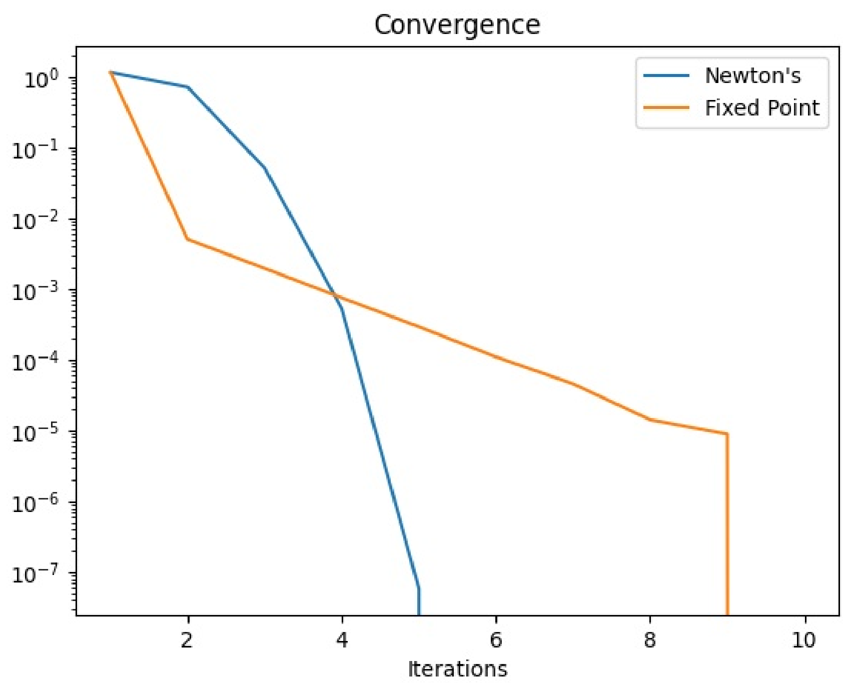
In [5]:
```
# do fixed point method

FIXED_DIFF = 0.000000001
FIXED_MAX_ITERS = 100

eccentric_anomaly = 1  # initial guess

iters, values, diffs = fixed_pt_iter(
    fixed_pt_fn, eccentric_anomaly, ECC, MEAN_ANOM, FIXED_DIFF, FIXED_MAX_ITERS)

plot_convergence(iters, values, diffs, "Fixed point method")
```



2.1.c) Compare the number of iterations and rate of convergence.

In [6]:

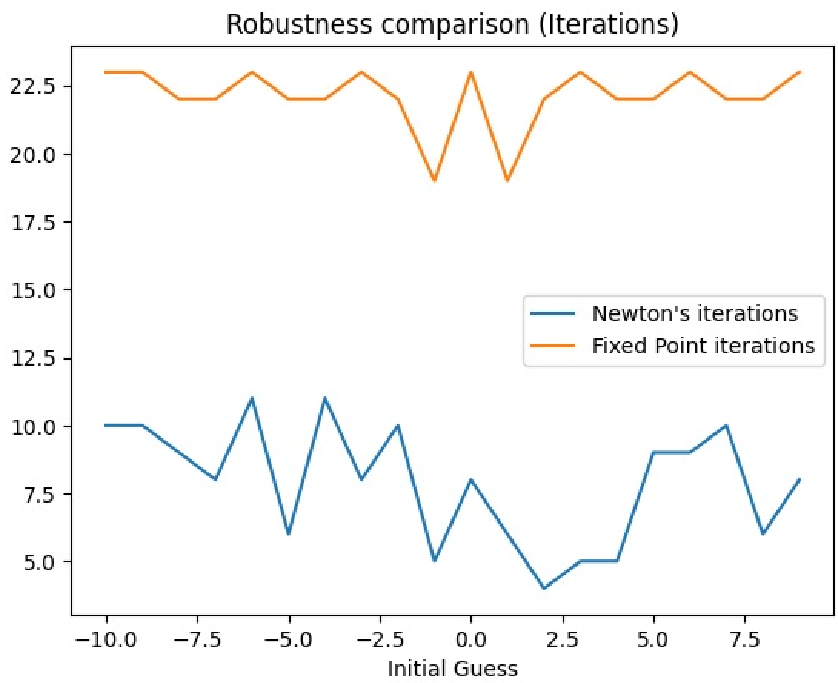
```
# comparing rate of convergence for same number of iterations
DIFF = 0.000000001
MAX_ITERS = 10

eccentric_anomaly = 1  # initial guess
iters_n, values_n, diffs_n = newton_iter(kepler_eq_res, kepler_eq_res_de,
                                         eccentric_anomaly, ECC, MEAN_ANOM, DIFF, MAX_ITERS)
eccentric_anomaly = 1  # initial guess
iters_f, values_f, diffs_f = fixed_pt_iter(
    fixed_pt_fn, eccentric_anomaly, ECC, MEAN_ANOM, DIFF, MAX_ITERS)

error_const_list_n = []
for value in values_n:
    error_const_list_n.append(abs(value-values_n[-1]))

error_const_list_f = []
for value in values_f:
    error_const_list_f.append(abs(value-values_f[-1]))

plt.plot(range(1, len(iters_n)+1), error_const_list_n, label="Newton's")
plt.plot(range(1, len(iters_f)+1), error_const_list_f, label="Fixed Point")
plt.yscale('log')
plt.xlabel("Iterations")
plt.legend(loc='best')
plt.title("Convergence")
plt.show()
```



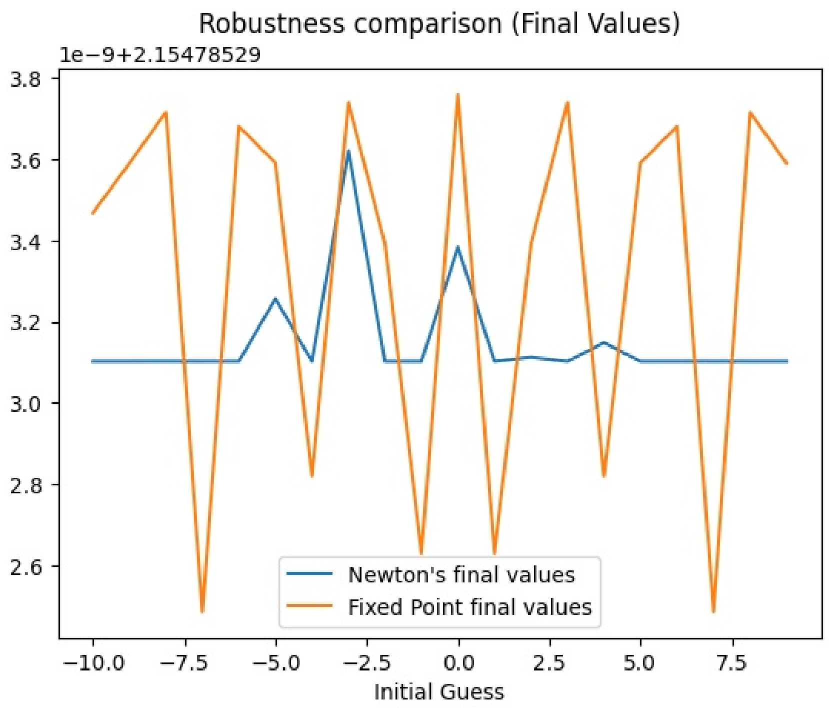
- The fixed point method converges in 19 iterations compared to 5 iterations for Newton's for the same precision of 1e-9
- the rate of convergence is a lot higher for Newton's method
  - Newton: p ~= 2
  - Fixed point: p ~= 1, y ~= 0.1

2.1.d) Evaluate the robustness of each method by trying different initial guesses for E

In [7]:
```
# robustness

ROB_RANGE = range(-10, 10, 1)
DIFF = 0.000000001
MAX_ITERS = 100

total_iters_n = []
total_vals_n = []
total_iters_f = []
total_vals_f = []
for guess in ROB_RANGE:
    iters_n, values_n, diffs_n = newton_iter(
        kepler_eq_res, kepler_eq_res_de, guess, ECC, MEAN_ANOM, DIFF, MAX_ITERS)
    total_iters_n.append(iters_n[-1])
    total_vals_n.append(values_n[-1])
    iters_f, values_f, diffs_f = fixed_pt_iter(
        fixed_pt_fn, guess, ECC, MEAN_ANOM, DIFF, MAX_ITERS)
    total_iters_f.append(iters_f[-1])
    total_vals_f.append(values_f[-1])

# comparing iterations needed for result
plt.plot(ROB_RANGE, total_iters_n, label="Newton's iterations")
plt.plot(ROB_RANGE, total_iters_f, label="Fixed Point iterations")
plt.xlabel("Initial Guess")
plt.legend(loc='best')
plt.title("Robustness comparison (Iterations)")
plt.show()
```



In [8]:
```
# comparing final values

plt.plot(ROB_RANGE, total_vals_n, label="Newton's final values")
plt.plot(ROB_RANGE, total_vals_f, label="Fixed Point final values")
# plt.yscale('log')
plt.xlabel("Initial Guess")
plt.legend(loc='best')
plt.title("Robustness comparison (Final Values)")
plt.show()
# print(total_vals_n, total_vals_f)
```

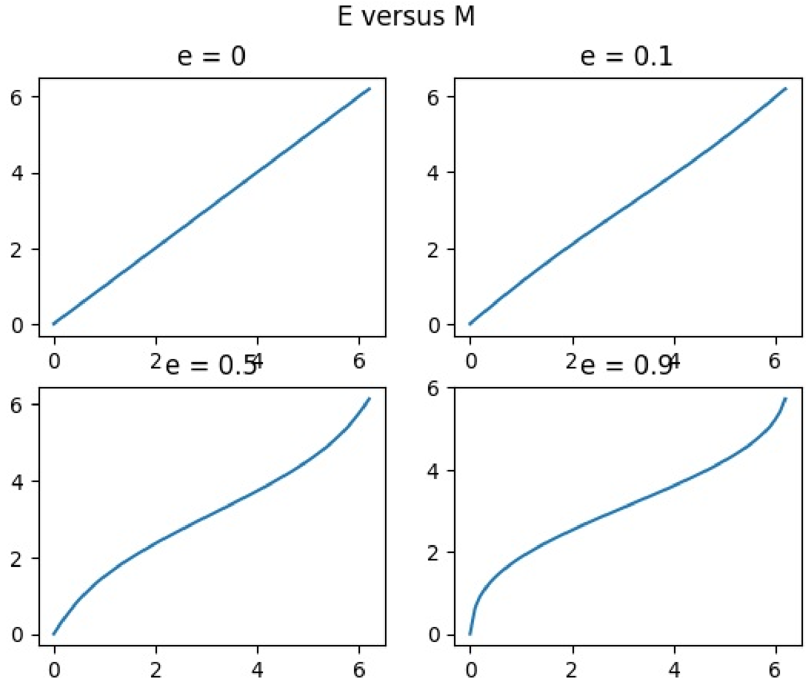


- both methods take a consistent amount of iterations to get to the final result within the same 1e-9 bounds
- Newton's method is more consistent compared to the fixed point method, but only by a small amount, in the order of 1e-9

2.1.e) Plot E versus M in the interval [0, 2π] for e = [0, 0.1, 0.5, 0.9]. Optional: interpret your results physically.

In [9]:
```
# generate data for the entire range

DIFF = 0.000000001
MAX_ITERS = 100

MEAN_ANOM_INTERVAL = np.arange(0, 2*np.pi, 0.1)
ECC_INTERVAL = [0, 0.1, 0.5, 0.9]

eccentric_anomaly = 1  # initial guess
```

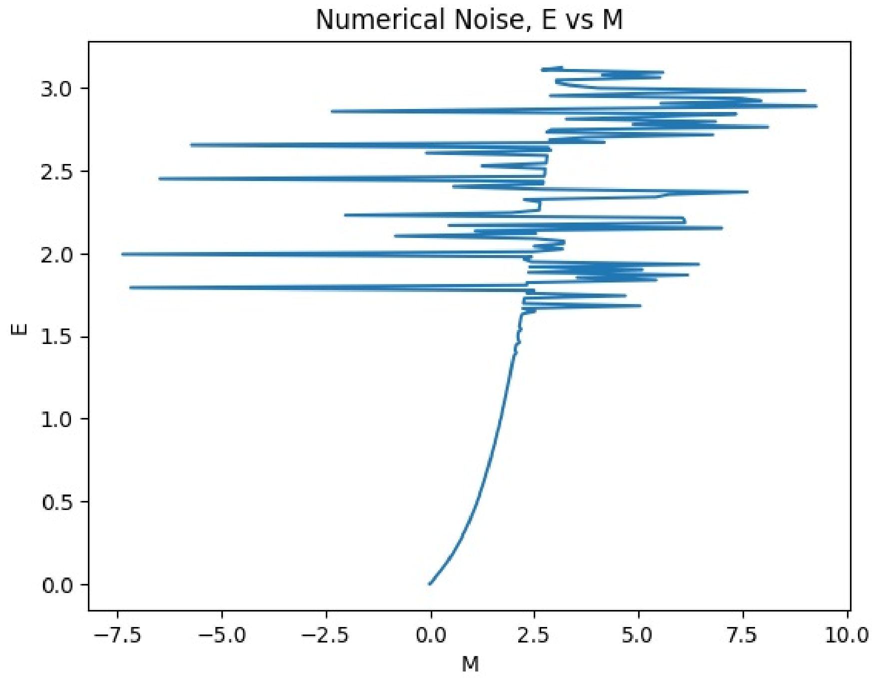
```
eccentric_anomaly_list = []

# # plot the data
fig, axs = plt.subplots(ncols=int(len(ECC_INTERVAL)/2),
                        nrows=int(len(ECC_INTERVAL)/2))
fig.suptitle("E versus M")

it = 0
for ecc in ECC_INTERVAL:
    eccentric_anomaly_list = []
    for mean_anom in MEAN_ANOM_INTERVAL:
        iters, values, diffs = newton_iter(
            kepler_eq_res, kepler_eq_res_de, eccentric_anomaly, ecc, mean_anom, NEWTON_DIFF, NEWTON_MAX_ITERS)
        eccentric_anomaly_list.append(values[-1])
    axs[int(it/2)][int(it % 2)].set_title(f"e = {ecc}")
    axs[int(it/2)][int(it % 2)].plot(MEAN_ANOM_INTERVAL, eccentric_anomaly_list)
    it += 1

plt.show()
```



2.1.f) Produce a plot showing the numerical noise by perturbing M in the neighborhood of M = π/2 with e = 0.7 using a solver convergence tolerance of |r| ≤ 0.01. Note: you might want to randomize the starting points for the solver.

In [18]:

```
DIFF = 0.01
MAX_ITERS = 5
RAND_RANGE = np.pi/2
MEAN_ANOM_INTERVAL = np.arange(
    np.pi/2-RAND_RANGE, np.pi/2+RAND_RANGE, RAND_RANGE/100)

# eccentric_anomaly = 1  # initial guess
rng = np.random.default_rng()
sample_list = []
for mean_anom in MEAN_ANOM_INTERVAL:
    iters, values, diffs = newton_iter(
        kepler_eq_res, kepler_eq_res_de, rng.random(), ECC, mean_anom, DIFF, MAX_ITERS)
    sample_list.append(values[-1])
    # print(values)


plt.plot(sample_list, MEAN_ANOM_INTERVAL)
plt.xlabel("M")
plt.ylabel("E")
plt.title("Numerical Noise, E vs M")
plt.show()

# print(sample_list)
# print(sample_list, MEAN_ANOM_SAMPLE)
```


Numerical Noise, E vs M
E
M
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

2.2

Analytically, we get partial derivatives as

- wrt x_1: $4x_1^3 + 9x_1^2 − 6x_2$
- wrt x_2: $6x_2 − 6x_1 −2$

Solving

- $4x_1^3 + 9x_1^2 − 6x_2 = 0$
- $6x_2 − 6x_1 −2 = 0$

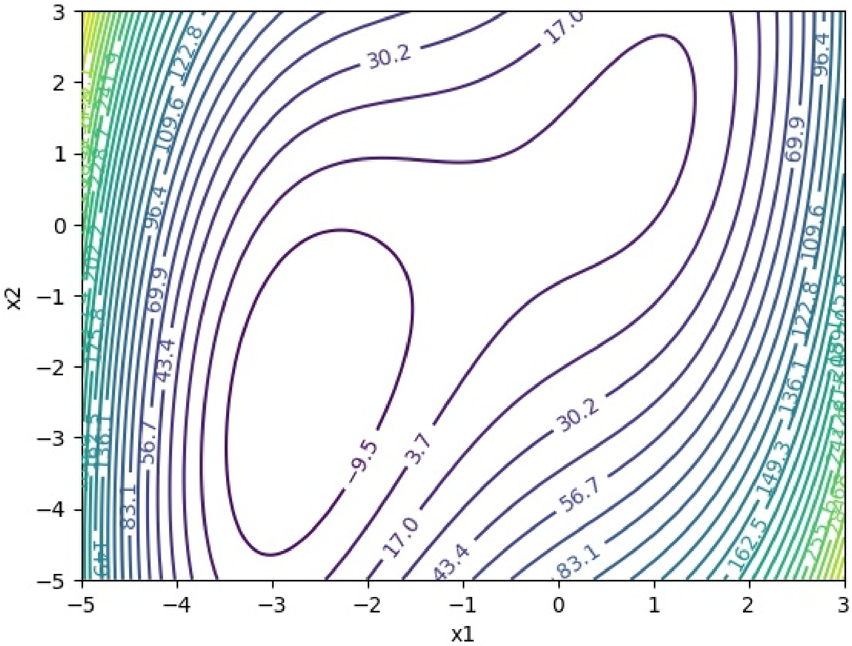
The roots are (x_1, x_2)

To classify them, we calculate the Hessians at the points and use their eigenvalues

- $(- \sqrt3 - 1), (-\sqrt3 - 2/3)$: Global minimum (-2.73205, -2.39871)
- $(-1 + \sqrt3) , (-2/3 + \sqrt3)$: Local minimum (0.73205, 1.06538)
- $-1/4, 1/12$: Saddle Point (-0.25, 0.08333)

In [8]:

```python
# import required modules
from typing import Callable

import numpy.typing as npt

import numpy as np
import matplotlib.pyplot as plt

# function from the problem


def function2(x: npt.ArrayLike) -> float:
    return pow(x[0], 4) + 3*pow(x[0], 3) + 3*pow(x[1], 2) - 6*x[0]*x[1] - 2*x[1]


def prep_data(function: Callable[[float, float], float], range_x1: tuple[float, float, float], range_x2: tuple[
    x1 = np.linspace(range_x1[0], range_x1[1], range_x1[2])
    x2 = np.linspace(range_x2[0], range_x2[1], range_x2[2])
    x1, x2 = np.meshgrid(x1, x2)
    fx = function([x1, x2])
    return x1, x2, fx


def plot_data(x1: npt.ArrayLike, x2: npt.ArrayLike, fx: npt.ArrayLike) -> None:
    _, ax = plt.subplots()
    levels = np.linspace(np.min(fx), np.max(fx), 30)
    CS = ax.contour(x1, x2, fx, levels=levels)
    ax.clabel(CS, inline=True, fontsize=10)
    ax.set_xlabel("x1")
    ax.set_ylabel("x2")
    plt.show()


RANGE_X1 = (-5, 3, 1000)
RANGE_X2 = (-5, 3, 1000)

# plot the function contour
x1, x2, fx = prep_data(function2, RANGE_X1, RANGE_X2)
plot_data(x1, x2, fx)
```



After plotting the results, all the critical points match what is seen on the contour plot

AE588 Assignment 2

2.3

In [26]:
```python
import numpy as np
import numpy.typing as npt
import matplotlib.pyplot as plt

# example 4.8 functions and directions
DIR_4_8 = np.array([4, 0.75])


def dir_4_8(x: npt.ArrayLike) -> float:
    return DIR_4_8


def grad_4_8(x: npt.ArrayLike):
    return [0.6*x[0]**5 - 6*x[0]**3 + 10*x[0] + 0.5*x[1], 0.4*x[1]**3 + 6*x[1] - 9 + 0.5*x[0]]


def fn_4_8(x: npt.ArrayLike) -> float:
    return 0.1*(x[0]**6) - 1.5*(x[0]**4) + 5*(x[0]**2) + 0.1 * (x[1]**4) + 3*(x[1]**2) - 9*x[1] + 0.5*x[0] * (x
```

In [27]:
```python
# interpolation
def interpolation_min(func, func_grad, a_low, a_high):
    return (2 * a_low * (func(a_high)-func(a_low)) + func_grad(a_low) * (a_low**2 - a_high**2)) / (2 * (func(a_l

# pinpointing


def pinpoint(func, func_grad, dir, a_low, a_high, phi_0, phi_low, phi_high, phi_0_grad, phi_low_grad, phi_high_g
    k = 0
    while True:
        a_p = interpolation_min(func, func_grad, a_low, a_high)
        print(
            f"after interpolation: a_low: {a_low}, a_high: {a_high}, a_p: {a_p}")
        phi_p = func(a_p)
        phi_p_grad = func_grad(a_p)
        print(f"k: {k}, a_low: {a_low}, a_high: {a_high}, phi_0: {phi_0}, phi_low: {phi_low}, phi_high: {phi_hig
        if phi_p > phi_0 + suff_dec*np.dot(a_p*phi_0_grad, dir) or phi_p > phi_low:
            a_high = a_p
            phi_high = phi_p
            # phi_high_grad = phi_p_grad
        else:
            if abs(np.dot(phi_p_grad, dir)) <= -suff_cur*np.dot(phi_0_grad, dir):
                a = a_p
                return a_p
            elif np.dot(phi_p_grad * (a_high - a_low), dir) >= 0:
                a_high = a_low
            a_low = a_p
        k = k+1


# bracketing
# suff_dec = u1
# suff_cur = u2
# step_inc = σ or sigma
def bracket(func, func_grad, dir: npt.ArrayLike, guess: npt.ArrayLike, initial_step: float, suff_dec: float, su
    step = initial_step
    brkt_start = guess  # a1
    brkt_end = guess + initial_step  # a2
    func_0 = func(guess)  # phi 0
    func_grad_0 = func_grad(guess)  # phi 0 prime
    func_start = func_0  # phi 1
    func_start_grad = func_grad_0  # phi 1 prime
    # func_end = guess_diff # phi 2
    first = True
    while (True):
        print(f"step: {step}, brkt_start: {brkt_start}, brkt_end: {brkt_end}")
        func_end = func(brkt_end)  # phi 2
        # check if sufficient decrease conditions already met or the end is higher than start
        if (func_end > (func_0 + suff_dec * step * np.dot(dir, func_start_grad))) or (not first and func_end > 
            step = pinpoint(func, func_grad, dir, brkt_start,
                            brkt_end, func_0, func_end, func(brkt_start), func_grad_0, func_grad(brkt_start), fu
            return step
        func_end_grad = func_grad(brkt_end)  # phi 2 prime
        # check if sufficient curvature conditions met
        if abs(func_end_grad) <= -suff_cur*func_grad_0:
```

```
            step = brkt_end
            return step
        # check if end gradient is positive, suggesting the min is within the bracket
        elif func_end_grad >= 0:
            # step = pinpoint(...)
            step = pinpoint(func, func_grad, dir, brkt_end,
                            brkt_start, func_0, func(brkt_start), func_end, func_grad_0, func_grad(brkt_end), fu
            return step
        else:
            brkt_start = brkt_end
            brkt_end = brkt_start*step_inc
        first = False


# backtracking line search
def bktrk_lin_search(func, grad: npt.ArrayLike, dir: npt.ArrayLike, guess: npt.ArrayLike, initial_step: float, s
    step = initial_step
    steps = [initial_step]
    fn_list = [func(guess + (step * dir))]
    # print(f"step: {step}, fx: {func(guess + (step * dir))}")
    # step+dir = step in a particular dir
    # dot prod to know how much the fn is expected to decrease in a particular dir
    while func(guess + (step * dir)) > (func(guess) + suff_dec * step * np.dot(grad, dir)):
        step = bktrk * step
        # print(f"step: {step}, fx: {func(guess + (step * dir))}")
        steps.append(step)
        fn_list.append(func(guess + (step * dir)))
    return steps, fn_list

# gradient optimization


def grad_opt(func, func_grad, func_dir, guess: npt.ArrayLike, tolerance: float, initial_step: float, suff_dec: 
    it = 0
    step = initial_step
    grad = func_grad(guess)
    val_list = [func(guess)]
    # gradient should tend towards 0, but wont here because we will never change to the right direction
    while np.linalg.norm(func_grad(guess), np.inf) > tolerance:
        print(
            f"it: {it}, step: {step}, guess: {guess}, fx: {func(guess)}, grad: {func_grad(guess)}")
        dir = func_dir(guess)
        steps, fn_list = bktrk_lin_search(func, func_grad(
            guess), dir, guess, step, suff_dec, bktrk)
        step = steps[-1]
        guess = guess + step * dir
        it += 1
        val_list.append(func(guess))
        # print(f"backtrack: {fn_list}")
        # print(
        #     f"it: {it}, step: {step}, guess: {guess}, fx: {func(guess)}, grad: {func_grad(guess)}")

    # plot optimization fn vs iterations
    # plt.plot(val_list)
    # plt.xlabel("iteration")
    # plt.ylabel("f")
    # plt.title("Optimization: Function vs iterations")
    # plt.show()
    return guess, func(guess)
```

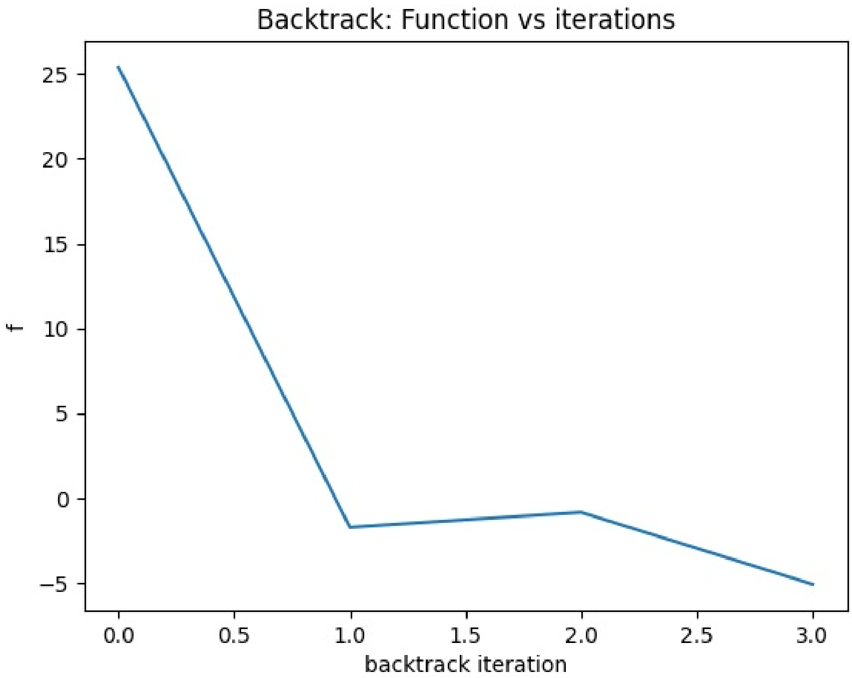
2.3.a) Graphs for Example 4.8

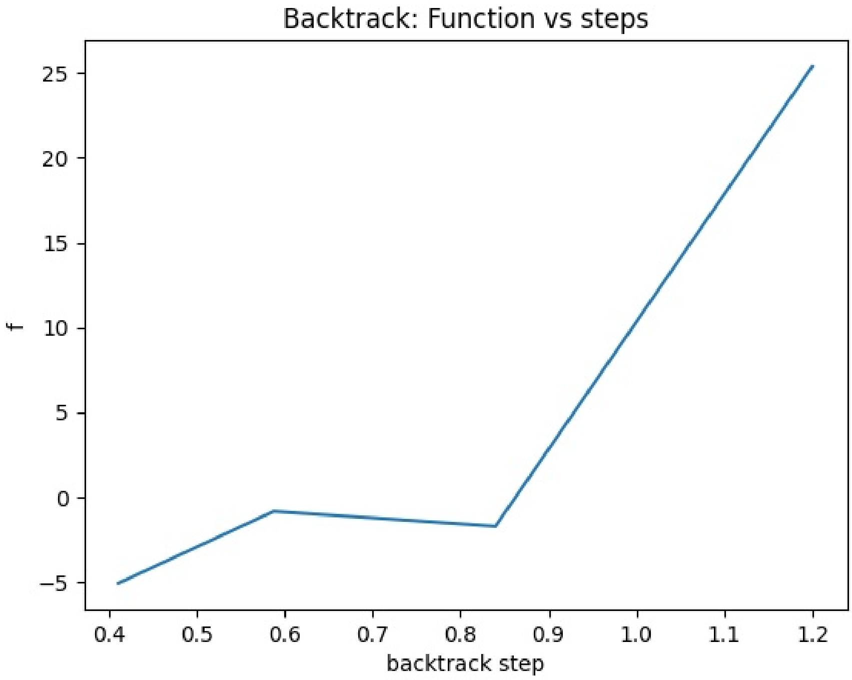
In [30]:

```
# run optimization on 4.8 with defaults
SUFF_DEC = 1e-4  # u
BKTRK = 0.7  # p
TOLERANCE = 1e-6  # t
GUESS_4_8 = np.array([-1.25, 1.25])
INITIAL_STEP = 1.2
# grad opt wont work because the direction function isnt implemented
# x, fx = grad_opt(fn_4_8, grad_4_8, dir_4_8, GUESS_4_8, TOLERANCE,
#                  INITAL_STEP, SUFF_DEC, BKTRK)
steps, fx = bktrk_lin_search(fn_4_8, grad_4_8(GUESS_4_8), dir_4_8(
    GUESS_4_8), GUESS_4_8, INITIAL_STEP, SUFF_DEC, BKTRK)

print(
    f"final guess: {GUESS_4_8 + steps[-1]}, grad: {grad_4_8(GUESS_4_8 + steps[-1])}, dir grad: {np.dot(grad_4_8


# plot backtrack vs iterations
plt.plot(fx)
plt.xlabel("backtrack iteration")
plt.ylabel("f")
plt.title(f"Backtrack: Function vs iterations")
```

```
plt.show()


# plot backtrack vs step
plt.plot(steps, fx)
plt.xlabel("backtrack step")
plt.ylabel("f")
plt.title(f"Backtrack: Function vs steps")
plt.show()
```

```
final guess: [-0.8384  1.6616], grad: [-4.265805066897123, 2.3854142531584], dir grad: -15.274159577719692
```





2.3.a) Graphs for example 4.9

In [29]:

```
# run bracketing on 4.9 with defaults
SUFF_DEC = 1e-4  # u1
SUFF_CUR = 0.9  # u2
STEP_INCR = 2
TOLERANCE = 1e-6  # t
GUESS_4_8 = np.array([-1.25, 1.25])
INITAL_STEP = 1.2
x, fx = bracket(fn_4_8, grad_4_8, dir_4_8(GUESS_4_8), GUESS_4_8,
                INITAL_STEP, SUFF_DEC, SUFF_CUR, STEP_INCR)
print(x, fx)
```

```
step: 1.2, brkt_start: [-1.25  1.25], brkt_end: [-0.05  2.45]
after interpolation: a_low: [-1.25  1.25], a_high: [-0.05  2.45], a_p: [-1.26822172  1.93450807]
k: 0, a_low: [-1.25  1.25], a_high: [-0.05  2.45], phi_0: -2.5677490234375, phi_low: -0.4882587484374941, phi_hi
gh: -2.5677490234375, phi_0_grad: [-1.9873046875, -1.34375], phi_low_grad: [-1.9873046875, -1.34375], phi_high_g
rad: [0.7257498124999997, 11.557450000000001], suff_dec: 0.0001, suff_cur: 0.9
after interpolation: a_low: [-1.25  1.25], a_high: [-1.26822172  1.93450807], a_p: [-1.22945165  1.80762476]
k: 1, a_low: [-1.25  1.25], a_high: [-1.26822172  1.93450807], phi_0: -2.5677490234375, phi_low: -0.488258748437
4941, phi_high: -1.4321575131679616, phi_0_grad: [-1.9873046875, -1.34375], phi_low_grad: [-1.9873046875, -1.343
75], phi_high_grad: [0.7257498124999997, 11.557450000000001], suff_dec: 0.0001, suff_cur: 0.9
after interpolation: a_low: [-1.25  1.25], a_high: [-1.22945165  1.80762476], a_p: [-1.29527395  1.99527651]
k: 2, a_low: [-1.25  1.25], a_high: [-1.22945165  1.80762476], phi_0: -2.5677490234375, phi_low: -0.488258748437
4941, phi_high: -2.033701370953655, phi_0_grad: [-1.9873046875, -1.34375], phi_low_grad: [-1.9873046875, -1.3437
5], phi_high_grad: [0.7257498124999997, 11.557450000000001], suff_dec: 0.0001, suff_cur: 0.9
after interpolation: a_low: [-1.25  1.25], a_high: [-1.29527395  1.99527651], a_p: [-1.20953795  1.75311408]
k: 3, a_low: [-1.25  1.25], a_high: [-1.29527395  1.99527651], phi_0: -2.5677490234375, phi_low: -0.488258748437
4941, phi_high: -1.0826495627275987, phi_0_grad: [-1.9873046875, -1.34375], phi_low_grad: [-1.9873046875, -1.343
75], phi_high_grad: [0.7257498124999997, 11.557450000000001], suff_dec: 0.0001, suff_cur: 0.9
after interpolation: a_low: [-1.25  1.25], a_high: [-1.20953795  1.75311408], a_p: [-1.38812529  2.16184795]
k: 4, a_low: [-1.25  1.25], a_high: [-1.20953795  1.75311408], phi_0: -2.5677490234375, phi_low: -0.488258748437
4941, phi_high: -2.255884529183462, phi_0_grad: [-1.9873046875, -1.34375], phi_low_grad: [-1.9873046875, -1.3437
5], phi_high_grad: [0.7257498124999997, 11.557450000000001], suff_dec: 0.0001, suff_cur: 0.9
after interpolation: a_low: [-1.25  1.25], a_high: [-1.38812529  2.16184795], a_p: [-1.17252024  1.66275123]
k: 5, a_low: [-1.25  1.25], a_high: [-1.38812529  2.16184795], phi_0: -2.5677490234375, phi_low: -0.488258748437
4941, phi_high: 0.02844030378756801, phi_0_grad: [-1.9873046875, -1.34375], phi_low_grad: [-1.9873046875, -1.343
75], phi_high_grad: [0.7257498124999997, 11.557450000000001], suff_dec: 0.0001, suff_cur: 0.9
after interpolation: a_low: [-1.17252024  1.66275123], a_high: [-1.25  1.25], a_p: [-2.63691248  2.87462715]
k: 6, a_low: [-1.17252024  1.66275123], a_high: [-1.25  1.25], phi_0: -2.5677490234375, phi_low: -0.488258748437
4941, phi_high: 0.02844030378756801, phi_0_grad: [-1.9873046875, -1.34375], phi_low_grad: [-1.9873046875, -1.343
75], phi_high_grad: [0.7257498124999997, 11.557450000000001], suff_dec: 0.0001, suff_cur: 0.9
after interpolation: a_low: [-1.17252024  1.66275123], a_high: [-2.63691248  2.87462715], a_p: [-4.53114118  5.7
5316318]
k: 7, a_low: [-1.17252024  1.66275123], a_high: [-2.63691248  2.87462715], phi_0: -2.5677490234375, phi_low: -0.
4882587484374941, phi_high: -2.1807601046210796, phi_0_grad: [-1.9873046875, -1.34375], phi_low_grad: [-1.987304
6875, -1.34375], phi_high_grad: [0.7257498124999997, 11.557450000000001], suff_dec: 0.0001, suff_cur: 0.9
after interpolation: a_low: [-1.17252024  1.66275123], a_high: [-4.53114118  5.75316318], a_p: [-1.13191436  1.6
0786626]
k: 8, a_low: [-1.17252024  1.66275123], a_high: [-4.53114118  5.75316318], phi_0: -2.5677490234375, phi_low: -0.
4882587484374941, phi_high: 479.8532132266785, phi_0_grad: [-1.9873046875, -1.34375], phi_low_grad: [-1.98730468
75, -1.34375], phi_high_grad: [0.7257498124999997, 11.557450000000001], suff_dec: 0.0001, suff_cur: 0.9
after interpolation: a_low: [-1.13191436  1.60786626], a_high: [-1.17252024  1.66275123], a_p: [-0.69864306  1.1
4658957]
k: 9, a_low: [-1.13191436  1.60786626], a_high: [-1.17252024  1.66275123], phi_0: -2.5677490234375, phi_low: -0.
4882587484374941, phi_high: 479.8532132266785, phi_0_grad: [-1.9873046875, -1.34375], phi_low_grad: [-1.98730468
75, -1.34375], phi_high_grad: [0.7257498124999997, 11.557450000000001], suff_dec: 0.0001, suff_cur: 0.9
after interpolation: a_low: [-0.69864306  1.14658957], a_high: [-1.13191436  1.60786626], a_p: [0.78571184 1.470
66435]
k: 10, a_low: [-0.69864306  1.14658957], a_high: [-1.13191436  1.60786626], phi_0: -2.5677490234375, phi_low: -0
.4882587484374941, phi_high: 479.8532132266785, phi_0_grad: [-1.9873046875, -1.34375], phi_low_grad: [-1.9873046
875, -1.34375], phi_high_grad: [0.7257498124999997, 11.557450000000001], suff_dec: 0.0001, suff_cur: 0.9
after interpolation: a_low: [0.78571184 1.47066435], a_high: [-1.13191436  1.60786626], a_p: [-0.27793992  0.835
34289]
k: 11, a_low: [0.78571184 1.47066435], a_high: [-1.13191436  1.60786626], phi_0: -2.5677490234375, phi_low: -0.4
882587484374941, phi_high: 479.8532132266785, phi_0_grad: [-1.9873046875, -1.34375], phi_low_grad: [-1.987304687
5, -1.34375], phi_high_grad: [0.7257498124999997, 11.557450000000001], suff_dec: 0.0001, suff_cur: 0.9
after interpolation: a_low: [-0.27793992  0.83534289], a_high: [0.78571184 1.47066435], a_p: [0.01938839 1.40587
562]
k: 12, a_low: [-0.27793992  0.83534289], a_high: [0.78571184 1.47066435], phi_0: -2.5677490234375, phi_low: -0.4
882587484374941, phi_high: 479.8532132266785, phi_0_grad: [-1.9873046875, -1.34375], phi_low_grad: [-1.987304687
5, -1.34375], phi_high_grad: [0.7257498124999997, 11.557450000000001], suff_dec: 0.0001, suff_cur: 0.9
0.019388386057349535 1.4058756160704389
```