Lab 2

In [45]:
```
# Imports
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from matplotlib import colormaps
```

Task 1: Gradient Decent

1. What's the gradient of our function f? Define a gradient function g and plot it.

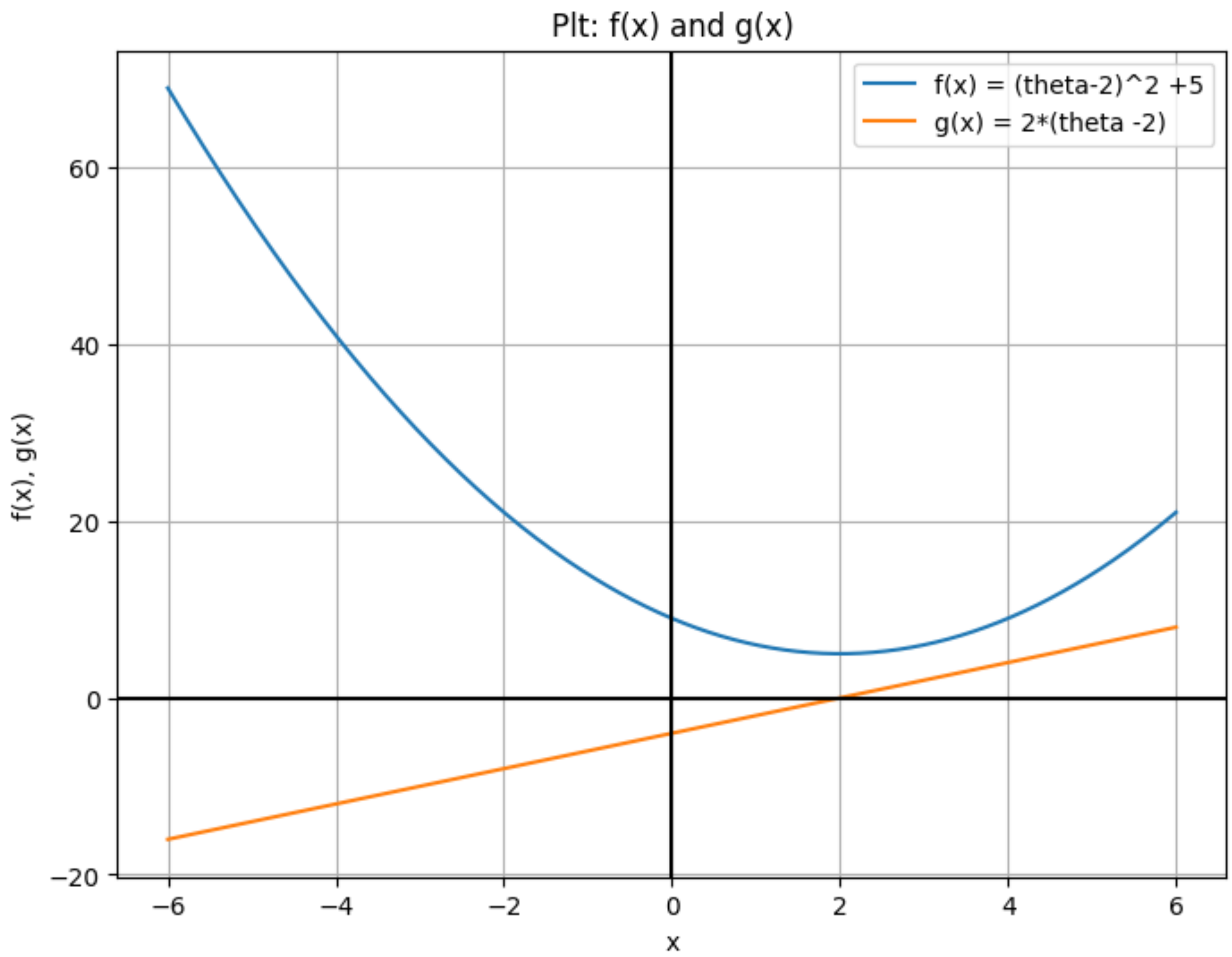
$$f(\theta) = (\theta - 2)^2 + 5$$
$$f'(\theta) = g(\theta) = 2 * (\theta - 2)$$

In [46]:
```
def f(theta):
    return (theta-2)**2 + 5

def g(theta):
    return 2*(theta -2)

results = [f(x) for x in range(1,10,1)]

results
```

Out[46]: [6, 5, 6, 9, 14, 21, 30, 41, 54]

In [47]:
```
x = np.linspace(-6,6,400)
y_f = f(x)
y_g = g(x)

plt.figure(figsize=(8, 6))
plt.plot(x, y_f, label='f(x) = (theta-2)^2 +5')
plt.plot(x, y_g, label='g(x) = 2*(theta -2)')
plt.title('Plt: f(x) and g(x)')
plt.xlabel('x')
plt.ylabel('f(x), g(x)')

ax = plt.gca()

ax.axhline(0, color='black', linewidth=1.5)
ax.axvline(0, color='black', linewidth=1.5)

plt.grid(True)
plt.legend()

plt.show
```

Out[47]: <function matplotlib.pyplot.show(close=None, block=None)>



2. Assume a constant learning rate of λ = .8. Write down the general update step for gradient descent.

General update step:

$$\theta_{t+1} = \theta_t - \lambda f'(\theta_t)$$

In [48]:
```
# Learning rate
learning_rate = 0.8

# Updating theta step by step with the learning rate
update_step = lambda theta_old: theta_old - learning_rate * g(theta_old)

# Initial value for theta
theta = 5.0

# List to store the theta and gradient values
theta_values = [theta]
gradient_values = [g(theta)]
f_values = [f(theta)]

# Changing the value of theta iteratively
step = 0
for _ in range(1,10,1):
    step += 1
    theta = update_step(theta)
    theta_values.append(theta)
    gradient_values.append(g(theta))
    f_values.append(f(theta))
    print(f"Theta at iteration {step}: " + str(theta))
```

```
Theta at iteration 1: 0.1999999999999993
Theta at iteration 2: 3.0800000000000005
Theta at iteration 3: 1.3519999999999996
Theta at iteration 4: 2.3888000000000003
Theta at iteration 5: 1.7667199999999998
Theta at iteration 6: 2.139968
Theta at iteration 7: 1.9160192
Theta at iteration 8: 2.05038848
Theta at iteration 9: 1.9697669119999999
```

3. Implement gradient descent for minimizing f making use of your defined gradient function g. Compute 20 iterations to find the θ that minimizes $f(\theta)$. Plot the sequence of θ against the iteration t. Start with θ_0 = 5.

In [18]:
```
# Learning rate
learning_rate = 0.8

# Stepwise approach in which theta is updated
update_step = lambda theta_old: theta_old - learning_rate * g(theta_old)
```

```
# Initial value
theta = 5.0

# List for storing theta and the gradient
theta_values = [theta]
gradient_values = [g(theta)]
f_values = [f(theta)]

for _ in range(0,20,1):
    theta = update_step(theta)
    theta_values.append(theta)
    gradient_values.append(g(theta))
    f_values.append(f(theta))

"""
# Convergence criterion: gradient close to zero
while abs(g(theta)) > 0.001:  # Converge criteria: Gradient close/equal to zero
    theta = update_step(theta)
    theta_values.append(theta)
    gradient_values.append(g(theta))
    f_values.append(f(theta))
"""
# Print results
print("Final Theta",theta_values[-1:])
print("\nConverged Theta Values:", theta_values)
print("\nConverged Gradient Values:", gradient_values)
print("\nValue of the Function:", f_values)
```

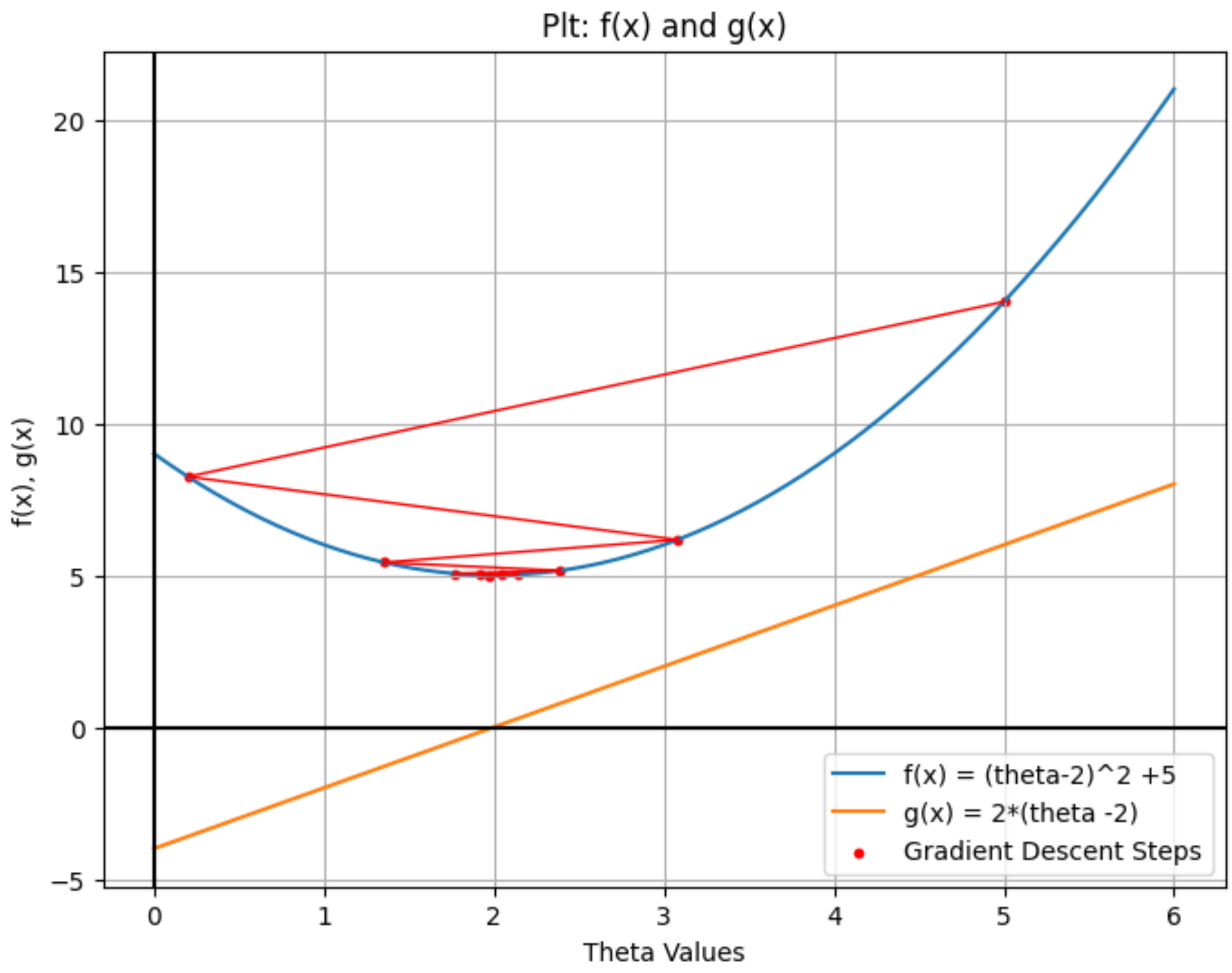
```
Final Theta [2.000109684753202]

Converged Theta Values: [5.0, 0.1999999999999993, 3.0800000000000005, 1.3519999999999996, 2.3888000000000003, 1.7667
199999999998, 2.139968, 1.9160192, 2.05038848, 1.9697669119999999, 2.0181398528, 1.9891160883199999, 2.0065303470080
003, 1.9960817917951998, 2.00235092492288, 1.998589445046272, 2.0008463329722366, 1.999492200216658, 2.0003046798700
05, 1.999817192077997, 2.000109684753202]

Converged Gradient Values: [6.0, -3.6000000000000014, 2.160000000000001, -1.2960000000000007, 0.7776000000000005,
-0.4665600000000003, 0.2799360000000002, -0.16796159999999993, 0.10077696000000014, -0.060466176000000026, 0.03627970
5600000156, -0.021767823360000027, 0.013060694016000518, -0.00783641640960031, 0.004701849845759831, -0.0028211099074
559876, 0.0016926659444731484, -0.0010155995666840667, 0.0006093597400100847, -0.00036561584400596203, 0.00021936950
640366604]

Value of the Function: [14.0, 8.240000000000002, 6.166400000000001, 5.419904000000001, 5.15116544, 5.0544195584, 5.0
19591041024, 5.00705277476864, 5.002538998916711, 5.000914039610016, 5.0003290542596055, 5.000118459533458, 5.000042
645432045, 5.000015352355536, 5.000005526847993, 5.000001989665278, 5.0000007162795, 5.00000025786062, 5.00000009282
9823, 5.000000033418736, 5.000000012030745]
```

$$min_{\theta} f(\theta) \approx 2.000109684753202$$

In [49]:

```
x = np.linspace(0,6,400)
y_f = f(x)
y_g = g(x)

plt.figure(figsize=(8, 6))
plt.plot(x, y_f, label='f(x) = (theta-2)^2 +5')
plt.plot(x, y_g, label='g(x) = 2*(theta -2)')
plt.scatter(theta_values, f_values, color='red', s=10, label='Gradient Descent Steps')
plt.title('Plt: f(x) and g(x)')
plt.xlabel('Theta Values')
plt.ylabel('f(x), g(x)')

# Draw lines connecting the red points
for i in range(1, len(theta_values)):
    plt.plot([theta_values[i-1], theta_values[i]], [f_values[i-1], f_values[i]], color='red', linewidth=1)

ax = plt.gca()
ax.axhline(0, color='black', linewidth=1.5)
ax.axvline(0, color='black', linewidth=1.5)

plt.grid(True)
plt.legend()

plt.show
```

Out[49]: <function matplotlib.pyplot.show(close=None, block=None)>



4. Replace the analytical gradient by a two-sided numerical approximation. This is often necessary in practice when the analytical gradient is hard to compute. Use a two-sided approximation such that $g(\theta) = f(\theta + h) - f(\theta - h)$ Repeat part 3 using the numerical gradient

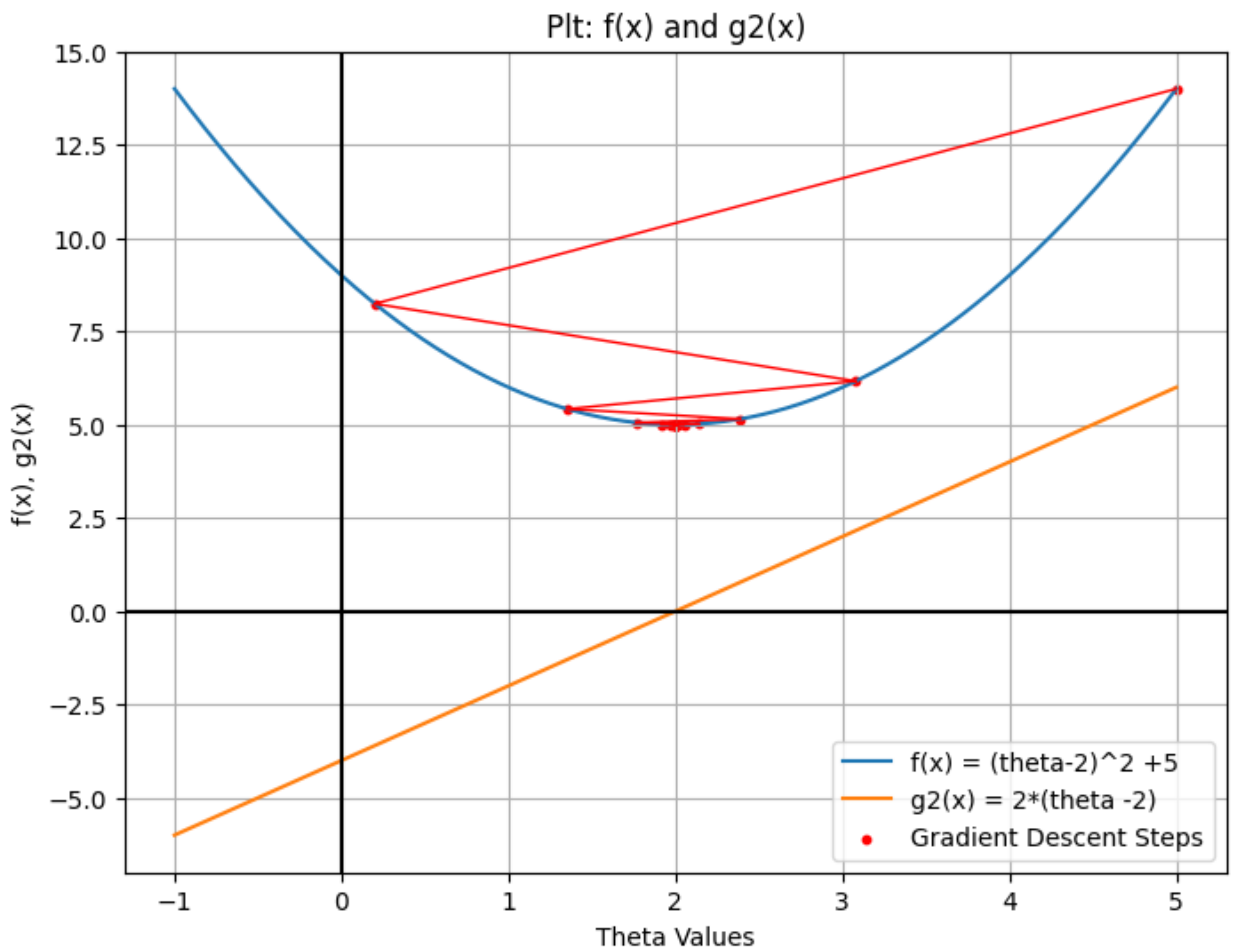
In [50]:
```
def g2(theta):
    h = 0.01
    return (f(theta+h)-f(theta-h))/(2*h)
```

In [51]:
```
# Learning rate
learning_rate = 0.8

# Stepwise approach in which theta is updated
update_step = lambda theta_old: theta_old - learning_rate * g2(theta_old)

# Initial value
theta = 5.0

# List for storing theta and the gradient
theta_values = [theta]
gradient_values = [g2(theta)]
f_values = [f(theta)]

# Convergence criterion: gradient close to zero
while abs(g2(theta)) > 0.001:
    theta = update_step(theta)
    theta_values.append(theta)
    gradient_values.append(g2(theta))
    f_values.append(f(theta))

# Print results
print("Final Theta",theta_values[-1:])
print("\nConverged Theta Values:", theta_values)
print("\nConverged Gradient Values:", gradient_values)
print("\nValue of the Function:", f_values)
```

```
Final Theta [2.000304679869984]

Converged Theta Values: [5.0, 0.20000000000010232, 3.0799999999999272, 1.3520000000001176, 2.3887999999999465, 1.766
720000000035, 2.139968000000003, 1.916019200000001, 2.050388479999974, 1.9697769120000327, 2.018139852799976, 1.9891
160883200243, 2.0065303470079954, 1.9960817917952056, 2.0023509249228866, 1.9985894450462638, 2.0008463329722304, 1.
999492200216686, 2.000304679869984]

Converged Gradient Values: [5.999999999999872, -3.599999999999781, 2.159999999999762, -1.2959999999997862, 0.7775999
999998895, -0.46655999999996034, 0.27993600000000285, -0.16796159999996618, 0.10077695999992642, -0.0604661759999292
06, 0.03627970559993976, -0.021767823359963856, 0.013060694015987195, -0.007836416409601199, 0.004701849845778483,
-0.002821109907458208, 0.0016926659444305159, -0.0010155995666227824, 0.0006093597399559059]

Value of the Function: [14.0, 8.239999999999633, 6.166399999999843, 5.419903999999848, 5.151165439999958, 5.05441955
8399983, 5.019591041024001, 5.00705277476864, 5.002538998916708, 5.000914039610014, 5.000329054259605, 5.00011845953
3457, 5.000042645432045, 5.000015352355536, 5.0000055268479935, 5.000001989665278, 5.0000007162795, 5.00000025786062
, 5.000000092829823]
```

In [52]:
```
x = np.linspace(-1,5,400)
y_f = f(x)
y_g = g2(x)

plt.figure(figsize=(8, 6))
plt.plot(x, y_f, label='f(x) = (theta-2)^2 +5')
plt.plot(x, y_g, label='g2(x) = 2*(theta -2)')
plt.scatter(theta_values, f_values, color='red', s=10, label='Gradient Descent Steps')
plt.title('Plt: f(x) and g2(x)')
plt.xlabel('Theta Values')
plt.ylabel('f(x), g2(x)')

# Draw lines connecting the red points
for i in range(1, len(theta_values)):
    plt.plot([theta_values[i-1], theta_values[i]], [f_values[i-1], f_values[i]], color='red', linewidth=1)

ax = plt.gca()
ax.axhline(0, color='black', linewidth=1.5)
ax.axvline(0, color='black', linewidth=1.5)

plt.grid(True)
plt.legend()

plt.show
```

Out[52]: `<function matplotlib.pyplot.show(close=None, block=None)>`



Task 2: Ordinary Least Squares

In [53]:
```
# Loading the data
data = pd.read_csv("Lab2_Optimization.csv", delimiter=";")
data
```

Out [53]:

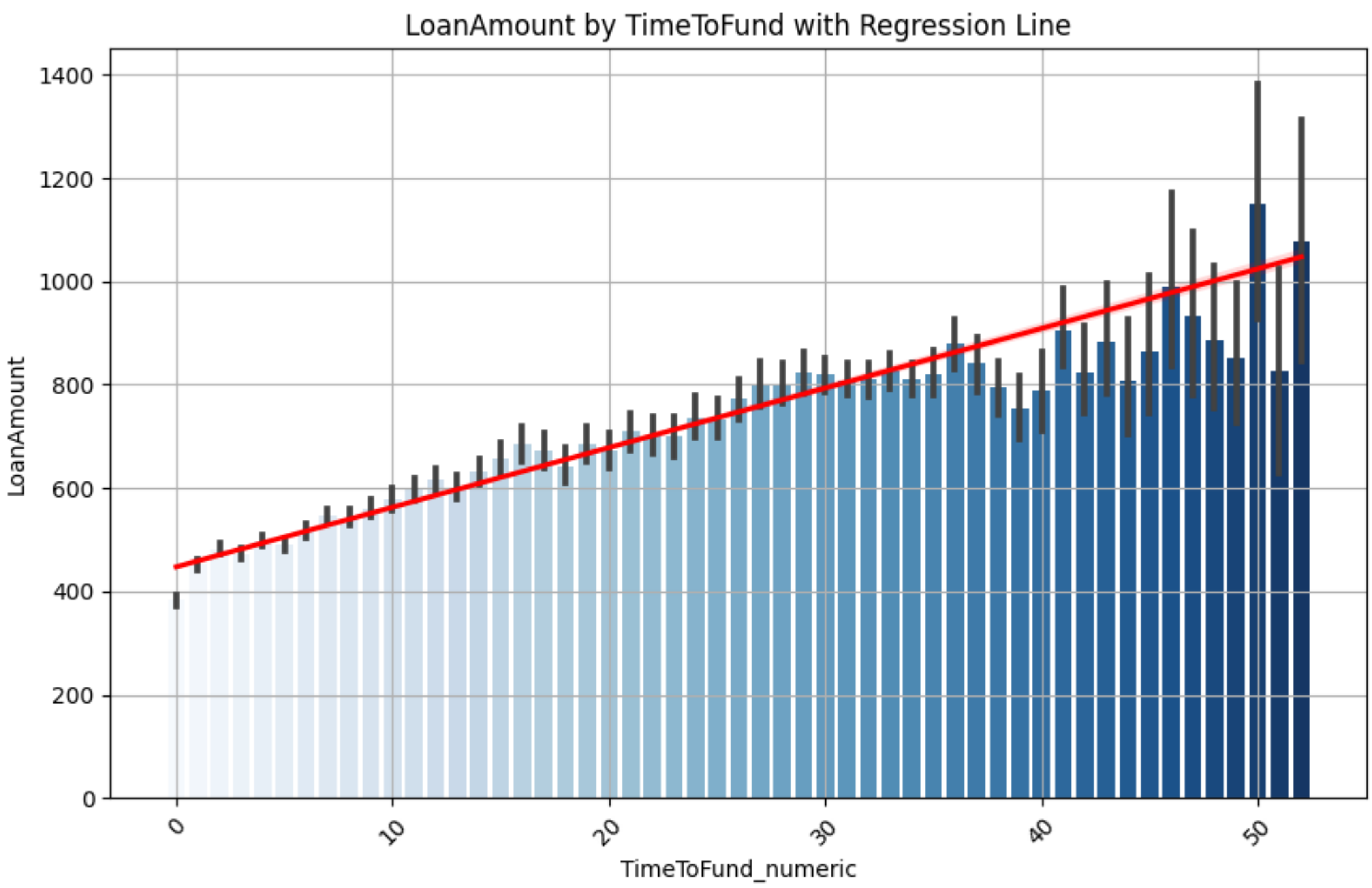
	id	LoanAmount	TimeToFund
0	109570	575	0
1	111913	900	1
2	1457371	200	1
3	1470250	700	21
4	228017	450	3
...	...	...	...
65278	649617	250	5
65279	755176	200	11
65280	1011000	225	2
65281	1011212	500	12
65282	808806	100	4

65283 rows × 3 columns

1. Plot LoanAmount against TimeToFund to get a sense of the relationship between these two variables.

In [54]:

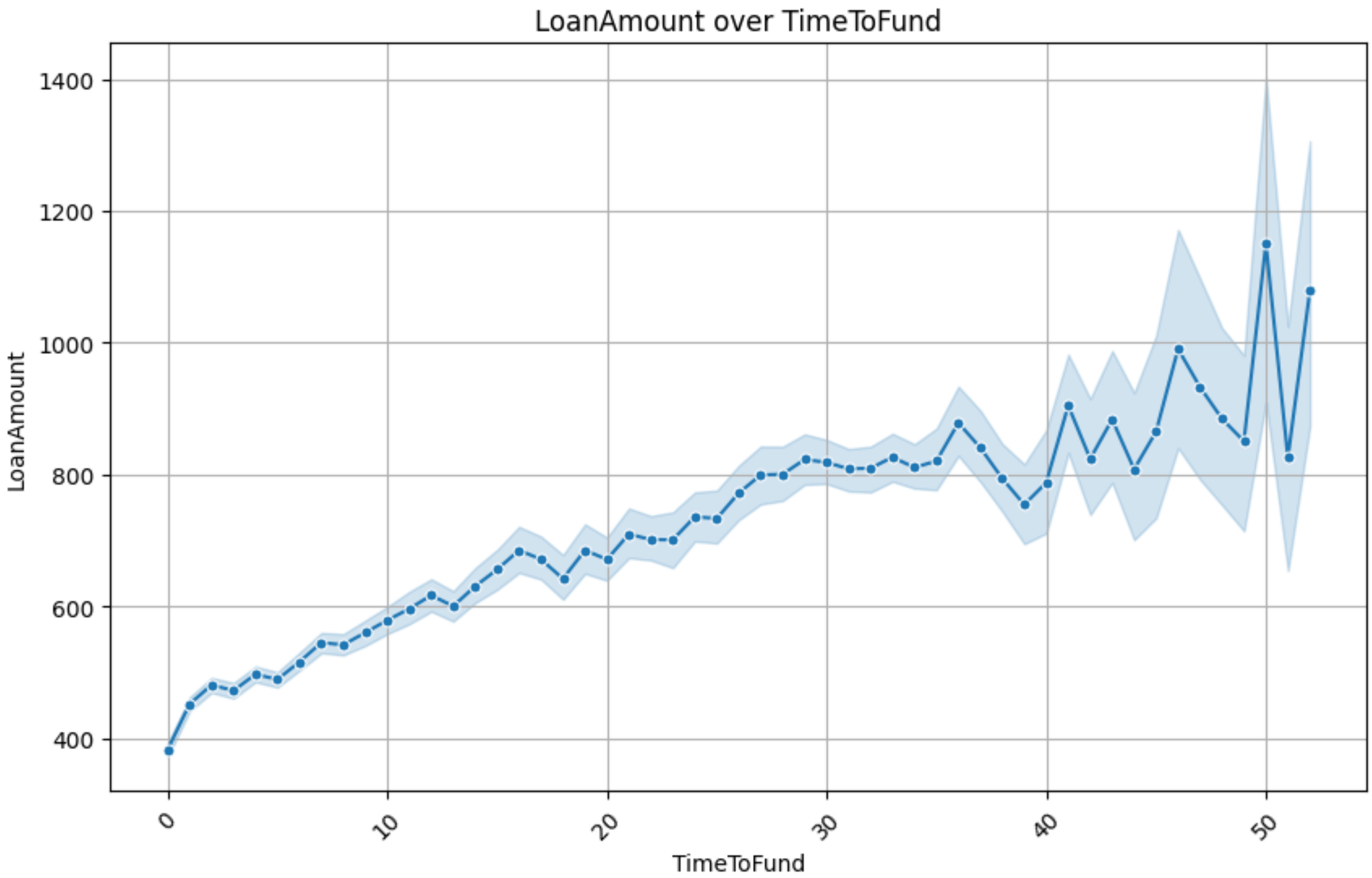
```
plt.figure(figsize=(10, 6))
sns.barplot(x="TimeToFund", y="LoanAmount", data=data, palette="Blues")
plt.title("LoanAmount by TimeToFund with Regression Line")
plt.xlabel("TimeToFund")
plt.ylabel("LoanAmount")
plt.xticks(rotation=45)
plt.xticks(np.arange(0, 51, 10))
plt.grid(True)

# Convert TimeToFund to numeric for regression line
data["TimeToFund_numeric"] = pd.to_numeric(data["TimeToFund"])

# Add regression line
sns.regplot(x="TimeToFund_numeric", y="LoanAmount", data=data, scatter=False, color='red')

plt.show()
```



In [55]:

```
plt.figure(figsize=(10, 6))
sns.lineplot(x="TimeToFund", y="LoanAmount", data=data, marker='o', markersize=5)
plt.title("LoanAmount over TimeToFund")
plt.xlabel("TimeToFund")
plt.ylabel("LoanAmount")
plt.xticks(rotation=45)
plt.grid(True)
plt.show()
```

```
/Users/ilsuucar/anaconda3/lib/python3.11/site-packages/seaborn/_oldcore.py:1119: FutureWarning: use_inf_as_na option is deprecated and will be removed in a future version. Convert inf values to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
/Users/ilsuucar/anaconda3/lib/python3.11/site-packages/seaborn/_oldcore.py:1119: FutureWarning: use_inf_as_na option is deprecated and will be removed in a future version. Convert inf values to NaN before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
```

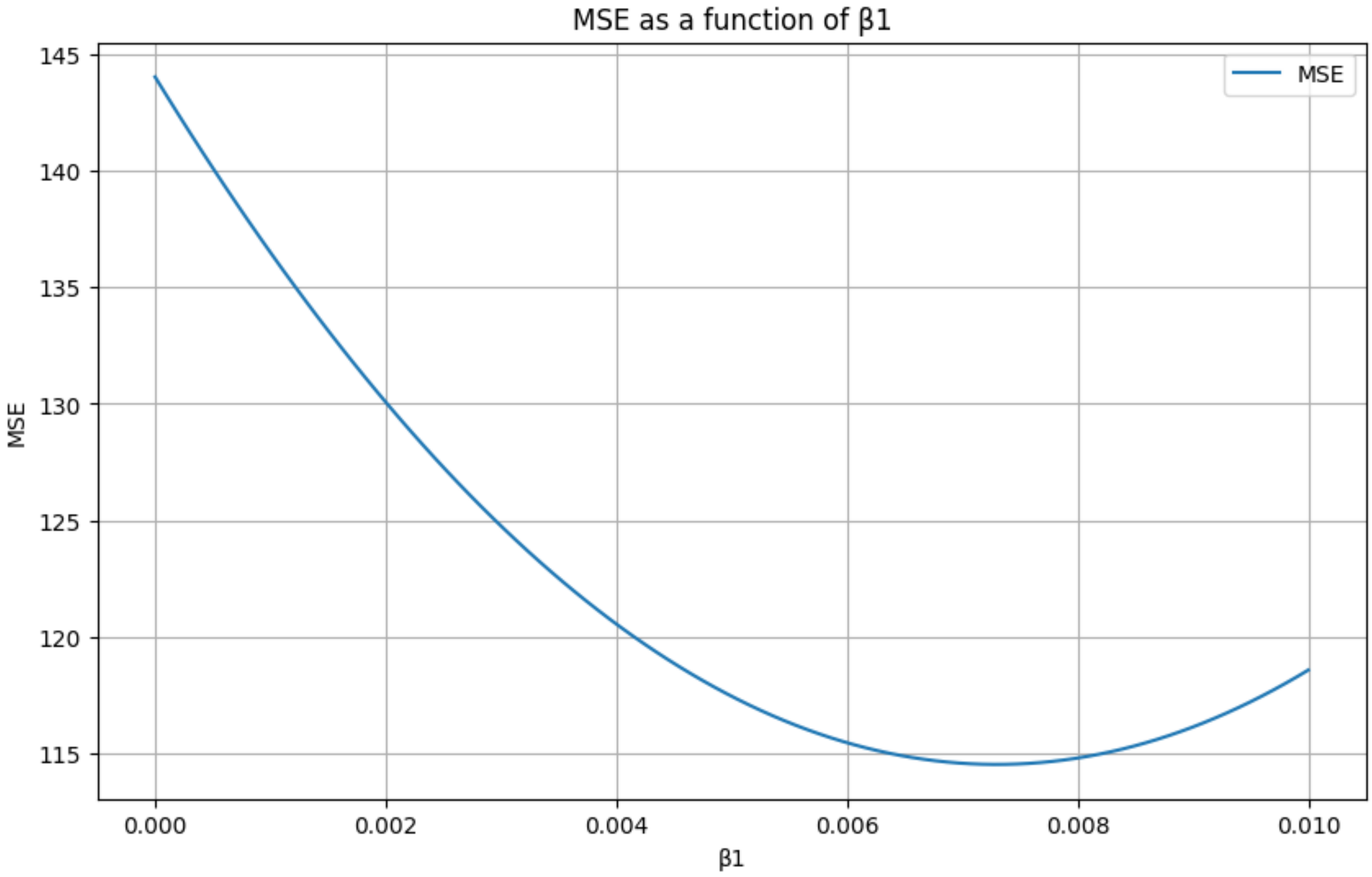


2. Plot the objective function (average loss) as a function of $\beta_1 \in [0,.01]$, keeping β_0 fixed at 7.

In [56]:
```python
# Functino to calculate MSE
def MSE(beta_0, beta_1, y, X):
    residuals = y - (beta_0 + beta_1 * X)
    return sum(residuals**2) / len(y)

y = data["TimeToFund"]
X = data["LoanAmount"]
beta_0 = 7
beta_1 = 0.1


f = MSE(beta_0, beta_1, y, X)
print("The value of the MSE: " + str(f))
```

```
The value of the MSE: 4880.036234548044
```

In [57]:
```python
beta_1_values = np.linspace(0, 0.01, 100)

mse_values = [MSE(7, beta_1, y, X) for beta_1 in beta_1_values]

plt.figure(figsize=(10, 6))
plt.plot(beta_1_values, mse_values, label='MSE')
plt.title('MSE as a function of β1')
plt.xlabel('β1')
plt.ylabel('MSE')
plt.legend()
plt.grid(True)
plt.show()
```



3. Use the gradient function to optimize the MSE via gradient descent, starting at β_0 = 5 and β_1 = .005. Use a learning rate of λ = .0001 and 1000 iterations. Why does the algorithm yield NaNs for β_0 and β_1?

If the learning rate is **too big**, then we will make too big steps in the gradient descent. This has the advantage of going down quickly to the minimum of the loss function, but we risk **missing the minimum** by oscillating around it at **infinity**, which is what happens in *this example*.

In [58]:
```
def residuals(beta_0, beta_1, y, X):
    return y - (beta_0 + beta_1 * X)


def gradientOLS(beta_0, beta_1, y, X):
    error = residuals(beta_0, beta_1, y, X)
    g0 = 2 * sum(error) / len(y)
    g1 = 2 * sum(error * X) / len(y)
    return (g0, g1)
```

In [59]:
```
# Initial values for β0 and β1
beta_0 = 5
beta_1 = 0.005

# Learning rate and number of iterations
learning_rate = 0.0001
iterations = 1000

# List for values of β0 and β1
beta_0_values = [beta_0]
beta_1_values = [beta_1]

# Gradient decent
for _ in range(iterations):
    # calculate gradient for current iteration
    g0, g1 = gradientOLS(beta_0, beta_1, y, X)

    # Update β0 and β1
    beta_0 -= learning_rate * g0
    beta_1 -= learning_rate * g1

    # Saving updated values
    beta_0_values.append(beta_0)
    beta_1_values.append(beta_1)

# Show final values for β0 and β1
print("Final β0:", beta_0_values[-1])
print("Final β1:", beta_1_values[-1])
```

```
Final β0: -inf
Final β1: -inf
```

4. Does it help to change the learning rate?

Yes, it does help to change the learning rate since we get results that are **not indefinite**.

In [60]:
```
# Initial values for β0 and β1
beta_0 = 5
beta_1 = 0.005

# Learning rate and number of iterations
learning_rate = 0.00000001
iterations = 1000

# List for values of β0 and β1
beta_0_values = [beta_0]
beta_1_values = [beta_1]

# Gradient decent
for _ in range(iterations):
    # calculate gradient for current iteration
    g0, g1 = gradientOLS(beta_0, beta_1, y, X)

    # Update β0 and β1
    beta_0 -= learning_rate * g0
    beta_1 -= learning_rate * g1

    # Saving updated values
    beta_0_values.append(beta_0)
    beta_1_values.append(beta_1)

# Show final values for β0 and β1
print("Final β0:", beta_0_values[-1])
print("Final β1:", beta_1_values[-1])
```

```
Final β0: 4.716643073200601
Final β1: -270.7339258378315
```

5. What happens when we express LoanAmount in 1000USD terms rather than in raw dollar terms? Try a learning rate of λ ∈ {.1,.01}.

From the results we can see that the final value for β_1 **changes drastically** and adapts a **similar range** to that of β_0. In addition, this also changes the scale of the gradient and allows **bigger learning rates** to be used to find the optimal values for β_0 and β_1 **faster**.

In [61]:
```
X = data["LoanAmount"]/1000

# Initial values for β0 and β1
beta_0 = 5
beta_1 = 0.005

# Learning rate and number of iterations
learning_rate = [0.1,0.01]
iterations = 1000

# Gradient decent
for learning_rate in learning_rate:
    trys = 0
    for _ in range(iterations):
        # calculate gradient for current iteration
        trys += 1
        g0, g1 = gradientOLS(beta_0, beta_1, y, X)

        # List for values of β0 and β1
        beta_0_values = [beta_0]
        beta_1_values = [beta_1]

        # Update β0 and β1
        beta_0 -= learning_rate * g0
        beta_1 -= learning_rate * g1

        # Saving updated values
        beta_0_values.append(beta_0)
        beta_1_values.append(beta_1)


    # Show final values for β0 and β1
    print(f"Final β0 for learning rate {learning_rate}:", str(beta_0_values[-1]), " and the number of total iterati
    print(f"Final β1 for learning rate {learning_rate}", str(beta_1_values[-1]), " and the number of total iteratio
```

```
Final β0 for learning rate 0.1: -6.65637292426726e+107  and the number of total iterations is 1000
Final β1 for learning rate 0.1 -4.575035432423971e+107  and the number of total iterations is 1000
Final β0 for learning rate 0.01: -6.409219134347971e+119  and the number of total iterations is 1000
Final β1 for learning rate 0.01 -4.405162536328034e+119  and the number of total iterations is 1000
```