Runge-Kutta Methods

One difficulty in high order Taylor series methods lies in the fact that it uses the higher order derivatives $x'', x''', \cdots$, which might be very difficult to get.

A better method should only use $f(t,x)$, not its derivatives. These methods are called Runge-Kutta methods.

1st order method: The same as Euler's method.

2nd order method: Let $h = t_{k+1} - t_k$. Given x_k, the next value x_{k+1} is computed as

$$x_{k+1} = x_k + \frac{1}{2}(K_1 + K_2)$$

where

$$\begin{cases} K_1 = h \cdot f(t_k, x_k) \\ K_2 = h \cdot f(t_k + h, x_k + K_1) \end{cases}$$

This is called Heun's method.

Theorem: Heun's method is of second order.

Proof: It suffices to show that the local truncation error is of order 3.

Taylor expansion in two variables gives

$$f(t_k + h, x_k + K_1) = f(t_k, x_k) + hf_t(t_k, x_k) + K_1 f_x(t_k, x_k) + \mathcal{O}\left(h^2, K_1^2\right).$$

We have $K_1 = hf(t_k, x_k)$, so the last term above is actually $\mathcal{O}\left(h^2\right)$. We also have

$$K_2 = h\left[f(t_k, x_k) + hf_t(t_k, x_k) + hf(t_k, x_k) f_x(t_k, x_k) + \mathcal{O}\left(h^2\right)\right]$$

Then, our method is:

$$\begin{aligned} x_{k+1} &= x_k + \frac{1}{2}\left[hf + hf + h^2 f_t + h^2 f f_x + \mathcal{O}\left(h^3\right)\right] \\ &= x_k + hf + \frac{1}{2}h^2\left[f_t + f f_x\right] + \mathcal{O}\left(h^3\right) \end{aligned}$$

$$x_{k+1} = x_k + hf + \frac{1}{2}h^2\left[f_t + f f_x\right] + \mathcal{O}\left(h^3\right)$$

Compare this with Taylor expansion for $x(t_{k+1}) = x(t_k + h)$

$$\begin{aligned} x(t_k + h) &= x(t_k) + hx'(t_k) + \frac{1}{2}h^2 x''(t_k) + \mathcal{O}\left(h^3\right) \\ &= x(t_k) + hf(t_k, x_k) + \frac{1}{2}h^2\left[f_t + f_x x'\right] + \mathcal{O}\left(h^3\right) \\ &= x(t_k) + hf + \frac{1}{2}h^2\left[f_t + f_x f\right] + \mathcal{O}\left(h^3\right). \end{aligned}$$

We see the first 3 terms are identical, this gives the local truncation error:

$$e_L = |x_{k+1} - x(t_k + h)| = \mathcal{O}(h^3)$$

Integral form and Trapezoid rule:

Integrating the ODE $x' = f(t, x)$ over the integral $t \in [t_k, t_k + h]$, we get

$$x(t_k + h) = x(t_k) + \int_{t_k}^{t_k+h} x'(t)dt = x(t_k) + \int_{t_k}^{t_k+h} f(t, x(t))dt.$$

Once $x(t_k) \approx x_k$ is given, then $x_{k+1} \approx x(t_k + h)$ can be computed by suitably approximating the integral. For the Heun's method, we see that

$$K_1 \approx hx'(t_k), \quad K_2 \approx hx'(t_k + h).$$

Then, the trapezoid rule

$$\int_{t_k}^{t_k+h} x'(t)dt \approx \frac{h}{2}\left[x'(t_k) + x'(t_k + h)\right] = \frac{1}{2}(K_1 + K_2)$$

4th order Runge-Kutta method

These methods take the form

$$x_{k+1} = x_k + w_1 K_1 + w_2 K_2 + \cdots + w_m K_m$$

where

$$\begin{cases} K_1 = h \cdot f(t_k, x_k) \\ K_2 = h \cdot f(t_k + a_2 h, x + b_2 K_1) \\ K_3 = h \cdot f(t_k + a_3 h, x + b_3 K_1 + c_3 K_2) \\ \quad \vdots \\ K_m = h \cdot f\left(t_k + a_m h, x + \sum_{i=1}^{m-1} \phi_i K_i\right) \end{cases}$$

The parameters w_i, a_i, b_i, ϕ_i are carefully chosen to guarantee the order m.

This choice is not unique.

This elegant 4th order method takes the form

$$x_{k+1} = x_k + \frac{1}{6}[K_1 + 2K_2 + 2K_3 + K_4]$$

where

$$
\begin{aligned}
K_1 &= h \cdot f\left(t_k, x_k\right), \\
K_2 &= h \cdot f\left(t_k + \frac{1}{2}h, x_k + \frac{1}{2}K_1\right), \\
K_3 &= h \cdot f\left(t_k + \frac{1}{2}h, x_k + \frac{1}{2}K_2\right), \\
K_4 &= h \cdot f\left(t_k + h, x_k + K_3\right).
\end{aligned}
$$

Integral form and Simpsons rule.

The integral form of the ODE $x' = f(t, x)$ gives

$$
x\left(t_k + h\right) = x\left(t_k\right) + \int_{t_k}^{t_k+h} x'(t)dt = x\left(t_k\right) + \int_{t_k}^{t_k+h} f(t, x(t))dt.
$$

For the RK4 method, we see that

$$
K_1 \approx hx'\left(t_k\right), \quad K_2 \approx hx'\left(t_k + h/2\right), \quad K_3 \approx hx'\left(t_k + h/2\right), \quad K_4 \approx hx'\left(t_k + h\right)
$$

Then, the Simpson's rule

$$
\int_{t_k}^{t_k+h} x'(t)dt \approx \frac{h}{6}\left[x'\left(t_k\right) + 4x'\left(t_k + h/2\right) + x'\left(t_k + h\right)\right] = \frac{1}{6}\left(K_1 + 2K_2 + 2K_3 + K_4\right)
$$

Optional Topics

1. Explicit Adam-Bashforth method
2. Implicit Adam-Bashforth-Moulton (ABM) methods

First order systems of ODEs

We consider

$$
\vec{x}' = F(t, \vec{x}), \quad \vec{x}\left(t_0\right) = \vec{x}_0
$$

Here $\vec{x} = \left(x_1, x_2, \cdots, x_n\right)^T$ is a vector, and $F = \left(f_1, f_2, \cdots, f_n\right)^T$ is a vector-valued function.

Write it out

$$
\begin{cases}
x_1' = f_1\left(t, x_1, x_2, \cdots, x_n\right) \\
x_2' = f_2\left(t, x_1, x_2, \cdots, x_n\right) \\
\cdots \\
x_n' = f_n\left(t, x_1, x_2, \cdots, x_n\right)
\end{cases}
$$

All methods for scalar equation can be used for systems of first order ODEs.

Taylor series methods:

$$\vec{x}(t+h) = \vec{x} + h\vec{x}' + \frac{1}{2}h^2\vec{x}'' + \cdots + \frac{1}{m!}h^m\vec{x}^{(m)}$$

Higher order ODEs and systems

Consider the higher order ODE

$$u^{(n)} = f\left(t, u, u', u'', \cdots, u^{(n-1)}\right), \quad \text{ICs:} \quad u\left(t_0\right), u'\left(t_0\right), u''\left(t_0\right), \cdots, u^{(n-1)}\left(t_0\right).$$

Introduce a systematic change of variables

$$x_1 = u, \quad x_2 = u', \quad x_3 = u'', \quad \cdots \quad x_n = u^{(n-1)}.$$

We then have

$$\begin{cases} x_1' = u' = x_2 \\ x_2' = u'' = x_3 \\ x_3' = u''' = x_4 \\ \quad\vdots \\ x_{n-1}' = u^{(n-1)} = x_n \\ x_n' = u^{(n)} = f\left(t, x_1, x_2, \cdots, x_n\right) \end{cases}$$

This is a system of 1st order ODEs, with initial data given at

$$x_1\left(t_0\right) = u\left(t_0\right), \quad x_2\left(t_0\right) = u'(t_0)\,, \quad \cdots \quad , x_n(t_0) = u^{(n-1)}\left(t_0\right).$$

Gradient Descent

In gradient descent, our objective is to solve the optimization problem: $\min_{x\in\mathbb{R}^d} \; f(x)$, where f is a convex function $f : \mathbb{R}^n \to \mathbb{R}$.

Algorithm description:

- Start with some initial guess, x_0.
- Generate new guess x_1 by moving in the negative gradient direction:

  $$x_1 = x_0 - \alpha_0 \nabla f\left(x_0\right),$$

  where α_0 is the step size.
- Repeat to successively to refine the guess:

  $$x_{k+1} = x_k - \alpha_k \nabla f\left(x_k\right), \quad \text{for } k = 1, 2, 3, \ldots$$

  where we might use a different step-size α_k on each iteration.
- Stop when stopping criteria is met.

- In practice, you can stop if you detect that you aren't making progress, or if $\left\|\nabla f\left(w^k\right)\right\| \le \epsilon$, or if a certain number of iterations has been conducted.

Assumption (Lipschitz Gradients):

> The function $f : \mathbb{R}^n \to \mathbb{R}$ has L-Lipschitz continuous gradients.
>
> In other words, there exists a positive constant L such that
>
> $$\|\nabla f(v) - \nabla f(w)\| \leq L\|v - w\|, \quad \forall v, w \in \mathbb{R}^d$$

What this assumption suggests is that “Gradients cannot change arbitrarily fast”.

Lemma (Descent Lemma):

> If $f : \mathbb{R}^n \to \mathbb{R}$ has L-Lipschitz continuous gradients, then
>
> $$f(v) \leq f(w) + \nabla f(w)^T(v - w) + \frac{L}{2}\|v - w\|^2, \quad \forall v, w \in \mathbb{R}^d$$

The descent lemma gives us a convex quadratic upper bound on f.

A proof to this lemma can be found at "Zhou, Xingyu. “On the Fenchel Duality between Strong Convexity and Lipschitz Continuous Gradient.” arXiv preprint arXiv:1803.06573 (2018)". It is fairly simple, as long as you have taken courses in linear algebra and calculus, you should be able to understand the proof.

Theorem:

> If $f : \mathbb{R}^n \to \mathbb{R}$ has L-Lipschitz continuous gradients, and $f^* = \min_x f(x) > -\infty$, then the gradient descent algorithm with fixed step size satisfying $\alpha < \frac{2}{L}$ will converge to a stationary point.

Here we assume a fixed step size and $\alpha < \frac{2}{L}$ only for the purpose of a mathematical proof. In practice, you should never use $\alpha < \frac{2}{L}$. L is usually expensive to compute, and this step-size is really small. You only need a step-size this small in the worst case. Usually, you can start with a larger step size and gradually decrease it.

Note that the Descent Lemma also holds for C^1 functions, you can find more information online if you are interested.

Proof of the Theorem:

Sketch of the proof: we show that f is always decreasing in every iteration; that is $f\left(x_{k+1}\right) \leq f\left(x_k\right) - \xi$ for some $\xi > 0$.

Recall the Descent Lemma:

$$f(v) \leq f(w) + \nabla f(w)^T(v - w) + \frac{L}{2}\|v - w\|^2$$

If we substitle x_{k+1} and x_k into the descent lemma we get:

$$f\left(x_{k+1}\right) \leq f\left(x_k\right) + \nabla f(x_k)^T\left(x_{k+1} - x_k\right) + \frac{L}{2}\|x_{k+1} - x_k\|^2$$

Now if we use that $\left(x_{k+1} - x_k\right) = -\alpha \nabla f\left(x_k\right)$ in gradient descent:

$$
\begin{aligned}
f(x_{k+1}) &\leq f(x_k) - \alpha \nabla f(x_k)^T \nabla f(x_k) + \frac{\alpha L}{2} \|\alpha \nabla f(x_k)\|^2 \\
&= f(x_k) - \alpha \|\nabla f(x_k)\|^2 + \frac{\alpha^2 L}{2} \|\nabla f(x_k)\|^2 \\
&= f(x_k) - \frac{2\alpha - \alpha^2 L}{2} \|\nabla f(x_k)\|^2
\end{aligned}
$$

When $\alpha < \frac{2}{L}$, the second term is always positive:

$$
\frac{2\alpha - \alpha^2 L}{2} > 0
$$

We have derived a bound on guaranteed progress.

Rewrite the equation:

$$
\|\nabla f(x_k)\|^2 \leq \frac{2}{2\alpha - \alpha^2 L} [f(x_k) - f(x_{k+1})]
$$

Sum up the squared norms of all the gradients up to iteration T:

$$
\sum_{k=0}^{T} \|\nabla f(x_k)\|^2 \leq \frac{2}{2\alpha - \alpha^2 L} \sum_{k=0}^{T} [f(x_k) - f(x_{k+1})]
$$

Apply some middle school algebra tricks:

$$
\begin{aligned}
\sum_{k=0}^{T} [f(x_k) - f(x_{k+1})] &= f(x_0) - \underbrace{f(x_1) + f(x_1)}_{0} - \underbrace{f(x_2) + f(x_2)}_{0} - \ldots f(x_{T+1}) \\
&= f(x_0) - f(x_{T+1})
\end{aligned}
$$

Now, we have:

$$
\begin{aligned}
\sum_{k=0}^{T} \|\nabla f(x_k)\|^2 &\leq \frac{2}{2\alpha - \alpha^2 L} (f(x_0) - f(x_{T+1})) \\
&\leq \frac{2}{2\alpha - \alpha^2 L} (f(x_0) - f^*)
\end{aligned}
$$

Finally,

$$
\frac{\sum_{k=0}^{T} \|\nabla f(x_k)\|^2}{T} \leq \left[\frac{2}{2\alpha - \alpha^2 L} (f(x_0) - f^*)\right] \cdot \frac{1}{T}
$$

This concludes the convergence of GD, with a convergence rate of $O(\frac{1}{T})$. Gradient descent requires $T = O(\frac{1}{\epsilon})$ iterations to achieve $\|\nabla f(x_T)\|^2 \leq \epsilon$.

Summary of GD and Takeaways from the Proof

- Gradient descent is suitable for solving high-dimensional problems.
- Guaranteed progress if gradient is Lipschitz.
- Practical step size strategies (e.g. step decay step-size).

Given a function $f(x) = \sqrt{x^2 + 5}$, $x \in \mathbf{R}$.

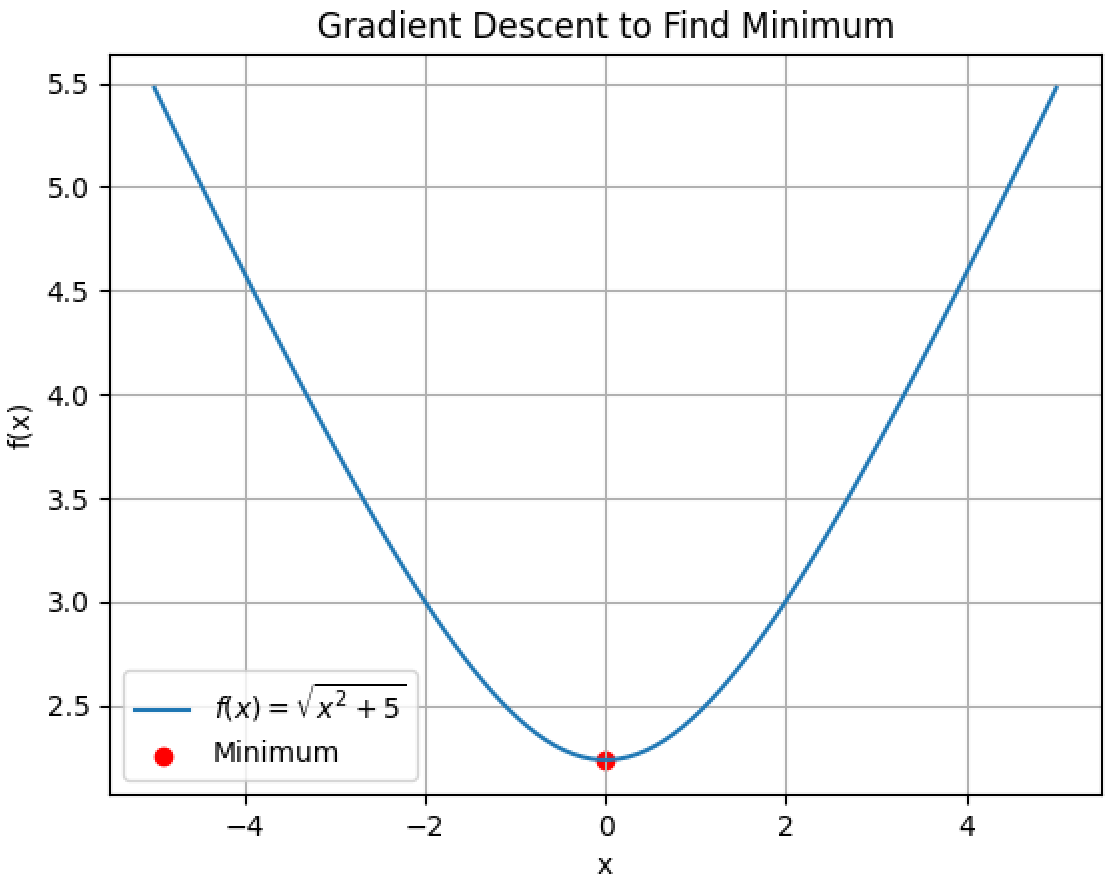
Fact: This function is L-smooth.

1. Find the gradient of this function and the minimum value of this function analytically.

1. Plot this function over the interval $[-5, 5]$.
2. Perform the Gradient Descent algorithm to find the minimum value of f for 50 iterations (T) with a step size of 1 (α). Use 3.26 (maybe use last digit SBID/3) as the initial guess.
3. Record the values of x_k at the k-th iteration during GD and report x_T.
4. Plot the value of $f(x_k)$ v.s. the iteration number k.
5. For each of the step sizes 5, 3, 1, and 0.5, perform gradient descent and record the values of x_k in each step k. Plot $f(x_{k-1}) - f(x_k)$ v.s. k for each step size. Your graphs should all be included in a single plot. Examine if $f(x_{k-1}) - f(x_k)$ (which means that $f(x_k)$ is always decreasing) is alway positive for all k.

In [ ]:

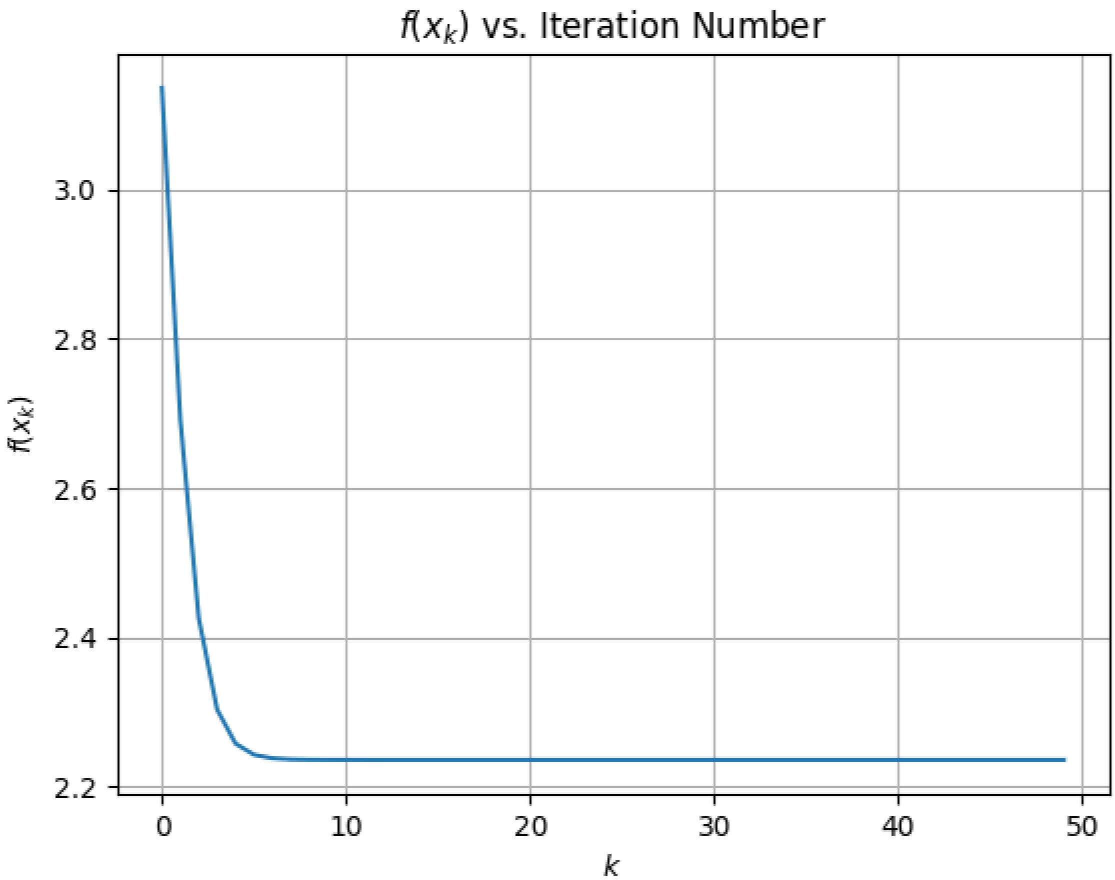
```
import numpy as np
import matplotlib.pyplot as plt

# Define the function f(x) = x^2 + 4x + 4
def f(x):
    return np.sqrt(x**2 + 5)

# Plot the function f(x)
x = np.linspace(-5, 5, 100)
plt.plot(x, f(x), label=r'$f(x) = \sqrt{x^2+5}$')

# Plot the minimum found by gradient descent
plt.scatter(0, np.sqrt(5), color='red', marker='o', label='Minimum')

plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('Gradient Descent to Find Minimum')
plt.legend()
plt.grid(True)
plt.show()
```



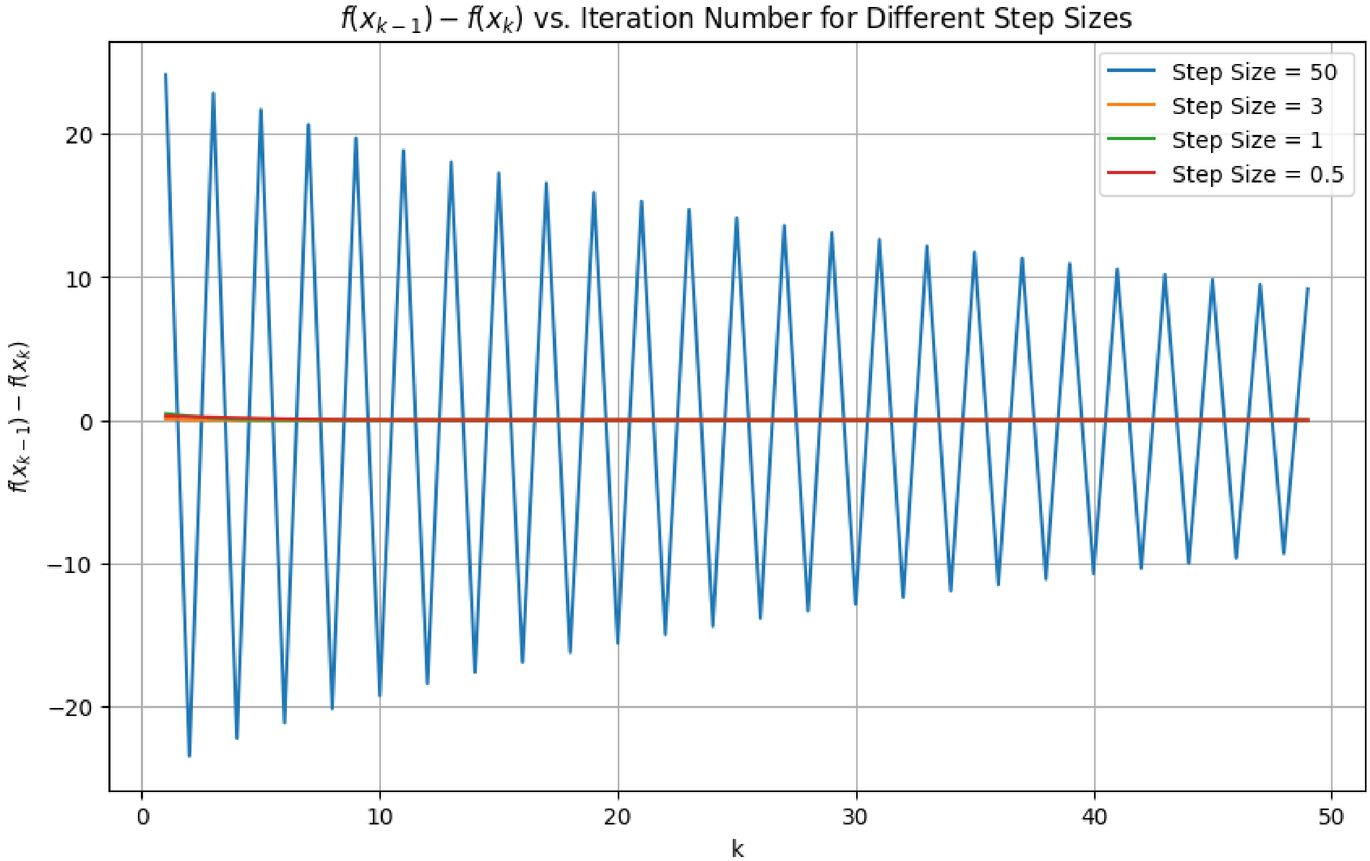
In [ ]:

```python
# Define the derivative of the function f(x)
def df(x):
    return x / np.sqrt(x**2 + 5)

# Gradient Descent Algorithm
def gradient_descent(initial_x, step_size, num_iterations):
    x_values = []  # Store x values at each iteration
    gradient_values = []  # Store absolute gradient values at each iteration
    x = initial_x

    for i in range(num_iterations):
        gradient = df(x)
        x -= step_size * gradient
        x_values.append(x)

    return x_values

# Initial values
initial_x = 3
step_size = 1
num_iterations = 50

# Run gradient descent
x_values = gradient_descent(initial_x, step_size, num_iterations)
print(x_values[-1])
# Plot the absolute gradient values vs. iteration number
plt.plot(range(num_iterations), f(np.array(x_values)))
plt.xlabel(r'$k$')
plt.ylabel(r'$f(x_k)$')
plt.title(r'$f(x_k)$ vs. Iteration Number')
plt.grid(True)
plt.show()
```

```
8.230507466005927e-13
```



In [ ]:
```
step_sizes = [50,3, 1, 0.5]
step_size_x_values = []
for step_size in step_sizes:
    x_values = gradient_descent(initial_x, step_size, num_iterations)
    step_size_x_values.append(np.array(x_values))

plt.figure(figsize=(10, 6))
for i, step_size in enumerate(step_sizes):
    plt.plot(range(1, num_iterations), f(np.array(step_size_x_values[i]))[0:49] - f(np.a

plt.xlabel(r'k')
plt.ylabel(r'$f(x_{k-1}) - f(x_{k})$')
plt.title('$f(x_{k-1}) - f(x_{k})$ vs. Iteration Number for Different Step Sizes')
plt.legend()
plt.grid(True)
plt.show()
```


$f(x_{k-1}) - f(x_k)$ vs. Iteration Number for Different Step Sizes
Step Size = 50
Step Size = 3
Step Size = 1
Step Size = 0.5
20
10
0
−10
−20
$f(x_{k-1}) - f(x_k)$
0
10
20
30
40
50
k