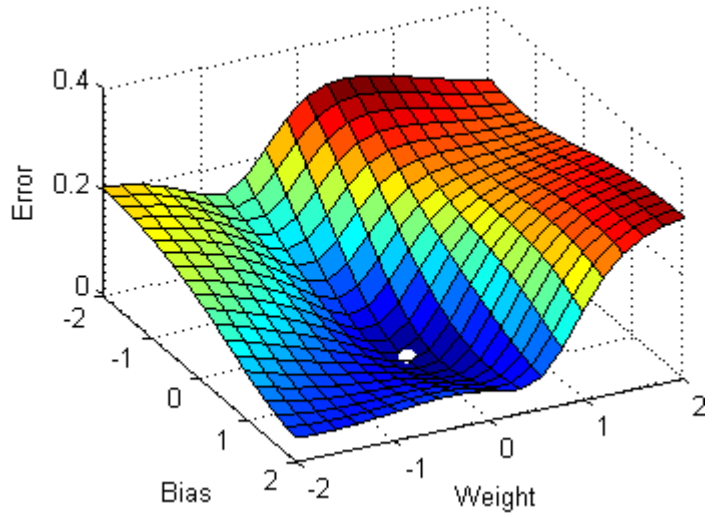
Lab 5: Optimization of functions

Authors: Juan Andrés Méndez, German Montoya, Carlos Lozano

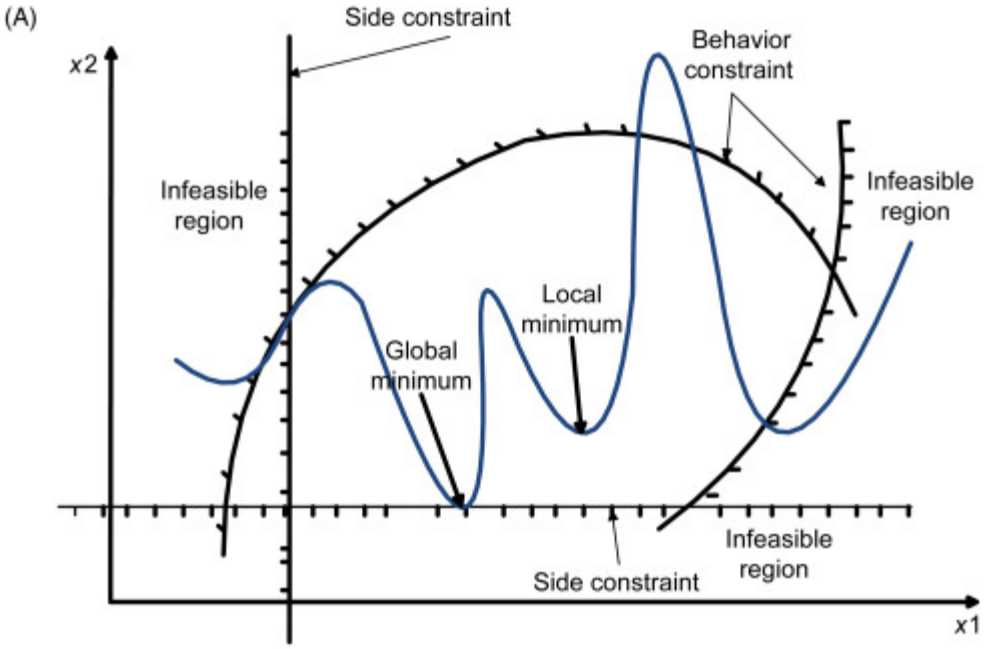
In this lab we are going to learn how to optimize functions using Python. We will be using the `scipy`, `numpy`, `sympy` and `matplotlib` libraries.

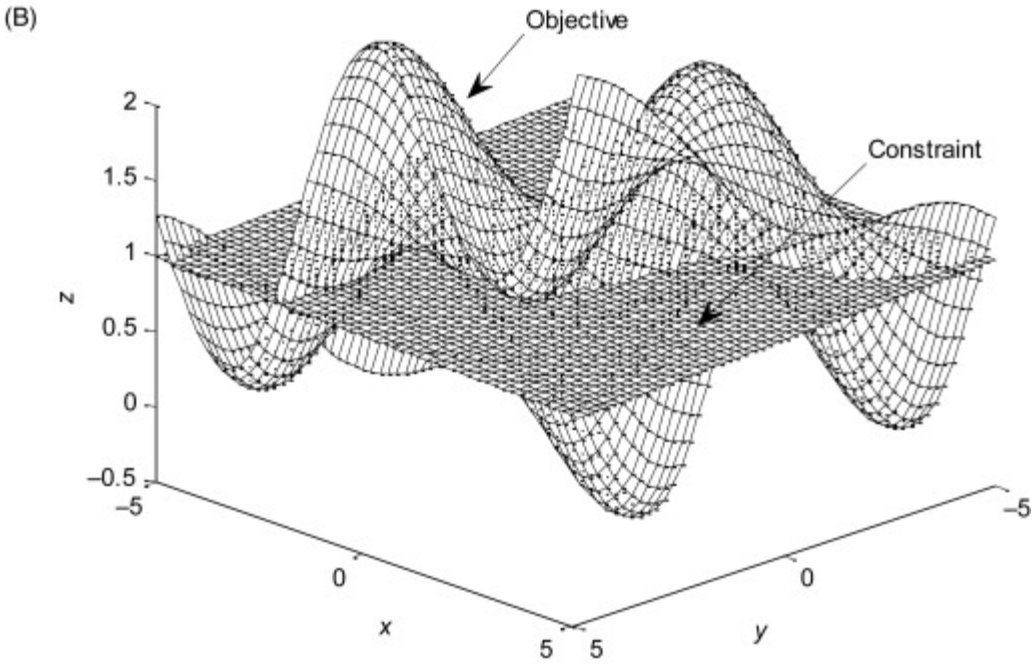
What is function optimization?



Function optimization is the process of finding the input value that results in the minimum or maximum output value of a given function. This is a common problem in mathematics and computer science, and it has many practical applications in fields such as engineering, economics, and machine learning.

How it diferenciates from optimization techniques such as Pyomo and GAMS?





Optimization techniques such as Pyomo and GAMS are used to solve complex optimization problems that involve multiple variables and constraints. These tools are typically used to find the optimal solution to a given problem, subject to a set of constraints. In contrast, function optimization is focused on finding the optimal value of a single function, without any constraints. Just given a certain range of values, we want to find the value that minimizes or maximizes the function.

What are the most common optimization algorithms?

There are many optimization algorithms that can be used to optimize functions. Some of the

most common ones include:

- **Gradient descent**: An iterative optimization algorithm that uses the gradient of the function to find the minimum value.
- **Gauss-Newton algorithm**: An optimization algorithm that is used to solve non-linear least squares problems.
- **Hill climbing**: An optimization algorithm that starts at a random point and iteratively moves towards the direction of the steepest ascent.
- **Simplex algorithm**: An optimization algorithm that is used to solve linear programming problems.
- **Levenberg-Marquardt algorithm**: An optimization algorithm that is used to solve non-linear least squares problems.
- **Newton's method**: An optimization algorithm that uses the second derivative of the function to find the minimum value.

> Each of these algorithms has its own strengths and weaknesses, and the choice of algorithm will depend on the specific problem that you are trying to solve. Due to time constraints, we will focus on the Newton's method in this lab.

Newton's method

The objective for this lab is to implement the Newton's method to find the minimum value of a given function. The Newton's method is an iterative optimization algorithm that uses the second derivative of the function to find the minimum value. The algorithm works by starting at an initial guess for the minimum value, and then iteratively updating the guess until it converges to the true minimum value. In this lab, we will implement the Newton Raphson method in both 1D and 2D functions.

For a visual explanation please see the following video: Newton's method

1D Newton's Method

Now that we have the bases covered, let's start by defining the Newton's method for 1D functions. The algorithm is as follows:

Algorithm: 1D Newton's Method

---

1. Initialize:
   - $i \leftarrow 0$
   - $x_i \leftarrow$ initial guess

- tolerance $\leftarrow$ 0.0001

2. **while** $|f(x_i)| >$ tolerance **do**
    - Compute the next approximation:
        - $x_{i+1} \leftarrow x_i - \frac{f(x_i)}{f'(x_i)}$
    - Increment the iteration count:
        - $i \leftarrow i + 1$
3. **end while**
4. Converged solution:
    - $x_{min} \leftarrow x_i$

return x_{min}

2D Newton's Method

The Newton's method for 2D functions is similar to the 1D case, but it involves computing the gradient and the Hessian of the function. The algorithm is as follows:

Algorithm: 2D Newton's Method

---

1. Initialize:
    - $i \leftarrow 0$
    - $x_i \leftarrow$ initial guess
    - tolerance $\leftarrow$ 0.0001
2. **while** $||\nabla f(x_i)|| >$ tolerance **do**
    - Compute the next approximation:
        - $x_{i+1} \leftarrow x_i - H^{-1}(x_i)\nabla f(x_i)$
    - Increment the iteration count:
        - $i \leftarrow i + 1$
3. **end while**
4. Converged solution:
    - $x_{min} \leftarrow x_i$

return x_{min}

Where H is the Hessian matrix of the function, $\nabla f(x)$ is the gradient of the function, and $||\nabla f(x_i)||$ is the norm of the gradient.

Before we start

Now that we have the theory covered, we now need to know how we can use Python to implement the Newton's method. We will be using the `scipy`, `numpy`, `sympy` and `matplotlib` libraries to implement the algorithm.

Here we are going to cover up the basics of these libraries:

Let's start by exploring the `numpy` library.

Numpy

NumPy is a powerful Python library that is used for numerical computing. It provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays. NumPy is widely used in scientific computing, data analysis, and machine learning. We will cover up the following topics:

- Creating NumPy arrays
- Indexing and slicing arrays
- Performing mathematical operations on arrays
- Creating linespace and meshgrid

Creating `numpy` arrays

To create a NumPy array, you can use the `np.array()` function. You can pass a list, tuple, or any iterable object to the function to create an array. Here's an example:

```
import numpy as np

# Create a 1D array

a = np.array([1, 2, 3, 4, 5])

print(a)
```

Arrays can have multiple dimensions as well. You can create a 2D array by passing a list of lists to the `np.array()` function. Here's an example:

```
import numpy as np

# Create a 2D array
```

```
a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

print(a)
```

> Create an array with $n \times m$ dimensions and initialize it with zeros using the `np.zeros()` function. Here's an example:

```
import numpy as np

# Create a 2D array with zeros

a = np.zeros((3, 3))

print(a)
```

In [ ]:
```
"""
numpy arrays
 @TODO: Create a np array for the first 10 prime numbers
"""
import numpy as np
```

Indexing and slicing arrays

You can access elements of a NumPy array using indexing and slicing. Indexing is used to access a single element of the array, while slicing is used to access a subset of the array. Here's an example:

```
import numpy as np

# Create a 1D array

a = np.array([1, 2, 3, 4, 5])

# Access the first element of the array

print(a[0])

# Access a subset of the array using slicing

print(a[1:4])
```

In [ ]:
```
"""
Indexing and slicing arrays
@TODO: For the array created above, bring the last 5 elements
"""
```

Performing mathematical operations on arrays

NumPy provides a wide range of mathematical functions that can be used to perform operations on arrays. You can perform element-wise addition, subtraction, multiplication,

and division on arrays. Here's an example:

```
import numpy as np

# Create two arrays

a = np.array([1, 2, 3, 4, 5])
b = np.array([5, 4, 3, 2, 1])

# Perform element-wise addition

c = a + b

print(c)
```

> Dot product of two arrays can be calculated using the `np.dot()` function. Here's an example:

```
import numpy as np

# Create two arrays

a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

# Calculate the dot product

c = np.dot(a, b)

print(c)
```

> Matrix multiplication:

```
# Create two matrices

a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6], [7, 8]])

# Perform matrix multiplication

c = np.matmul(a, b)

print(c)
```

> Transpose of a matrix:

```
# Create a matrix
a = np.array([[1, 2], [3, 4]])

# Calculate the transpose

b = np.transpose(a)

print(b)
```

> Inverse of a matrix:

```
# Create a matrix

a = np.array([[1, 2], [3, 4]])

# Calculate the inverse

b = np.linalg.inv(a)

print(b)
```

In [ ]:
```
"""
Operations between numpy arrays and scalars
@TODO: Do the cross product between the two given matrices
"""

a = np.array([[1, 2, 3], [4, 5, 6]])

b = np.array([[1, 2, 3], [4, 5, 6]])
```

Creating linespace and meshgrid

NumPy provides functions to create evenly spaced arrays and grids. You can use the `np.linspace()` function to create an array of evenly spaced values between two endpoints. Here's an example:

```
x = np.linspace(0, 10, 100)

print(x)
```

Meshgrid is a function that generates coordinate matrices from coordinate vectors. You can use the `np.meshgrid()` function to create a meshgrid of 2D arrays. Here's an example:

> Create a meshgrid using the `np.meshgrid()` function. Here's an example:

```
x = np.linspace(0, 10, 100)
y = np.linspace(0, 10, 100)

X, Y = np.meshgrid(x, y)

print(X)
print(Y)
```

In [ ]:
```
"""
Linspaces and meshgrids
@TODO: Create a meshgrid for the following x and y values
"""

x = np.linspace(0, 1, 5)

y = np.linspace(0, 1, 5)
```

Scipy

Scipy is a Python library that is used for scientific and technical computing. It provides a wide range of functions for optimization, interpolation, integration, linear algebra, and more. Scipy is built on top of NumPy, and it provides additional functionality that is useful for scientific computing.

We will cover up the following topics:

- How to create functions in `scipy`
- Optimization with `scipy.optimize`
- Derivate and integrate with `scipy.misc`
- How to create a Hessian matrix using `scipy`

How to create functions in `scipy`

For creating a function in `scipy`, there is no need to import any specific module. You can create a function using the `lambda` keyword or by defining a function using the `def` keyword. Here's an example:

```
import numpy as np

# Create a lambda function

f = lambda x: x**2

print(f(2))
```

> You can also define a function using the `def` keyword. Here's an example:

```
import numpy as np

# 3 variables function

def f(x, y, z):
    return x**2 + y**2 + z**2

print(f(1, 2, 3))
```

Optimization with `scipy.optimize`

Scipy provides a module called `scipy.optimize` that contains functions for optimization. You can use these functions to find the minimum or maximum value of a function. The `scipy.optimize` module provides several optimization algorithms, including the Newton's method, BFGS, and Nelder-Mead algorithms.

> To find the minimum value of a function using the Newton's method, you can use the `scipy.optimize.minimize()` function. Here's an example:

Note: While some libraries like `scipy.optimize` provide optimization algorithms, it is important to understand the underlying theory and implementation of these algorithms to use them effectively. Thus it wont be valid to use the `scipy.optimize` module in this lab.

```
import numpy as np
from scipy.optimize import minimize

# Define the function

def f(x):
    return x**2

# Find the minimum value of the function

result = minimize(f, x0=0)

print(result.x)
```

In [ ]:
```
"""
using scipy optimize find the minimum of the following function
f(x) = x^2 + 10sin(x)

@TODO: Find the minimum of the function
"""
import scipy.optimize as opt
```

Derivate and integrate with `scipy.misc`

Scipy provides a module called `scipy.misc` that contains functions for numerical differentiation and integration. You can use these functions to compute derivatives and integrals of functions. The `scipy.misc` module provides functions like `derivative()` and `quad()` for numerical differentiation and integration, respectively.

> To compute the derivative of a function at a given point, you can use the `scipy.misc.derivative()` function. Here's an example:

```
import numpy as np
from scipy.misc import derivative
```

```
# Define the function

def f(x):
    return x**2

# Compute the derivative of the function at x=2

result = derivative(f, 2, dx=1e-6)

print(result)
```

> To compute the integral of a function over a given interval, you can use the `scipy.integrate.quad()` function. Here's an example:

```
import numpy as np
from scipy.integrate import quad

# Define the function

def f(x):
    return x**2

# Compute the integral of the function over the interval [0, 1]

result, error = quad(f, 0, 1)

print(result)
```

In [ ]:
```
"""
using scipy.misc derivative find the derivative of the following function

f(x) = x^2 + 10sin(x)
@TODO: Find the derivative of the function
"""
import scipy.misc as misc
```

Hessian matrix with `scipy`

`scipy` Really doesn't provide a direct way to calculate the Hessian matrix of a function. But you can use the derivative function along with `numpy` to calculate the Hessian matrix. Here's an example:

```
import numpy as np
from scipy.misc import derivative

# Define the function

def f(x, y):
    return x**2 + y**2
```

```
# Compute the Hessian matrix of the function at x=1, y=2

def dfdx(x, y):
    return derivative(lambda x: f(x, y), x, dx=1e-6)

def dfdy(x, y):
    return derivative(lambda y: f(x, y), y, dx=1e-6)

def d2fdxdy(x, y):
    return derivative(lambda x: dfdy(x, y), x, dx=1e-6)

def d2fdydx(x, y):
    return derivative(lambda y: dfdx(x, y), y, dx=1e-6)

H = np.array([[dfdx(1, 2), d2fdxdy(1, 2)], [d2fdydx(1, 2), dfdy(1,
2)]])

print(H)
```

In [ ]:
```
"""
Find the hessian matrix of the following function:
f(x,y) = 41x^2 + 2xy + 10y^2
@TODO: Use scipy.misc derivative and numpy to find the hessian matrix
"""
```

Sympy

Sympy is a Python library that is used for symbolic mathematics. It provides support for symbolic computation, algebraic manipulation, calculus, and more. Sympy is widely used in mathematics, physics, and engineering for solving complex mathematical problems.

- How to create symbolic expressions in `sympy`
- Differentiate and integrate symbolic expressions
- How to create a Hessian matrix using `sympy`
- How and when to use the `lambdify` function

How to create symbolic expressions in `sympy`

To create a symbolic expression in `sympy`, you need to import the `sympy` library and use the `symbols()` function to define symbolic variables. Here's an example:

```
import sympy as sp

# Define symbolic variables

x, y = sp.symbols('x y')

# Create a symbolic expression
```

```
expr = x**2 + y**2

print(expr)
```

In [ ]:
```
"""
Using sympy write the following function
f (x,y,z) = x^2 + y^2 + z^2
@TODO: Write the function
"""
```

Differentiate and integrate symbolic expressions

You can differentiate and integrate symbolic expressions in `sympy` using the `diff()` and `integrate()` functions, respectively. Here's an example:

```
import sympy as sp

# Define symbolic variables

x, y = sp.symbols('x y')

# Create a symbolic expression

expr = x**2 + y**2

# Differentiate the expression with respect to x

diff_expr = sp.diff(expr, x)

print(diff_expr)

# Integrate the expression with respect to x

int_expr = sp.integrate(expr, x)

print(int_expr)
```

In [ ]:
```
"""
Using sympy derive the following function:
f(x,y) = log((exp(x) + exp(y))^2) + x^2 + y^2
@TODO: Derive the function
"""
```

How to create a Hessian matrix using `sympy`

To create a Hessian matrix using `sympy`, you can use the 'hessian()' function. Here's an example:

```
import sympy as sp
```

```
# Define symbolic variables

x, y = sp.symbols('x y')

# Create a symbolic expression

expr = x**2 + y**2

variables = [1, 2]

# Compute the Hessian matrix of the expression

H = sp.hessian(expr, variables)

print(H)
```

In [ ]:
```
"""
Using hessian() from sympy find the hessian matrix of the following function
f(x,y) = x^2 + y^2 + x^2y + xy^2
x_0 = 1, y_0 = 2
"""
```

How and when to use the `lambdify` function

The `lambdify()` function in `sympy` is used to convert symbolic expressions into numerical functions. You can use the `lambdify()` function to create a function that can be evaluated at specific values. Here's an example:

```
import sympy as sp

# Define symbolic variables

x, y = sp.symbols('x y')

# Create a symbolic expression

expr = x**2 + y**2

# Create a function from the expression

f = sp.lambdify((x, y), expr, 'numpy')

# Evaluate the function at x=1, y=2

result = f(1, 2)

print(result)
```

In [ ]:
```
"""
Using lambdify from sympy convert the following function to a numpy function
f(x) = x^2 + 10sin(x)
```

```
@TODO: Convert the function
"""
```

Matplotlib

Matplotlib is a Python library that is used for creating visualizations. It provides support for creating line plots, scatter plots, bar plots, histograms, and more. Matplotlib is widely used in data visualization, scientific computing, and machine learning.

- How to create line plots in `matplotlib`
- How to graph 2D functions in `matplotlib`
- How to create contour plots in `matplotlib`
- How to create surface plots in `matplotlib`

How to create line plots in `matplotlib`

To create a line plot in `matplotlib`, you can use the `plt.plot()` function. You can pass the x and y values to the function to create a line plot. Here's an example:

```
import matplotlib.pyplot as plt

# Create x and y values

x = [1, 2, 3, 4, 5]
y = [1, 4, 9, 16, 25]

# Create a line plot

plt.plot(x, y)
```

In [ ]:
```
"""
Create a line plot of the following arrays
x = np.linspace(0, 10, 100)
y = np.sin(x)
@TODO: Create the plot
"""
```

How to graph 2D functions in `matplotlib`

To create a line plot in `matplotlib`, you can use the `plt.plot()` function. You can pass the x and y values to the function to create a line plot. Here's an example:

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

# Define the symbolic variables
```

```
x = sp.symbols('x')

# Create a symbolic expression

expr = x**2

# Create a function from the expression

f = sp.lambdify(x, expr, 'numpy')

# Generate x values

x_values = np.linspace(-10, 10, 100)

# Generate y values

y_values = f(x_values)

# Create a line plot

plt.plot(x_values, y_values)

plt.show()
```

In [ ]:
```
"""
Plot the following function
f(x) = x^2 + 10sin(x)
@TODO: Create the plot
"""
```

How to create contour plots in `matplotlib`

To create a contour plot in `matplotlib`, you can use the `plt.contour()` function. You can pass the x and y values, as well as the z values, to the function to create a contour plot. Here's an example:

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

# Create x and y values

x = np.linspace(-10, 10, 100)
y = np.linspace(-10, 10, 100)

# Create a meshgrid

X, Y = np.meshgrid(x, y)

# Create z values

Z = X**2 + Y**2
```

```
# Create a contour plot

plt.contour(X, Y, Z)
```

In [ ]:
```
"""
Create a contour plot of the following function:
f(x,y) = x^2 + y^2
@TODO: Create the plot
"""
```

How to create surface plots in `matplotlib`

To create a surface plot in `matplotlib`, you can use the `plt.plot_surface()` function. You can pass the x, y, and z values to the function to create a surface plot. Here's an example:

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

# Create x and y values

x = np.linspace(-10, 10, 100)
y = np.linspace(-10, 10, 100)

# Create a meshgrid

X, Y = np.meshgrid(x, y)

# Create z values

Z = X**2 + Y**2

# Create a surface plot

ax = plt.axes(projection='3d')

ax.plot_surface(X, Y, Z)
```

In [ ]:
```
"""
Create a 3D plot of the following function:
f(x,y) = x^2 + y^2
@TODO: Create the plot
"""
```

Let's get started!

Now that we have the algorithms defined, let's implement the Newton's method for 1D and 2D functions using Python.

Problem 1: 1D Newton's Method

Plot the minimun and maximum values of the funciton:

$$y = 3x^3 - 10x^2 + 56x + 50$$

The range of values for x is $[-6, 6]$.

In [ ]:
```
"""
@TODO: Implement the 1D Newton's method and find
the min and of the previous function
"""
```

Problem 2: 1D Newton's Method

Design a script that shows and finds all the minimum and maximum values of the function using newton's method.

$$y = x^5 - 8x^3 + 10x + 6$$

The range of values for x is $[-3, 3]$.

In [ ]:
```
"""
@TODO: Create a script that using the 1D Newton's method
finds all the min and max of the previous function
within a given range
"""
```

Problem 3: 2D Newton's Method

Create a algorithm based on the Newton's 2D method to find the minimum value of the function:

$$z = (x - 1)^2 + 100(y - x^2)^2$$

The range of values for x and y is $y = [-10, 10]$ and $x = [-6, 6]$.

Note: its recommended to set

$$x_0 = [2, 11]$$

In [ ]:
```
"""
@TODO: Create a script that using the 2D Newton's method
finds the min of the previous function
"""
```