Optimization

Optimization problems

Optimization is a field of mathematics that focuses on the problem of finding the solution to a minimization problem. More precisely, given a function $f : \mathbb{R}^k \to \mathbb{R}$, we seek $x^* \in \mathcal{C} \subset \mathbb{R}^k$ such that

$$f(x^*) \leqslant f(x) \text{ for all } x \in \mathcal{C}.$$

The set $\mathcal{C}$ constraints the solution to leave in a subset of $\mathbb{R}^k$

When $\mathcal{C}$ coincides with $\mathbb{R}^k$ the problem is said to be **unconstrained**.

In unconstrained optimization, the objective function $f(x)$ needs to be minimized (or maximized) without any restrictions on the variable x. The problem is described as

$$\min_x f(x), \quad x \in \mathbb{R}^k.$$

Many problems involve constraints that the solution must satisfy, that is, $\mathcal{C}$ does not coincide with $\mathbb{R}^k$. For instance, we want to minimize the function over a space where $x_j < c_j$, $c_j \in \mathbb{R}$, $j = 1, \ldots, k$ or we may be interested in values of x that minimizes f when certain restrictions are satisfied. Generally, the constrained optimization is

$$\begin{aligned} &\min_x \ f(x) \\ &\text{subject to} \\ &g_i(x) \leq 0, \quad i = 1, \ldots, m, \\ &h_j(x) = 0, \quad j = 1, \ldots, p. \end{aligned}$$

Here, $g_i(x)$ and $h_j(x)$ are functions that represent inequality and equality constraints.

In what follows, we will focus on the class of unconstrained problems where f is smooth, that is, a function that everywhere continuously differentiable.

Unconstrained problems

$$\min_x f(x), \quad x \in \mathbb{R}^k. \tag{1}$$

A point x^* is a **global** minimizer of f if $f(x^*) \leqslant f(x)$ for all x ranging over of of $\mathbb{R}^k$.

The global minimizer can be difficult to find. The algorithms to solve Equation 1 exploit local knowledge of f, we do not have a clear picture of the overall shape of f and, as such, we cannot ever be sure that the solution we find is indeed a global solution to the minimization problem. Most algorithms are able to find only a *local* minimizer, which is a point that achieves the samllest value of f in a neighborhood of x.

A point x^* is a **local** minimizer if there is a neighborhood[1] $\mathcal{N}$ of x^* such that $f(x^*) \leqslant f(x)$ for all $x \in \mathcal{N}$.

A point x^* is a **strict local** minimizer if there a neighborhood of $\mathcal{N}$ such that $f(x^*) < f(x)$ for all $x \in \mathcal{N}$.

For instance, for the function $f(x) = 2024$, every point is a weak local minimizer, while the function $f(x) = (x-a)^2$ has a strict local minimizer at $x = a^{-1}$.

Notation: Given a function $f : \mathbb{R}^k \to \mathbb{R}$, the *gradient* of f evaluated at x^o is:

$$\nabla f(x^o) := \begin{pmatrix} \left.\frac{\partial f(x)}{\partial x_1}\right|_{x=x^o} \\ \left.\frac{\partial f(x)}{\partial x_2}\right|_{x=x^o} \\ \vdots \\ \left.\frac{\partial f(x)}{\partial x_k}\right|_{x=x^o} \end{pmatrix}.$$

Similarly, the $k \times k$ *hessian* matrix of f evaluated at x^o is

$$\nabla^2 f(x^o) = \begin{pmatrix} \left.\frac{\partial^2 f(x)}{\partial x_1 \partial x_1}\right|_{x=x^o} & \left.\frac{\partial^2 f(x)}{\partial x_1 \partial x_2}\right|_{x=x^o} & \cdots & \left.\frac{\partial^2 f(x)}{\partial x_1 \partial x_k}\right|_{x=x^o} \\ \left.\frac{\partial^2 f(x)}{\partial x_2 \partial x_1}\right|_{x=x^o} & \left.\frac{\partial^2 f(x)}{\partial x_2 \partial x_2}\right|_{x=x^o} & \cdots & \left.\frac{\partial^2 f(x)}{\partial x_2 \partial x_k}\right|_{x=x^o} \\ \vdots & \vdots & & \vdots \\ \left.\frac{\partial^2 f(x)}{\partial x_k \partial x_1}\right|_{x=x^o} & \left.\frac{\partial^2 f(x)}{\partial x_k \partial x_2}\right|_{x=x^o} & \cdots & \left.\frac{\partial^2 f(x)}{\partial x_k \partial x_k}\right|_{x=x^o} \end{pmatrix}.$$

Fundamental mathematical tools

The necessary conditions for optimality are derived assuming that x^* is a local minimizer and the proving facts about $\nabla f(x^*)$ and $\nabla^2 f(x^*)$.

Theorem 1 If x^* is a local optimizer and f is continuously differentiable in an open neighborhood of x^*, then $\nabla f(x^*) = 0$ (first-order necessary condition); if $\nabla^2 f$ exists and its continuous in an open neighborhood of x^*, then $\nabla^2 f(x^*)$ is positive definite (second-order necessary conditions).[2]

Sufficient conditions for optimality are conditions on the derivatives of f at the point x^* that guarantee that x^* is a local minimizer.

Theorem 2 Suppose that $\nabla^2 f(x^*)$ is continuous in an open neighborhood of x^* and that $\nabla f(x^*) = 0$ and $\nabla^2 f(x^*)$ is positive definite. Then x^* is a strict local minimizer of f.

The second-order sufficient conditions guarantee something stronger than the necessary conditions; namely, that the minimizer is a strict local minimizer. Note too that the second-order sufficient conditions are not necessary: a point x^* may be a strict local minimizer, and yet may fail to satisfy the sufficient conditions. A simple example is given by the function $f(x) = x^3$, for which the point $x^* = 0$ is a strict local minimizer at which the Hessian matrix vanishes (and is therefore not positive definite).

When the objective function is convex, local and global minimizers are simple to characterize.

Theorem 3 When f is convex, any local minimizer of x^* is a global minimizer of f. If in addition, f is differentiable, any stationary point x^* is a global minimizer.

Example 1 The sum of square residuals

$$SSR(\beta) = \sum_{i=1}^{n} (Y_i - X_i'\beta)^2$$

is strictly convex provided that $\sum_{i=1}^{n} X_i X_i'$ is invertible (that is, it has full column rank).

Overview of algorithms

A common approach to optimization is to incrementally improve a point x by taking a step that minimizes the objective value based on a local model. The local model may be obtained, for example, from a first- or second-order Taylor approximation.

Optimization algorithms that follow this general approach are referred to as descent direction methods. They start with a design point $x^{(0)}$ and then generate a sequence of points, sometimes called iterates, to converge to a local minimum.

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

# Define the function and its derivative
def f(x):
    return x**2 - np.log(x)

def df(x):
    return 2*x - 1/x

def connectpoints(x,y,p1,p2):
    x1, x2 = x[p1], x[p2]
    y1, y2 = y[p1], y[p2]
    plt.plot([x1,x2],[y1,y2],'k-')


# Initial point
x0 = 2
alpha = 0.3

# Gradient descent update
x1 = x0 - alpha * df(x0)
x2 = x1 - alpha * df(x1)
# Points for the function plot
x = np.linspace(-2.5, 2.5, 400)
y = f(x)

# Tangent line at x0 (y = m*x + b)

# Creating the plot

plt.figure(figsize=(8, 5))
plt.plot(x, y, label='f(x) = x^2')
plt.scatter([x0, x1], [f(x0), f(x1)], color='red')  # Points
m = df(x0)
b = f(x0) - m*x0
tangent_line = m*x + b
plt.plot(x, tangent_line, 'b--', label=f'Tangent at x0={x0}')
plt.arrow(x0, 0.4, x1-x0, 0.0, head_width=0.1, length_includes_head=True, colo

m = df(x1)
b = f(x1) - m*x1
tangent_line = m*x + b
plt.plot(x, tangent_line, 'b--', label=f'Tangent at x0={x1}', )
plt.ylim([-0.2,6])
plt.xlim([-0.4,3])
plt.plot(x, tangent_line, 'b--', label=f'Tangent at x0={x1}', )
m = df(x0)
b = f(x0) - m*x0
tangent_line = m*x + b

plt.scatter(x0, f(x0), color='green')  # Initial point
plt.scatter(x1, f(x1), color='green')  # Next point after step
plt.scatter(x2, f(x2), color='green')  # Next point after step

plt.arrow(x1, 0., x2-x1, 0., head_width=0.1, length_includes_head=True, color =

plt.title('Gradient Descent on f(x)')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.xticks([])  # Remove x-axis ticks
plt.xticks([x0, x1, x2], [r"$x^{(0)}$", r"$x^{(1)}$", r"$x^{(2)}$"])
#plt.legend()
plt.grid(True)
plt.plot([x0, x0],[f(x0), 0],'g--')
plt.plot([x1, x1],[f(x1), 0],'g--')
plt.plot([x2, x2],[f(x1), 0],'g--')
plt.show()
```

```
/var/folders/72/d3_1t_ps0z1fw0_l58cwv_s80000gn/T/ipykernel_1891/189701708.py
:6: RuntimeWarning: invalid value encountered in log
  return x**2 - np.log(x)
```

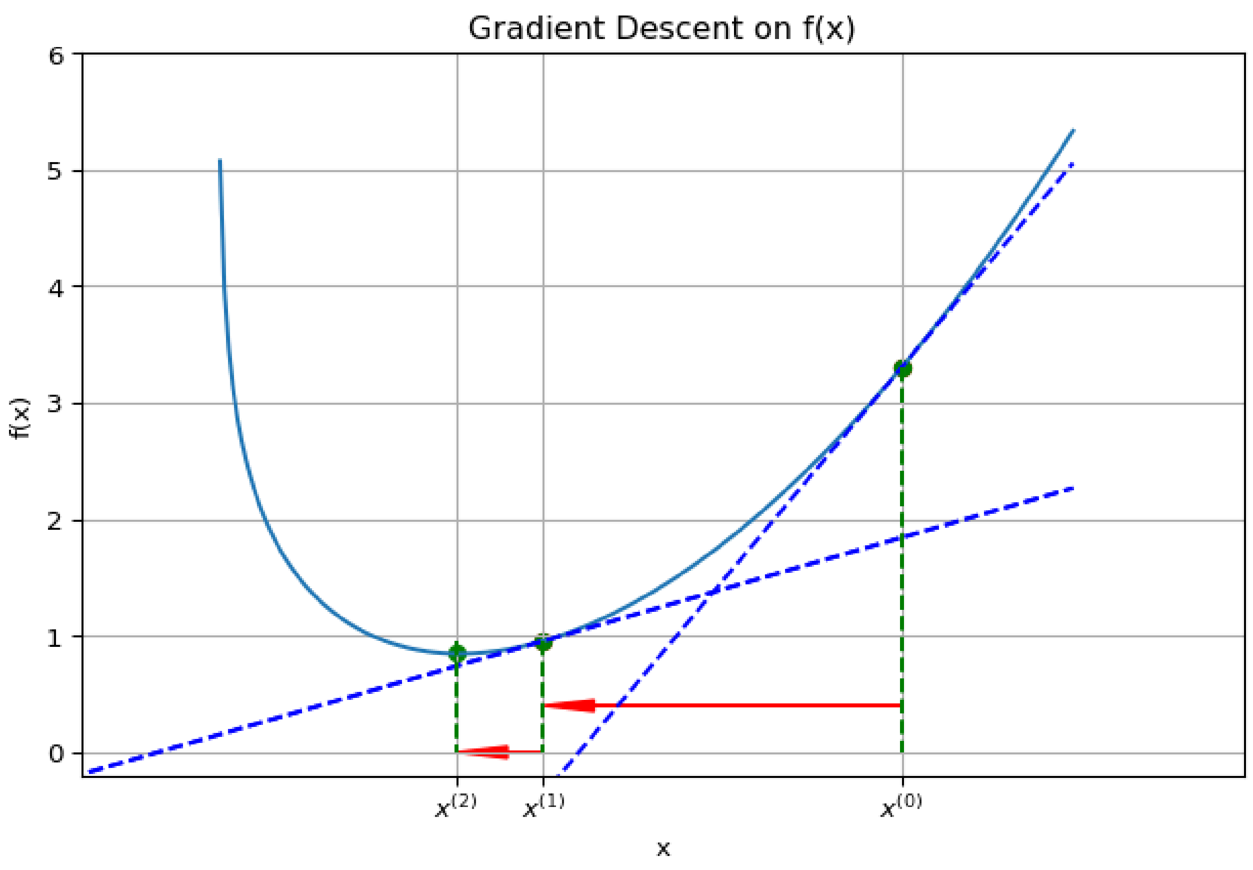

Figure 1: Gradient descent

The intuition behind this approach is relatively simple. Consider the function plotted in Figure 1. We start the algorithm at $x^{(0)}$. The derivative at this point (the tangent line in blue) is positive, that is, $f'(x^{(0)}) > 0$. Thus, to *descend* toward smaller values of the function have to set a point $x^{(1)}$ smaller. One possibility is to use the following iteration

$$x^{(1)} = x^{(0)} - \alpha \nabla f(x^{(0)}),$$

where $\alpha \in (0, 1)$ is the step factor. As it can be seen from Figure 1, at this point the value of f is now lower. Applying a new iterate we obtain $x^{(2)} = x^{(1)} - \alpha f'(x^{(1)})$ which is now very close to the minimum value of the function.

We will keep iterating until the *termination condition* is satisfied. The termination condition will be satisfied when the current iterate is likely to be a local minimum. [^termination]

[3]: The most common termination conditions are 1. *Maximum iterations*. We may want to terminate when the number of iterations k exceeds some threshold k_max. Alternatively, we might want to terminate once a maximum amount of elapsed time is exceeded. 2. *Absolute improvement*. This termination condition looks at the change in the function value over subsequent steps. If the change is smaller than a given

threshold, it will terminate:

$$f(x^{(k)}) - f(x^{(r+1)}) < \epsilon_a$$

1. *Relative improvement*. This termination condition looks at the change in function value but uses the relative change of the function. The iterations will terminate if

$$f(x^{(k)}) - f(x^{(r+1)}) < \epsilon_r |f(x^{(k)})|$$

1. *Gradient magnitude*. We can terminate if the derivative is smaller than a certain tolerance

$$\|\delta f(x^{(k)})\| < \epsilon_g$$

The same logic can be applied to the case in which $f : \mathbb{R}^k \to \mathbb{R}$ the only difference is that now $\nabla f(x^{(0)})$ is a $k \times 1$ vector instead of being a scalar.

The gradient descent idea can be generalized by considering the following iterate

$$x^{(r)} = x^{(r-1)} - \alpha d^{(r)},$$

where $d^{(r)}$ is a descent direction. The idea of this generalization is that instead of using the gradient as direction, we can use different directions that might speed up the algorithm.

When $d^{(r)} = \nabla f(x^{(r)})$, the algorithm is called the gradient descent (and direction is called the direction of deepest descent).

Directions that can be used belong to two classes: 1. *first-order*: the direction only uses information about the gradient of f. The many variations of the gradient descent (*Ada*, *Momentum*, etc.) and the conjugate gradient method belong to this class. 2. *second order*: the direction incorporate information about the second derivatives of f. The key idea here is that

$$f(x^{(r+1)}) = f(x^{(r)}) + \nabla f'(x^{(r)})(x^{(r+1)} - x^{(r)}) + (x^{(r+1)} - x^{(r)})\dot{H}_k(x^{(r+1)} - x^{(r)}),$$

where $\dot{H}_k = \nabla^2 f(\dot{x}^{(r)})$ is the Hessian evaluated at some point between $x^{(r+1)}$ and $x^{(r)}$. Then, approximating the gradient of f and setting it equal to zero yields

$$x^{(r+1)} = x^{(r)} - [\nabla^2 f(x^{(r)})]^{-1} \nabla f(x^{(r)}) +$$

which suggests using the inverse of the hessian to form the direction. Since evaluating the Hessian at each iterate is too costly computationally, different algorithms approximate the Hessian in different ways.

A simple implementation of gradient descent

The following code implements a gradient descent with steepest direction in Python.

```python
from numpy import linalg as la

def steepest_descent(f, gradient, initial_guess, learning_rate, num_iterations
    x = initial_guess
    for i in range(num_iterations):
        grad = gradient(x)
        x = x - learning_rate * grad
        normg = la.norm(grad)
        print(f"Iteration {i+1}: x = {x}, f(x) = {f(x)}, ||g(x)||={normg}")
        ## Termination condition
        if  normg < epsilon_g:
            break
    return x
```

This is the Julia version.

```julia
using LinearAlgebra ## needed for norm
function steepest_descent(f, gradient, initial_guess, learning_rate;
num_iterations = 100; epsilon_g = 1e-7)
    x = initial_guess
    for i in 1:num_iterations
        grad = gradient(x)
        x = x - learning_rate * grad
        normg = norm(g)
        println("Iteration $i: x = $x, f(x) =
$(objective_function(x)), ||g(x)||=$(normg)")
        if normg < epsilon_g
            break
        end
    end
    return x
end
```

Suppose we want to solve the following problem

$$\min_x f(x)$$

where, for some $d > 1$,

$$f(x) = \sum_{i=1}^{d} \left((x_i - 3)^2\right)$$

The gradient of this function is

$$\nabla f(x) = \begin{pmatrix} 2(x_1 - 3) \\ 2(x_2 - 3) \\ \vdots \\ 2(x_d - 3) \end{pmatrix}.$$

The minimum of this function is $x_i = 3$.

```
def f(x):
    return np.sum((x-3.0)**2)

def gradient(x):
    return 2*(x-3.0)

steepest_descent(f, gradient, np.array([0., 0.]), 0.2)
```

```
Iteration 1: x = [1.2 1.2], f(x) = 6.479999999999999,
||g(x)||=8.48528137423857
Iteration 2: x = [1.92 1.92], f(x) = 2.3327999999999993,
||g(x)||=5.091168824543142
Iteration 3: x = [2.352 2.352], f(x) = 0.8398079999999992,
||g(x)||=3.0547012947258847
Iteration 4: x = [2.6112 2.6112], f(x) = 0.3023308799999997,
||g(x)||=1.8328207768355302
Iteration 5: x = [2.76672 2.76672], f(x) = 0.10883911679999973,
||g(x)||=1.099692466101318
Iteration 6: x = [2.860032 2.860032], f(x) = 0.039182082047999806,
||g(x)||=0.6598154796607905
Iteration 7: x = [2.9160192 2.9160192], f(x) = 0.014105549537279988,
||g(x)||=0.39588928779647375
Iteration 8: x = [2.94961152 2.94961152], f(x) = 0.005077997833420814,
||g(x)||=0.23753357267788475
Iteration 9: x = [2.96976691 2.96976691], f(x) = 0.0018280792200315037,
||g(x)||=0.1425201436067311
Iteration 10: x = [2.98186015 2.98186015], f(x) = 0.0006581085192113414,
||g(x)||=0.08551208616403891
Iteration 11: x = [2.98911609 2.98911609], f(x) = 0.00023691906691608675,
||g(x)||=0.051307251698423345
Iteration 12: x = [2.99346965 2.99346965], f(x) = 8.529086408979587e-05,
||g(x)||=0.03078435101905426
```

```
Iteration 13: x = [2.99608179 2.99608179], f(x) = 3.070471107232651e-05,
||g(x)||=0.01847061061143306
Iteration 14: x = [2.99764908 2.99764908], f(x) = 1.1053695986035874e-05,
||g(x)||=0.011082366366859834
Iteration 15: x = [2.99858945 2.99858945], f(x) = 3.9793305549719125e-06,
||g(x)||=0.0066494198201153985
Iteration 16: x = [2.99915367 2.99915367], f(x) = 1.4325589997895879e-06,
||g(x)||=0.003989651892068737
Iteration 17: x = [2.9994922 2.9994922], f(x) = 5.15721239924432e-07,
||g(x)||=0.002393791135240991
Iteration 18: x = [2.99969532 2.99969532], f(x) = 1.8565964637257903e-07,
||g(x)||=0.0014362746811448456
Iteration 19: x = [2.99981719 2.99981719], f(x) = 6.683747269425835e-08,
||g(x)||=0.000861764808686405
Iteration 20: x = [2.99989032 2.99989032], f(x) = 2.4061490169894037e-08,
||g(x)||=0.0005170588852123454
Iteration 21: x = [2.99993419 2.99993419], f(x) = 8.662136461115092e-09,
||g(x)||=0.00031023533112715604
Iteration 22: x = [2.99996051 2.99996051], f(x) = 3.118369125973376e-09,
||g(x)||=0.0001861411986757912
Iteration 23: x = [2.99997631 2.99997631], f(x) = 1.1226128853672496e-09,
||g(x)||=0.00011168471920497228
Iteration 24: x = [2.99998578 2.99998578], f(x) = 4.0414063872210937e-10,
||g(x)||=6.70108315234858e-05
Iteration 25: x = [2.99999147 2.99999147], f(x) = 1.454906299338991e-10,
||g(x)||=4.020649891358905e-05
Iteration 26: x = [2.99999488 2.99999488], f(x) = 5.237662677983984e-11,
||g(x)||=2.4123899347651e-05
Iteration 27: x = [2.99999693 2.99999693], f(x) = 1.8855585639651492e-11,
||g(x)||=1.447433960909303e-05
Iteration 28: x = [2.99999816 2.99999816], f(x) = 6.788010829620027e-12,
||g(x)||=8.684603765204602e-06
Iteration 29: x = [2.99999889 2.99999889], f(x) = 2.4436838986632097e-12,
||g(x)||=5.210762258871547e-06
Iteration 30: x = [2.99999934 2.99999934], f(x) = 8.797262039900029e-13,
||g(x)||=3.1264573553229284e-06
Iteration 31: x = [2.9999996 2.9999996], f(x) = 3.167014331536526e-13,
||g(x)||=1.8758744136961865e-06
Iteration 32: x = [2.99999976 2.99999976], f(x) = 1.1401251593531493e-13,
||g(x)||=1.1255246477152823e-06
Iteration 33: x = [2.99999986 2.99999986], f(x) = 4.10445057876081e-14,
||g(x)||=6.753147886291694e-07
Iteration 34: x = [2.99999991 2.99999991], f(x) = 1.477602202246525e-14,
||g(x)||=4.0518887342871644e-07
Iteration 35: x = [2.99999995 2.99999995], f(x) = 5.31936792808749e-15,
||g(x)||=2.431133235548003e-07
Iteration 36: x = [2.99999997 2.99999997], f(x) = 1.914972432124978e-15,
||g(x)||=1.4586799413288017e-07
Iteration 37: x = [2.99999998 2.99999998], f(x) = 6.893900623730809e-16,
||g(x)||=8.752079597729852e-08
```

```
array([2.99999998, 2.99999998])
```

```
function f(x)
    sum((x.-3.0).^2)
end

function gradient(x)
    2.*(x.-3.0)
end

steepest_descent(f, gradient, [.0, .0], 0.2)
```

Calculating the derivative

When the gradient is difficult to calculate analytically, we can use algorithmically calculate the derivative of the function f.

- *Finite difference*: We use

$$\lim_{h} \frac{f(x+h)-f(x)}{h}$$

```
using FiniteDifferences

# Create a central finite difference method with the default settings
fdm = central_fdm(5, 1)

# Calculate the gradient at a point
x0 = [1.0, 2.0]
gradient = grad(fdm, f, x0)

println("Numerical Gradient:", gradient)
```

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



epsilon = np.sqrt(np.finfo(float).eps)

# Point at which to calculate the gradient
x0 = np.array([1.0, 2.0])

# Calculate the gradient at the point x0
gradient = approx_fprime(x0, f, epsilon)
```

```
print("Gradient at x0:", gradient)
```

```
Gradient at x0: [-4. -2.]
```

- *Automatic differentiation*

Automatic Differentiation (AD) is a computational technique used to evaluate the derivative of a function specified by a computer program. AD exploits the fact that any computer program, no matter how complex, executes a sequence of elementary arithmetic operations and functions (like additions, multiplications, and trigonometric functions). By applying the chain rule to these operations, AD efficiently computes derivatives of arbitrary order, which are accurate up to machine precision. This contrasts with numerical differentiation, which can introduce significant rounding errors.

One popular Python library that implements automatic differentiation is `autograd`. It extends the capabilities of NumPy by allowing you to automatically compute derivatives of functions composed of many standard mathematical operations. Here is a simple example of using `autograd` to compute the derivative of a function:

Now, let's compute the derivative of the function defined above.

```
import autograd.numpy as np   # Import wrapped NumPy
from autograd import grad    # Import the gradient function

# Create a function that returns the derivative of f
df = grad(f)

# Evaluate the derivative at x = pi
print("The derivative of f(x) at x = [0.2, 0.1] is:", df(np.array([0.2, 0.1])))
```

```
The derivative of f(x) at x = [0.2, 0.1] is: [-5.6 -5.8]
```

When comparing the precision and applicability of finite difference methods and automatic differentiation (AD), especially for functions with large input dimensions, there are several key points to consider.

Finite difference methods approximate derivatives by evaluating differences in function values at nearby points. The accuracy of these methods is highly dependent on the step size chosen: too large, and the approximation becomes poor; too small, and floating-point errors can dominate the result. This trade-off can be particularly challenging in high-dimensional spaces as errors can accumulate more significantly (each partial derivative may introduce errors that affect the overall gradient computation).

AD computes derivatives using the exact chain rule and is not based on numerical approximations of difference quotients. Therefore, it can provide derivatives that are accurate to machine precision. AD efficiently handles computations in high dimensions because it systematically applies elementary operations and chain rules, bypassing the curse of dimensionality that often affects finite difference methods. AD is less sensitive to the numerical issues that affect finite differences, such as choosing a step size or dealing with subtractive cancellation.

The visualization provided by `@fig-error` demonstrates the comparative performance of the finite difference method and automatic differentiation (AD) when used to calculate gradients. This graph illustrates a key observation: as the dimensionality of the input increases, the error associated with the finite difference method tends to rise almost linearly. This escalation in error is accompanied by an increase in computation time, underscoring the method's sensitivity to higher dimensions.

In contrast, the automatic differentiation method showcases remarkable stability across dimensions. The error remains negligible, essentially zero, highlighting AD's inherent precision. Furthermore, AD's computation time remains consistent regardless of input dimensionality. This performance characteristic of AD is due to its methodological approach, which systematically applies the chain rule to derive exact derivatives, bypassing the numerical instability and scaling issues often encountered with finite differences.

These insights are particularly relevant in fields that rely heavily on precise and efficient computation of derivatives, such as in numerical optimization, machine learning model training, and dynamic systems simulation. The stability and scalability of AD make it an invaluable tool in these areas, especially when dealing with high-dimensional data sets or complex models.

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

def multivariate_function(x):
    """ A sample multivariate function, sum of squares plus sin of each compone
    return np.sum(x**2 + np.sin(x))

def analytical_derivative(x):
    """ Analytical derivative of the multivariate function. """
    return 2*x + np.cos(x)

def finite_difference_derivative(f, x, h=1e-5):
    """ Compute the gradient of `f` at `x` using the central finite difference
    grad = np.zeros_like(x)
    for i in range(len(x)):
        x_plus = np.copy(x)
        x_minus = np.copy(x)
        x_plus[i] += h
        x_minus[i] -= h
        grad[i] = (f(x_plus) - f(x_minus)) / (2*h)
    return grad

# Range of dimensions
dimensions = range(1, 101)
errors = []
times = []

# Loop over dimensions
for dim in dimensions:
    x = np.random.randn(dim)

    # Analytical derivative
    true_grad = analytical_derivative(x)

    # Start timing
    start_time = time.time()

    # Finite difference derivative
    fd_grad = finite_difference_derivative(multivariate_function, x)

    # End timing
    elapsed_time = time.time() - start_time
    times.append(elapsed_time)

    # Error
    error = np.linalg.norm(fd_grad - true_grad)
    errors.append(error)

# Plotting error
plt.figure(figsize=(14, 6))

plt.subplot(1, 2, 1)
plt.plot(dimensions, errors, marker='o')
plt.xlabel('Number of Dimensions')
plt.ylabel('Error')
plt.title('Approximation Error by Dimension')
plt.grid(True)

# Plotting computational time
```

```
plt.subplot(1, 2, 2)
plt.plot(dimensions, times, marker='o')
plt.xlabel('Number of Dimensions')
plt.ylabel('Time (seconds)')
plt.title('Computational Time by Dimension')
plt.grid(True)

plt.tight_layout()
plt.show()
```

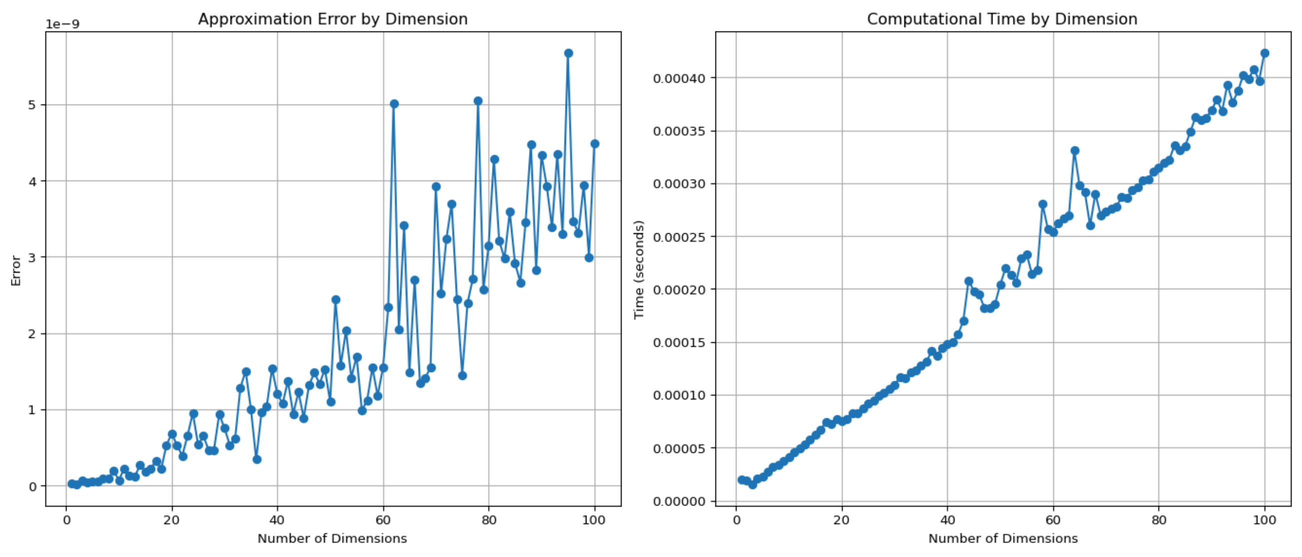

Figure 2

Using solver

It is almost always advantageous to utilize established minimization routines and libraries rather than crafting custom code from scratch. This approach not only saves time but also leverages the extensive testing and optimizations embedded within these libraries, which are designed to handle a broad range of mathematical challenges efficiently and accurately.

In Python, the `scipy.optimize` module from the SciPy library is a robust toolkit that provides several algorithms for function minimization, including methods for unconstrained and constrained optimization. Some of the notable algorithms include BFGS, Nelder-Mead, and conjugate gradient, among others. These algorithms are well-suited for numerical optimization in scientific computing.

In Julia, `Optim.jl` offers a similar breadth of optimization routines, with support for a variety of optimization problems, from simple univariate function minimization to complex multivariate cases. `Optim.jl` includes algorithms like L-BFGS, Gradient Descent, and Newton's Method, each tailored for specific types of optimization scenarios.

Both Python's SciPy and Julia's `Optim.jl` represent just a slice of the available tools. There are many other solvers and libraries dedicated to optimization. The most widely used is Ipopt which is particularly useful for problems that require embedding in lower-level systems for performance.

The choice of a solver can depend on several factors:

1. **Problem Type**: Some solvers are better suited for large-scale problems, others for problems with complex constraints, or for nondifferentiable or noisy functions.
2. **Accuracy and Precision**: Different algorithms and implementations can provide varying levels of precision and robustness, important in applications like aerospace or finance.
3. **Performance and Speed**: Depending on the implementation, some solvers might be optimized for speed using advanced techniques like parallel processing or tailored data structures.
4. **Ease of Use and Flexibility**: Some libraries offer more user-friendly interfaces and better documentation, which can be crucial for less experienced users or complex projects where customization is key.

Julia's `Optim.jl`

`Optim.jl` supports various optimization algorithms. For our simple example, we can use the BFGS method, which is suitable for smooth functions and is part of a family of quasi-Newton methods.

You can now call the `optimize` function from `Optim.jl`, specifying the function, an initial guess, and the optimization method. Here is how you do it:

```
# Initial guess
initial_guess = [0.0, 0.0]

# Perform the optimization
result = optimize(f, initial_guess, BFGS())
```

The `result` object contains all the information about the optimization process, including the optimal values found, the value of the function at the optimum, and the convergence status:

```
# Access the results
println("Optimal parameters: ", Optim.minimizer(result))
println("Minimum value: ", Optim.minimum(result))
println("Convergence information: ", result)
```

If your function is more complex or if you want to speed up the convergence for large-scale problems, you can also provide the gradient (and even the Hessian) to the optimizer. `Optim.jl` can utilize these for more efficient calculations.

For constrained optimization, `Optim.jl` has support for simple box constraints which can be set using the `Fminbox` method to wrap around other methods like BFGS.

```
# Define the bounds
lower_bounds = [0.5, 1.5]
upper_bounds = [1.5, 2.5]

# Initial guess within the bounds
initial_guess = [1.0, 2.0]

# Set up the optimizer with Fminbox
optimizer = Fminbox(BFGS())
# Run the optimization with bounds
result = optimize(f, lower_bounds, upper_bounds, initial_guess,
optimizer, Optim.Options(g_tol = 1e-6))
```

Python's `minimize`

The `scipy.optimize.minimize` function in Python is a versatile solver for minimization problems of both unconstrained and constrained functions. It provides a wide range of algorithms for different kinds of optimization problems.

SciPy's `minimize` function supports various methods like 'BFGS', 'Nelder-Mead', 'TNC', etc. The choice of method can depend on the nature of your problem (e.g., whether it has constraints, whether the function is differentiable, etc.). As in the Julia's discussion, we'll use 'BFGS'.

```python
import numpy as np
from scipy.optimize import minimize
initial_guess = np.array([0, 0])
result = minimize(f, initial_guess, method='BFGS')

print("Optimal parameters:", result.x)
print("Minimum value:", result.fun)
print("Success:", result.success)
print("Message:", result.message)
```

```
Optimal parameters: [2.99999999 2.99999999]
Minimum value: 2.4602836913395623e-16
```

```
Success: True
Message: Optimization terminated successfully.
```

To add some bounds to the variables we can use the Bounds module.

```python
from scipy.optimize import Bounds

# Define bounds (0, +Inf) for all parameters
bounds = Bounds(np.array([0., 0.]), [np.inf, np.inf])

# Run the optimization with bounds
result_with_bounds = minimize(f, initial_guess, method='L-BFGS-B', bounds=bounc
print("Optimal parameters:", result_with_bounds.x)
print("Minimum value:", result_with_bounds.fun)
print("Success:", result_with_bounds.success)
print("Message:", result_with_bounds.message)
```

```
Optimal parameters: [3.00000044 3.00000044]
Minimum value: 3.9332116120454443e-13
Success: True
Message: CONVERGENCE: NORM OF PROJECTED GRADIENT <= PGTOL
```

Assignment

$$Y_t = c + \phi_1 Y_{t-1} + \phi_2 Y_{t-2} + \varepsilon_t, \quad \varepsilon_t \, WN(0, \sigma^2).$$

Parameter Estimation - Estimate the parameters of the AR(2) model using both conditional and unconditional likelihood approaches on the **monthly log differences** of `INDPRO` from the `FRED-MD` (FRED data is here).

Tasks:

1. **Coding the Likelihood Function**

   - Implement the likelihood function for an AR(2) model in Python. You are required to code both the conditional and unconditional likelihood functions.
   - **Conditional Likelihood**: This approach uses the first 2 observations as given and starts the likelihood calculation from the 3rd observation.
   - **Unconditional Likelihood**: This approach integrates over $p(Y_1, Y_2)$, the unconditional distributions of the first two observations.

   Use the following model specification for the AR(7) process:

$$y_t = \phi_0 + \phi_1 y_{t-1} + \phi_2 y_{t-2} + \epsilon_t$$

where ϵ_t is i.i.d. normal with mean zero and variance σ^2.

2. **Maximizing the Likelihood**

   - Write Python or Julia code to maximize the likelihood functions (both conditional and unconditional) with respect to the parameters ϕ_0, ϕ_1, ϕ_2, and σ^2. You may consider using optimization routines available in libraries such as `scipy.optimize` or `Optim.jl`.

3. **Forecasting**

   - With the estimated parameters from both approaches, forecast the future values of the log differences of `INDPRO` for the next 8 months ($h = 1, 2, \ldots, 8$).
   - (Optional) Provide a brief comparison of the forecast accuracy from both sets of parameters based on out-of-sample forecasting. Discuss any notable differences and potential reasons for these differences.

Deliverables: - A Python (or Julia) script containing the implementations and results for the tasks outlined above. The script should run (you might give instructions on what is needed to make it run). Ideally, a Jupyter notebook. - A report discussing the methodology and the results.

Assessment Criteria: - Correctness of the likelihood function implementations. - Quality of the code, including readability and proper documentation.

Resources: - `FRED-MD` dataset: Access the dataset directly from the FRED website or through any API that provides access to it. - `SciPy` library documentation for optimization functions. - `Optim.jl`

This assignment will test your ability to implement statistical models, manipulate time-series data, perform parameter estimation, and use statistical methods to forecast future values.

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

def ar_likelihood(params, data, p):
    """
    Calculate the negative (unconditional) log likelihood for an AR(p) model.

    params: list of parameters, where the first p are AR coefficients and the l
    data: observed data.
    p: order of the AR model.
    """
    # Extract AR coefficients and noise variance
    c = params[0]
    phi = params[1:p+1]
    sigma2 = params[-1]

    # Calculate residuals
    T = len(data)
    residuals = data[p:] - c - np.dot(np.column_stack([data[p-j-1:T-j-1] for j

    # Calculate negative log likelihood
    log_likelihood = (-T/2 * np.log(2 * np.pi * sigma2) - np.sum(residuals**2)

    return -log_likelihood

def estimate_ar_parameters(data, p):
    """
    Estimate AR model parameters using maximum likelihood estimation.

    data: observed data.
    p: order of the AR model.
    """
    # Initial parameter guess (random AR coefficients, variance of 1)
    params_initial = np.zeros(p+2)
    params_initial[-1] = 1.0

    ## Bounds
    bounds = [(None, None)]
    # Then p AR coefficients, each bounded between -1 and 1
    bounds += [(-1, 1) for _ in range(p)]
    # The variance parameter, bounded to be positive
    bounds += [(1e-6, None)]

    # Minimize the negative log likelihood
    result = minimize(ar_likelihood, params_initial, args=(data, p), bounds=bou

    if result.success:
        estimated_params = result.x
        return estimated_params
    else:
        raise Exception("Optimization failed:", result.message)

# Example usage
data = np.random.randn(100)  # Simulated data; replace with actual data
p = 2  # AR(2) model
params = estimate_ar_parameters(data, p)
print("Estimated parameters:", params)
```

```
Estimated parameters: [ 0.11376649 -0.10794687  0.07171477  0.80931585]
```

```python
import numpy as np

def fit_ar_ols_xx(data, p):
    """
    data: observed data.
    p: order of the AR model.
    note: no constant
    """
    # Prepare the lagged data matrix
    T = len(data)
    Y = data[p:]  # Dependent variable (from p to end)
    X = np.column_stack([data[p-i-1:T-i-1] for i in range(p)])
    X = np.column_stack((np.ones(X.shape[0]), X))

    # Calculate OLS estimates using the formula: beta = (X'X)^-1 X'Y
    XTX = np.dot(X.T, X)  # X'X
    XTY = np.dot(X.T, Y)  # X'Y
    beta_hat = np.linalg.solve(XTX, XTY)  # Solve (X'X)beta = X'Y

    return beta_hat


beta_hat = fit_ar_ols_xx(data, p)
print("Estimated AR coefficients:", beta_hat)
```

```
Estimated AR coefficients: [ 0.11376621 -0.10794739  0.0717144 ]
```

```python
import numpy as np
import statsmodels.api as sm

def fit_ar_ols_sm(data, p):
    """
    Estimate parameters of an AR(p) model using OLS.

    data: observed data.
    p: order of the AR model.
    """
    # Prepare the lagged data matrix
    T = len(data)
    Y = data[p:]  # Dependent variable (from p to end)
    X = np.column_stack([data[p-i-1:T-i-1] for i in range(p)])
    X = np.column_stack((np.ones(X.shape[0]), X))
    # Fit the OLS model
    model = sm.OLS(Y, X)
    results = model.fit()
    return results

# Example usage
```

```
  # Simulated data; replace with actual data
p = 2  # AR(2) model
results = fit_ar_ols_sm(data, p)

# Print the results
print(results.summary())
```

```
                            OLS Regression Results
==============================================================================
==
Dep. Variable:                      y   R-squared:
0.019
Model:                            OLS   Adj. R-squared:
-0.002
Method:                 Least Squares   F-statistic:
0.9079
Date:                Fri, 07 Mar 2025   Prob (F-statistic):
0.407
Time:                        11:30:19   Log-Likelihood:
-129.68
No. Observations:                  98   AIC:
265.4
Df Residuals:                      95   BIC:
273.1
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
==
                 coef    std err          t      P>|t|      [0.025
0.975]
------------------------------------------------------------------------------
--
const          0.1138      0.095      1.196      0.235      -0.075
0.303
x1            -0.1079      0.102     -1.056      0.293      -0.311
0.095
x2             0.0717      0.103      0.694      0.489      -0.133
0.277
==============================================================================
==
Omnibus:                        1.946   Durbin-Watson:
1.995
Prob(Omnibus):                  0.378   Jarque-Bera (JB):
1.923
Skew:                          -0.329   Prob(JB):
0.382
Kurtosis:                       2.806   Cond. No.
1.28
==============================================================================
==

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is
correctly specified.
```

Julia code

```
using Distributions
using Optim
using LinearAlgebra

function ar_likelihood(params, data, p)
    """
    Calculate the negative (unconditional) log likelihood for an AR(p) model.

    params: list of parameters, where the first p+1 are AR coefficients and the
    data: observed data.
    p: order of the AR model.
    """
    # Extract AR coefficients and noise variance
    c = params[1]
    phi = params[2:p+1]
    sigma2 = params[p+2]

    # Calculate residuals
    T = length(data)
    Y = data[p+1:end]
    X = hcat([data[p-j:T-j-1] for j in 0:p-1]...)
    residuals = Y .- c .- X * phi

    # Calculate negative log likelihood
    log_likelihood = (-T/2 * log(2 * π * sigma2) - sum(residuals.^2) / (2 * sig

    return -log_likelihood
end

function estimate_ar_parameters(data, p)
    """
    Estimate AR model parameters using maximum likelihood estimation.

    data: observed data.
    p: order of the AR model.
    """
    # Initial parameter guess (zeros AR coefficients, variance of 1)
    params_initial = zeros(p+2)
    params_initial[p+2] = 1.0  # Initial variance
```

```julia
    # Setup lower and upper bounds
    lower_bounds = vcat(-Inf, fill(-1.0, p), 1e-6)
    upper_bounds = vcat(+Inf, fill(1.0, p), Inf)

    # Minimize the negative log likelihood
    result = optimize(params -> ar_likelihood(params, data, p),
                     lower_bounds, upper_bounds, params_initial,
                     Fminbox(LBFGS()))

    if Optim.converged(result)
        estimated_params = Optim.minimizer(result)
        return estimated_params
    else
        error("Optimization failed")
    end
end

# Example usage
data = randn(100)  # Simulated data; replace with actual data
p = 2  # AR(2) model
params = estimate_ar_parameters(data, p)
println("Estimated parameters: ", params)

function fit_ar_ols_xx(data, p)
    """
    data: observed data.
    p: order of the AR model.
    note: no constant
    """
    # Prepare the lagged data matrix
    T = length(data)
    Y = data[p+1:end]  # Dependent variable (from p+1 to end)
    X = hcat([data[p-j:T-j-1] for j in 0:p-1]...)
    X = [ones(T-p) X]  # Add a constant to the model
    # Calculate OLS estimates using the formula: beta = (X'X)^-1 X'Y
    XTX = X' * X  # X'X
    XTY = X' * Y  # X'Y
    beta_hat = XTX \ XTY  # Solve (X'X)beta = X'Y

    return beta_hat
end

beta_hat, X = fit_ar_ols_xx(data, p)
println("Estimated AR coefficients: ", beta_hat)

function fit_ar_ols_sm(data, p)
    """
    Estimate parameters of an AR(p) model using OLS.

    data: observed data.
    p: order of the AR model.
    """
    # Prepare the lagged data matrix
    T = length(data)
    Y = data[p+1:end]  # Dependent variable (from p+1 to end)
    X = hcat([data[p-j:T-j-1] for j in 0:p-1]...)
    # Add a constant to the model if you want an intercept (optional)
    X = hcat(ones(size(X, 1)), X)

    # Fit the OLS model using direct matrix calculation
    beta = (X'X) \ (X'Y)

    # Calculate residuals
    residuals = Y - X * beta

    # Calculate standard errors
    n, k = size(X)
    sigma2 = sum(residuals.^2) / (n - k)
    var_beta = sigma2 * inv(X'X)
    std_errors = sqrt.(diag(var_beta))

    # t-statistics
    t_stats = beta ./ std_errors

    # p-values (assuming normal distribution)
    p_values = 2 * (1 .- cdf.(Normal(), abs.(t_stats)))

    # Return a named tuple with results
    return (
        coefficients = beta,
        std_errors = std_errors,
        t_statistics = t_stats,
        p_values = p_values,
        residuals = residuals,
        sigma2 = sigma2
    )
end

# Example usage
results = fit_ar_ols_sm(data, p)

# Print summary
println("\nOLS Results Summary:")
println("====================")
println("Coefficients:")
for i in 1:p
    println("AR($i): $(results.coefficients[i+1]) (std err: $(results.std_error
end
println("Residual standard error: $(sqrt(results.sigma2))")
```

Footnotes

1. A neighborhood of a point $x \in \mathbb{R}$ is any open subset of $\mathbb{R}^k$ containing x. ↩
2. A square matrix H is positive definite if for every $z \in \mathbb{R}^k$ with $\backslash\mathrm{norm} z \neq 0$, $z'Hz > 0$. ↩
3. termanation ↩