# Proyecto de análisis numérico

Daniel Felipe Castro Moreno

28 de octubre del 2025

## 1. Introducción

Se aborda la estimación de cinco valores poblacionales faltantes (años 1996, 1998, 2008, 2011 y 2012) para un país asignado, a partir de la serie anual 1990–2023. El método elegido es la interpolación polinómica de Lagrange, empleando todos los años con dato disponible como nodos.

## 2. Elección del lenguaje

La solución se implementa en Python y se organiza en módulos para separar lectura/escritura, núcleo numérico y visualización. Entre sus ventajas sobre otros lenguajes de programación se enfatiza que Python ofrece una combinación de *claridad*, *rapidez de desarrollo* y *ecosistema científico* difícil de igualar en proyectos de análisis numérico y prototipado académico.

**Sintaxis legible y cercana al inglés.** Python prioriza la legibilidad: la indentación define bloques, se evita el "ruido" sintáctico y muchas construcciones (bucles, comprensiones de listas, manejo de contexto con `with`) se leen casi como pseudocódigo. Esta cercanía al inglés técnico reduce la curva de aprendizaje, facilita revisiones por pares y disminuye la probabilidad de errores sutiles. En comparación, C/C++ o Java requieren más *boilerplate* para tareas equivalentes; en MATLAB la sintaxis es concisa pero el lenguaje está más acotado al entorno numérico propietario.

**Ecosistema científico maduro.** La SciPy stack: NumPy, SciPy y Matplotlib (empleado para realizar la gráfica final); proporciona rutinas numéricas y visualización de alta calidad; se complementa con bibliotecas especializadas. Este ecosistema permite comenzar con una implementación directa y luego, si la escala lo exige, intercambiar módulos por alternativas más eficientes sin reescribir el proyecto completo.

**Rendimiento en la práctica.** Aunque Python interpretado es, en bruto, más lento que C/C++ o Rust, en *análisis numérico práctico* el rendimiento suele venir de:

a) **Vectorización** (NumPy): desplaza los bucles a código C altamente optimizado (BLAS/LAPACK).

b) **JIT** (`numba`): compila a código nativo las partes críticas (*hot paths*).

c) **Extensiones C/C++/Fortran** (`cython`, `cffi`, `ctypes`): núcleos en bajo nivel, pegamento en Python.

En este caso, el método de Lagrange directo tiene coste $O(mn^2)$ con $m$=5 años faltantes y $n$=29 puntos conocidos, lo que se resuelve en milisegundos en Python puro. Si el tamaño creciera (por ejemplo, cientos o miles de puntos), es trivial migrar a la forma *barycentric* (más estable y rápida por evaluación), vectorizar operaciones con NumPy o compilar el bucle doble con `numba`.

## 3. Fundamento teórico

Sean los puntos conocidos $\{(x_i, y_i)\}_{i=1}^{n}$ con $x_i$ distintos. El polinomio interpolante se define como

$$P_{n-1}(x) \;=\; \sum_{i=1}^{n} y_i\, \ell_i(x), \qquad \ell_i(x) \;=\; \prod_{\substack{j=1 \\ j\neq i}}^{n} \frac{x - x_j}{x_i - x_j}.$$

Para un año faltante $x^*$ se estima $P_{n-1}(x^*)$, utilizando *todos* los nodos disponibles.

## 4. Arquitectura y decisiones de implementación

Se adopta Python 3 por su claridad sintáctica y facilidad para estructurar el proyecto. La organización modular es:

- `data/population.txt`: archivo de entrada *UTF-8*, una línea por registro con el formato `<año><población>`. Los años cubren 1990–2023 en orden ascendente; se usa `-1` para indicar población desconocida.
- `output/interpolation_results.txt`: reporte compacto *sólo* de los años faltantes con su población aproximada (cabecera "INTERPOLATION RESULTS", columnas `Year` y `Population` con separador de miles, ordenado por año).
- `output/plot_data.txt`: datos completos 1990–2023 para graficación, con columnas: `Year`, `Population` (alineada y con separador de miles) y `Type {original|interpolated}`. Útil para auditoría y reproducción de la figura.
- `output/population_plot.png`: imagen de la serie Año–Población, con línea de tendencia y marcadores diferenciados (original vs. interpolado).
- `src/file_io.py`: funciones de E/S: `read_population_data()` (parseo robusto del `.txt` y marcado de faltantes), `write_plot_data()` (tabla alineada para gráficas) y `write_interpolation_results()` (salida resumida de estimaciones), con manejo de errores.
- `src/interpolation.py`: núcleo numérico con la implementación directa del polinomio de Lagrange.
- `src/visualization.py`: generación de la figura Año–Población con `matplotlib`: separación de puntos `original`/`interpolated`, rejilla, leyenda, ejes en formato "plain", y guardado en `output/`.
- `main.py`: orquestación del flujo de extremo a extremo (definición de rutas, creación de `output/`, lectura de datos, conteos y faltantes, interpolación, impresión en consola, escritura de archivos y creación de la gráfica).
- `requirements.txt`: dependencias mínimas para reproducir la figura (`matplotlib>=3.5.0`); la interpolación corre con la biblioteca estándar de Python.
- `README.md`: guía breve de uso (estructura del proyecto, cómo ejecutar, explicación del método, cómo insertar la imagen en LaTeX, y notas sobre limitaciones/escala).
- `.gitignore`: exclusiones recomendadas (p. ej., `__pycache__/`, `*.pyc`, entornos `.venv/`, y artefactos generados en `output/` salvo que se desee versionarlos).

Los datos internos se representan como tuplas `(year, population, has_data)`. Esta estructura permite distinguir puntos originales e interpolados sin añadir complejidad innecesaria. Por transparencia académica se emplea la forma directa de Lagrange empleando todos los datos, aunque, como se verá después, es preferible emplear pocos datos conocidos alrededor de cada valor que se desee aproximar.

# 5. Explicación bloque a bloque del núcleo `interpolation.py`

Se presenta el módulo central mediante alternancia de fragmento y explicación.

## Firma y propósito

```python
from typing import List, Tuple

def lagrange_interpolation(
    data: List[Tuple[int, float, bool]],
    target_year: int
) -> float:
    """Interpolacion de Lagrange en el periodo objetivo."""
```

La función recibe la serie completa y el año objetivo. Devuelve $P_{n-1}(x^*)$ evaluado en `target_year`.

## Extracción de nodos conocidos

```python
    known_years = []
    known_populations = []
    for year, population, has_data in data:
        if has_data:
            known_years.append(year)
            known_populations.append(population)
```

Se separan $(x_1, \ldots, x_n)$ y $(y_1, \ldots, y_n)$, es decir, años y poblaciones conocidas que conforman los nodos de interpolación.

## Validación y acumulador

```python
    n = len(known_years)
    if n < 2:
        raise ValueError("Not enough data points for interpolation")
    result = 0.0
```

Se exige $n \geq 2$ para realizar la interpolación. La variable `result` almacenará la suma $\sum_i y_i \ell_i(x^*)$.

## Implementación directa de la fórmula

```python
    for i in range(n):
        term = known_populations[i]
        for j in range(n):
            if i != j:
                term *= (target_year - known_years[j]) / \
                        (known_years[i] - known_years[j])
        result += term
    return result
```

Para cada $i$, se inicia el término en $y_i$ y se multiplica por todos los factores $\frac{x^* - x_j}{x_i - x_j}$ con $j \neq i$. Al finalizar el bucle interior, `term` corresponde a $y_i\, \ell_i(x^*)$ y se suma al acumulador. El retorno es $P_{n-1}(x^*)$.

### Aplicación a todos los años faltantes

```python
def interpolate_missing_data(
    data: List[Tuple[int, float, bool]]
) -> List[Tuple[int, float, bool]]:
    complete_data = data.copy()
    for i, (year, _, has_data) in enumerate(complete_data):
        if not has_data:
            try:
                y_hat = lagrange_interpolation(data, year)
                complete_data[i] = (year, round(y_hat), False)
            except ValueError as e:
                print(f"Error interpolating year {year}: {e}")
    return complete_data
```

Se recorre la serie; para cada año sin dato, se evalúa $P_{n-1}$ en ese año y se redondea al entero más cercano, conservando la marca de punto interpolado para su identificación posterior.

### Utilidades

```python
def get_missing_years(data):
    return [year for year, _, has_data in data if not has_data]

def count_known_data(data):
    return sum(1 for _, _, has_data in data if has_data)
```

Estas funciones permiten reportes rápidos: conteo de conocidos y listado de faltantes antes de interpolar.

## 6. Resultados

Los cinco años faltantes se presentan a continuación, aproximados al entero más cercano.

| Año | Población interpolada |
|---|---|
| 1996 | 3,304,963 |
| 1998 | 3,199,696 |
| 2008 | 2,993,236 |
| 2011 | 2,905,881 |
| 2012 | 2,902,922 |

Cuadro 1: Años faltantes interpolados para País 16

Los valores estimados siguen la tendencia decreciente observada en la serie histórica: de $\sim 3.30 \times 10^6$ en 1996 a $\sim 2.90 \times 10^6$ en 2012. El descenso más marcado se aprecia entre 1998 y 2008, mientras que entre 2011 y 2012 la variación es moderada.

## 7. Gráfica Año–Población

A partir de la serie completa 1990–2023 se generó la gráfica:

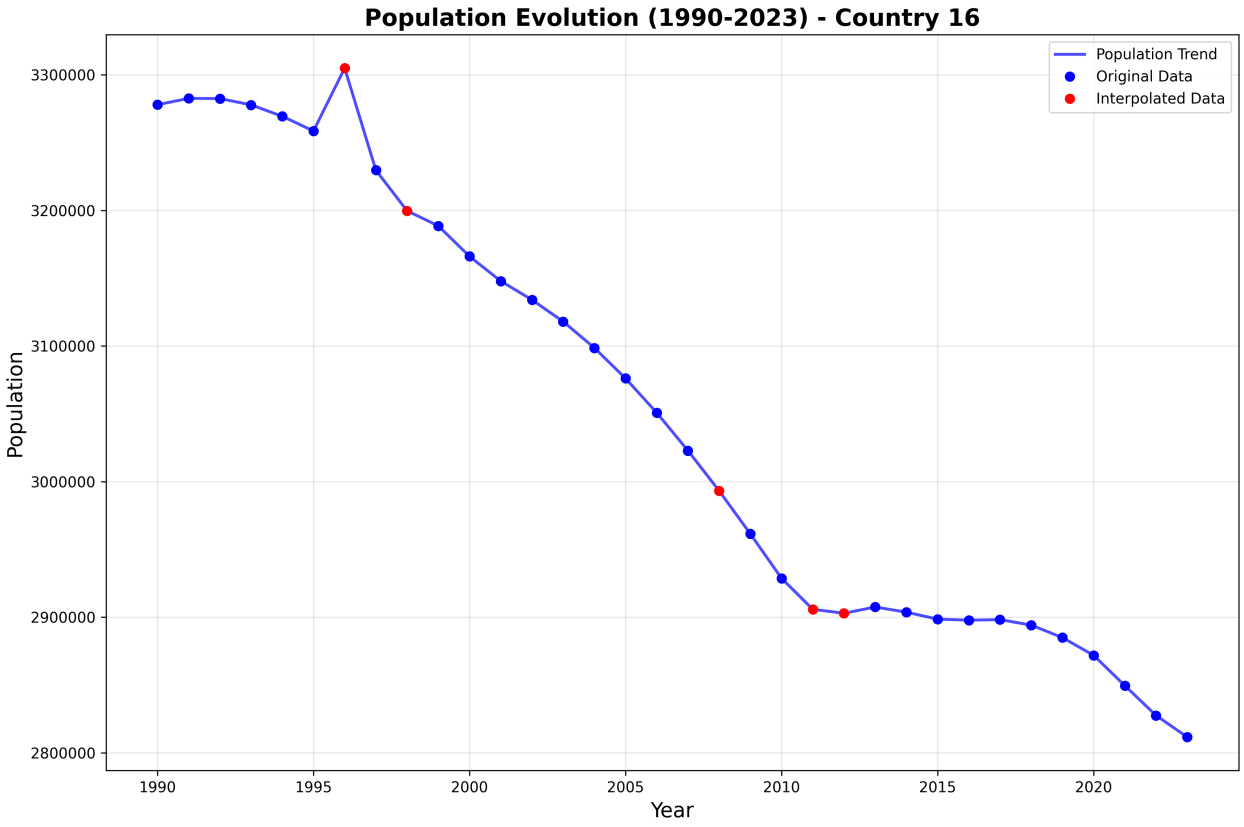


Figura 1: Evolución poblacional con identificación de puntos interpolados.

## 8. Discusión del valor estimado en 1996

El valor de 1996 se obtiene significativamente por encima de sus adyacentes. Mientras que 1995 reporta 3 258 573 y 1997 reporta 3 229 665, un promedio local simple de ambos es 3 244 119. La estimación por Lagrange empleando todos los nodos conocidos produce 3 304 963, superior en 60 844 unidades a ese promedio local.

Esta diferencia se asocia a la interpolación polinómica global de grado alto: al imponer paso exacto por todos los nodos, los polinomios base pueden asignar pesos relativos elevados en ciertas posiciones interiores, generando sobreimpulsos locales. La sensibilidad aumenta cuando la señal subyacente no es polinómica y presenta pequeñas irregularidades, y cuando se utiliza la forma directa en lugar de formulaciones numéricamente más estables. Para evitar este error se recomienda emplear una reducidas cantidad de datos cercanos al punto a aproximar.