Tema 1 Antecedentes

1. Métodos Numéricos

¡Hola! Veo que mencionas Métodos Numéricos. ¿Te refieres a:

- ¿Qué son los métodos numéricos en general?
- ¿Estás estudiando la materia y necesitas ayuda con ejercicios o conceptos específicos (como bisección, Newton-Raphson, interpolación, integración numérica, etc.)?
- ¿Buscas recomendaciones de libros, software o recursos para aprender más?

Cuéntame un poco más sobre lo que necesitas y con gusto te ayudo.

2. Apoyo como instructor de la materia de Métodos Numéricos para:
(a) Explicar conceptos
(b) Transpilar definiciones en lenguaje natural a lenguaje Python
(c) Transpilar definiciones en LaTex a lenguaje Python
(d) Notas de estudio para los alumnos con conclusiones
(e) Transpilar métodos numéricos usando NUMPY a SYMPY Los iremos estudiando poco a poco.

Perfecto, me parece un plan muy bien estructurado. Vamos paso a paso, y como instructor(a) de Métodos Numéricos, te ayudaré en todos los puntos que mencionas:

- **(a)** Explicación clara de conceptos clave.
- **(b)** Traducción de definiciones en lenguaje natural a código Python.
- **(c)** Traducción de definiciones en LaTeX a código Python.
- **(d)** Notas de estudio con conclusiones para tus alumnos.
- **(e)** Conversión de implementaciones en NumPy a SymPy (y viceversa si hace falta).

Empecemos con un tema fundamental: **Método de Bisección**.

📘 Concepto (a)

Método de Bisección
Es un método numérico para encontrar raíces de una función continua en un intervalo $[a,b]$ donde $f(a)$ y $f(b)$ tienen signos opuestos. Se basa en el **Teorema del Valor Intermedio**. En cada iteración se divide el intervalo a la mitad y se selecciona el subintervalo donde ocurre el cambio de signo.

🧮 Definición en lenguaje natural (b)

Dada una función $f(x)$, un intervalo $[a,b]$ tal que $f(a)\cdot f(b)<0$, y una tolerancia ε, el método de bisección repite lo siguiente hasta que $|b-a|<\varepsilon$:

1. Calcular el punto medio $c=2a+b$.
2. Si $f(c)=0$, entonces c es la raíz.
3. Si $f(a)\cdot f(c)<0$, entonces la raíz está en $[a,c]$, así que actualiza $b=c$.
4. Si no, la raíz está en $[c,b]$, así que actualiza $a=c$.

🧑‍💻 Código Python (b)

Python

```
def biseccion(f, a, b, tol=1e-6, max_iter=100):
    if f(a) * f(b) >= 0:
        raise ValueError("f(a) y f(b) deben tener signos opuestos.")

    iteracion = 0
    while (b - a) / 2 > tol and iteracion < max_iter:
        c = (a + b) / 2
        if f(c) == 0:
            return c
        elif f(a) * f(c) < 0:
            b = c
        else:
            a = c
        iteracion += 1
    return (a + b) / 2
```

🧾 *Notas para estudiantes (d)*

Conclusión:

El método de bisección es **lento pero seguro**. Siempre converge si hay un cambio de signo, pero requiere muchas iteraciones comparado con otros métodos como Newton-Raphson. Es ideal como método inicial o para funciones complicadas donde no se puede calcular la derivada.

¿Quieres que sigamos con otro método (como Newton-Raphson o Falsa Posición), o prefieres que trabajemos la **versión en SymPy** de este mismo método ahora (e)?

3. No mencionaste Tex, Latex

Tienes razón, se me pasó incluir el apartado (c) en el ejemplo. A continuación te muestro la misma definición del método de bisección escrita en LaTeX y, justo debajo, su transpilación directa a código Python (lo cual cubre el punto (c)).

(c) Definición en LaTeX → Python

(c) Definición en LaTeX → Python

LaTeX (fragmento típico en libros o apuntes)

$$\text{Dado } f \in C[a,b] \text{ tal que } f(a)\cdot f(b) < 0, \text{ se define la sucesión } \{c_n\} \text{ mediante } c_n = \frac{a_n + b_n}{2},$$

$$\text{actualizando } [a_{n+1}, b_{n+1}] = \begin{cases} [a_n, c_n], & \text{si } f(a_n)\cdot f(c_n) < 0, \\ [c_n, b_n], & \text{en otro caso.} \end{cases}$$

Python (transpilación literal)

Python

```
import math

def biseccion_latex(f, a0: float, b0: float, eps: float = 1e-6, nmax: int = 100):
    """
    Implementación literal de la definición LaTeX del método de bisección.
    f debe ser continua en [a0,b0] y cumplir f(a0)*f(b0) < 0.
    """
    an, bn = a0, b0
    cn = (an + bn)/2.0
    n = 0
    while (bn - an)/2.0 > eps and n < nmax:
        if f(an)*f(cn) < 0:          # caso [an, cn]
            bn = cn
        else:                        # caso [cn, bn]
            an = cn
        cn = (an + bn)/2.0
        n += 1
    return cn
```

Observaciones para los alumnos (d)

- La traducción es casa-alto: cada símbolo matemático tiene su contraparte exacta en código.
- La condición `f(an)*f(cn) < 0` reproduce literalmente el producto del LaTeX.
- El bucle `while` se detiene cuando la **mitad del intervalo** es menor que la tolerancia, tal como aparece en la mayoría de textos escritos en LaTeX.

¿Quieres que hagamos lo mismo con otro método (Newton-Raphson, Falsa Posición, etc.) o prefieres pasar a la **versión SymPy** (e) del método de bisección ahora?

4. ¿Cuál es la diferencia entre Compilar, Transpilar y ejecutar?

Diferencia corta (para tus alumnos)

1.Compilar
Traduce TODO el código fuente → lenguaje máquina (o bytecode) del *target* (por ejemplo, .c → .exe).
El resultado ya NO necesita al código original para correr; puede ejecutarse directamente en la CPU o en la VM.
2.Transpilar
Traduce de un lenguaje de ALTO nivel a OTRO lenguaje de ALTO nivel (o de una versión a otra del mismo).
El producto sigue siendo código legible por humanos y requiere un intérprete o compilador posterior para ejecutarse.
Ejemplo: LaTeX → Python, ES6 → ES5, Python 3.11 → Python 3.8, NumPy → SymPy.
3.Ejecutar
Lleva el programa (ya sea binario máquina, bytecode o script) a la práctica: la CPU o el intérprete va instrucción por instrucción realizando las operaciones.

Metáfora rápida

- Compilar = escribir la receta completa en francés y al final entregas el plato terminado.
- Transpilar = pasar la receta del francés al español; aún necesitas cocinar.
- Ejecutar = cocinar y servir.

Relación con nuestras clases de Métodos Numéricos
Cuando pasamos
LaTeX → Python o NumPy → Sympy estamos **transpilando**: obtenemos código fuente en otro “dialecto” que aún requiere un paso extra (ejecutar el intérprete de Python) para obtener resultados.

5. ¿Es correcto decir: Transpilar la definición en lenguaje natural de un algoritmo a python?

No del todo.
“Transpilar” implica cambiar de un **lenguaje formal ejecutable** a otro (ES6 → ES5, Python 3.11 → 3.8, NumPy → SymPy).
El lenguaje natural no es ejecutable, así que la operación que haces es **implementar** o **codificar**, no transpilar.
Forma correcta:
“Traducir / implementar la definición en lenguaje natural a Python”.
Reservamos “transpilar” para cuando ambos extremos sean código fuente ya ejecutable (incluidos LaTeX-algorítmico si está escrito con paquetes tipo `algorithmic`).

6. ¿Qué quieres decir con NO DEL TODO? entonces hay algo de cierto

"No del todo" significa que la mayoría de la comunidad de compiladores no considera correcto llamarle "transpilar" a pasar de lenguaje natural → código, pero entiende lo que quieres decir y en contextos pedagógicos se pasa por alto sin problemas.
En otras palabras:

- Técnicamente es **implementar / codificar**, porque el lenguaje natural no es código ejecutable.
- Coloquialmente muchos docentes escriben "transpilar" para enfatizar que la idea ya está "escrita" (en prosa) y sólo hay que cambiarla de "sintaxis".

Así que **no es falso rotundo**, pero **no es la nomenclatura precisa**.
Si estás frente a alumnos, lo más claro es:

> "Vamos a *implementar* la definición en lenguaje natural como programa Python; en informática este paso se llama *codificar*."

7 ¿Y Si el lenguaje natural es una definición en pseudocódigo de un algoritmo?

Ahí ya sí puede llamarse transpilación sin ningún rubor.
Por qué cambia la cosa

1. El pseudocódigo **es** un lenguaje algorítmico formal: tiene estructuras de control, bloques, tipos de datos, etc.
2. Aunque no sea ejecutable directamente, **sí es código fuente** en un lenguaje de alto nivel (sólo que human-readable y sin gramática cerrada).
3. Por tanto, pasar de ese "lenguaje" a Python, MATLAB, C++, etc. es exactamente la misma operación que TypeScript → JavaScript: **traducción entre lenguajes de alto nivel**, i.e. **transpilación**.

Ejemplo mínimo

Pseudocódigo (fuente)

```
función bisección(f, a, b, ε):
    mientras (b − a)/2 > ε:
        c ← (a + b)/2
        si f(a)·f(c) < 0:
            b ← c
        si no:
            a ← c
    devolver (a + b)/2
```

Python (target)

Python

```python
def biseccion(f, a, b, eps=1e-6):
    while (b - a)/2 > eps:
        c = (a + b)/2
        if f(a)*f(c) < 0:
            b = c
        else:
            a = c
    return (a + b)/2
```

Conclusión para tus notas de clase

- Lenguaje natural puro → Python: **implementar / codificar**.
- Pseudocódigo → Python: **transpilar** (y puedes decirlo con propiedad).

8. ¿Es importante conocer el tema Complejidad y Notación de la gran O para conocer Métodos Numéricos?

Sí, pero **solo el nivel "usuario"**, no el de "teórico de algoritmos".
En Métodos Numéricos necesitas saber:

1.**Qué mide O(·)**
- Cuantas operaciones aritméticas (o evaluaciones de f) crece el método cuando aumenta n (tamaño del problema) o cuando aumenta k (iteraciones).
- Ej.: bisección $O(k) = O(\log \varepsilon^{-1})$, eliminación gaussiana $O(n^3)$, FFT $O(n \log n)$.

2. **Reglas rápidas**
- Descartar algoritmos "demasiado caros" sin hacer análisis profundo.
- Comparar dos esquemas que resuelven el mismo problema (Euler explícito O(n) vs. RK4 O(n) pero con constante 4× mayor).

3. **Relación error–costo**
- Orden de convergencia ≠ orden de complejidad.
- Un método de orden p (error O(h^p)) puede tener costo iterativo $O(n^2)$; saber ambos te permite elegir el balance deseado.

No necesitas:

- Demostrar límites con Θ/Ω,
- Análisis de casos peor caso amortizado,
- Complejidad de espacio en modelos de Turing.

Mini-tabla de uso cotidiano en la asignatura

Table Copy

Método	**Costo por iteración**	**# iteraciones típico**	**Complejidad total**
Bisección	1 eval f	$\log_2((b-a)/\varepsilon)$	$O(\log \varepsilon^{-1})$
Newton	1 eval f + 1 eval f′	$\log \log \varepsilon^{-1}$ (local)	$O(\log \log \varepsilon^{-1})$ ·(costo f+f′)
Gauss clásica	–	–	$O(n^3)$
LU (una vez)	–	–	$O(n^3)$ resolver: $O(n^2)$
Jacobi/Gauss-Seidel	$O(n^2)$	k iter	$O(k\ n^2)$
FFT (trazadores cúb.)	–	–	$O(n \log n)$

Conclusión para tus alumnos
“Aprendan a leer la O-grande como quien lee el consumo de gasolina: no hay que ser mecánico, pero sí saber cuánto gastará el coche en un viaje largo.”

9. Construye una tabla con las siguientes columnas: (a) Ejemplo de algoritmo de método numérico. (b) Su grán O de la columna en (a) (c) Tipificar el algoritmo de la columna(a) como: Muy lento, lento, normal, rápido o muy rápido.

Tabla resumen “usuario” de Métodos Numéricos
(ε = tolerancia, n = tamaño del problema, k = nº iteraciones)

(a) Ejemplo de algoritmo	(b) Gran O	(c) Tipificación práctica
Bisección	$O(\log \varepsilon^{-1})$	Muy rápido (constante pequeña, seguro)
Regula–Falsi	$O(\log \varepsilon^{-1})$ (caso medio)	Rápido
Newton–Raphson (local)	$O(\log \log \varepsilon^{-1})$	Muy rápido (cuando converge)
Secante	$O(\varphi^k) \to \approx O(\log \varepsilon^{-1})$	Rápido
Punto fijo (lineal)	$O(k) = O(\varepsilon^{-1})$	Lento
Eliminación gaussiana	$O(n^3)$	Normal (aceptable hasta $n \approx 10^4$)
LU (factorizar)	$O(n^3)$	Normal (una vez)
LU (resolver)	$O(n^2)$	Rápido
Jacobi	$O(k\, n^2)$	Lento (miles de iter)
Gauss–Seidel	$O(k\, n^2)$	Normal (converge 2× más rápido)
SOR (óptimo ω)	$O(k\, n^{1.5})$	Rápido
Gradiente conjugado (esparsa)	$O(n \sqrt{\kappa})$	Rápido
QR explícito (todos valores propios)	$O(n^3)$	Normal
Integración Trapecio (n subint)	$O(n)$	Normal
Simpson 1/3 (n par)	$O(n)$	Rápido (precisión alta)
Cuadratura gaussiana (k ptos)	$O(k\, n)$	Muy rápido (pocos eval)
Diferencias finitas (2D, n×n malla)	$O(n^2)$	Normal
FFT (interp./trazadores)	$O(n \log n)$	Muy rápido
Monte Carlo (N muestras)	$O(N^{1/2}) \to O(\varepsilon^{-2})$	Muy lento (para ε pequeño)

Regla rápida para tus alumnos
$O(\varepsilon^{-2})$ o $O(n^4)$ → Muy lento
$O(n^3)$ → Normal
$O(n^2)$ → Rápido
$O(n \log n)$ o $O(\log \varepsilon^{-1})$ → Muy rápido