

Portafolios Taller Transversal I

Ekaitz Arriola Garcia

26 de junio de 2025

Índice

1. Introducción	5
2. Practica I	5
2.1. Criterios para Evaluar Código Fuente	5
2.1.1. Calidad del Código (Buenas Prácticas, Estructura y Organización) .	5
2.1.2. Reutilización y Modularidad	5
2.1.3. Gestión de Errores y Excepciones	5
2.1.4. Pruebas y Cobertura de Pruebas	5
2.1.5. Documentación y Comentarios	6
2.1.6. Seguridad	6
2.1.7. Eficiencia y Rendimiento	6
2.2. SGP4DC	6
2.3. Informe crítico sobre el código SGP4DC	6
2.3.1. Calidad del Código:	6
2.3.2. Reutilización y Modularidad:	7
2.3.3. Gestión de Errores:	7
2.3.4. Documentación:	7
2.3.5. Seguridad:	7
2.3.6. Eficiencia:	8
2.4. Mejoras necesarias en el código	8
2.4.1. Refactoring:	8
2.4.2. Mejoras estructurales:	8
2.4.3. Gestión de errores	8
2.4.4. Testing y validacion:	9
2.4.5. Documentacion:	9
2.4.6. Justificacion tecnologica:	9
2.5. Informe de compilacion del codigo SGP4DC	9

2.5.1. Constructor de Vector Bidimensional Inválido (Error Principal)	9
2.5.2. Headers Faltantes	10
2.5.3. Variables No Declaradas	10
2.5.4. Directivas de Preprocesador Malformadas	10
2.5.5. Errores de Switch-Case	11
2.5.6. Shadowing de Variables	11
2.5.7. Warnings de Formato	11
3. Practica II	12
3.1. Clasificación de lenguajes de programación	12
3.1.1. Clasificación	12
3.1.2. Criterios para elegir un lenguaje	15
3.2. Evaluación de GraalVM	17
3.2.1. Por nivel de abstracción	18
3.2.2. Por paradigma de programación	18
3.2.3. Por modelo de ejecución	18
3.2.4. Por sistema de tipos	18
3.2.5. Por dominio específico	18
3.2.6. Por concurrencia	19
3.2.7. Por gestión de memoria	19
3.2.8. Por plataforma objetivo	19
3.3. QuickSort en diferentes lenguajes de programación	19
3.4. Eficiencia de Quicksort en diferentes lenguajes de programación (version nativa y manual)	19
3.4.1. Python	20
3.4.2. C	21
3.4.3. C++	21
3.4.4. Java	21
3.5. Definir y justificar criterios para analizar el código Lambert Battin	21
3.6. Informe crítico Lambert Battin	21
4. Practica III	22
4.1. Traducción del código Lambert Battin	22
4.2. Lenguajes compilados e interpretados	22
4.2.1. Lenguajes Compilados	23
4.2.2. Lenguajes Interpretados	23
4.3. Criterios para analizar editores de texto	24
4.3.1. Objetivos de la evaluación	24
4.3.2. Criterios de evaluacion por categorias:	25

4.4.	Análisis de Editores de Texto e IDEs para Desarrollo en C/C++ y Python	27
4.4.1.	Objetivos y Criterios de Evaluación	27
4.4.2.	Análisis Individual por Editor	28
4.4.3.	Tabla Comparativa de Editores	35
5.	Practica IV	36
5.1.	Selección de Analizadores	36
5.2.	Criterios de Selección de Analizadores de Código Estático	36
5.2.1.	Marco de Evaluación	36
5.3.	Análisis Comparativo de Analizadores de Código Estático: PVS-Studio vs Cppcheck	39
5.3.1.	Criterios	40
5.3.2.	Criterios de Usabilidad y Experiencia	40
5.3.3.	Criterios Comerciales y de Sostenibilidad	41
5.3.4.	Criterios de Seguridad y Compliance	41
5.3.5.	Evaluación Práctica	42
6.	Practica V	43
6.1.	Herramientas de pruebas de software	43
6.2.	Criterios de Evaluación para Herramientas de Testing de Software	43
6.2.1.	Criterios de Evaluación	44
6.2.2.	Metodología de Evaluación	46
6.2.3.	Consideraciones Finales	46
6.3.	Análisis Comparativo de Herramientas de Testing de Software (assert.h . CppTest)	46
6.3.1.	Facilidad de Uso e Implementación	47
6.3.2.	Funcionalidades Técnicas	47
6.3.3.	Reportes y Debugging	48
6.3.4.	Rendimiento y Escalabilidad	49
6.3.5.	Compatibilidad y Portabilidad	49
6.3.6.	Ecosistema y Mantenimiento	50
6.3.7.	Aspectos Específicos del Contexto Académico	50
6.3.8.	Análisis Comparativo Final	51
7.	Practica VI	51
7.1.	Lectura de la guía de estilo e investigar sistemas de documentación	51
7.2.	Criterios de Evaluación para Sistemas de Documentación	52
7.2.1.	Criterios de Evaluación	52
7.2.2.	Metodología de Análisis	54
7.3.	Evaluación de Sistemas de Documentación: Doxygen vs JavaDoc	54

7.3.1. Doxygen	54
7.3.2. JavaDoc	59
7.4. Documentación del código Cpp del proyecto Lambert	63
8. Proyecto VII	64
8.1. Criterios de Evaluación para Herramientas de Análisis de Rendimiento . . .	64
8.1.1. Criterios de Evaluación	64
8.1.2. Consideraciones Específicas del Contexto	66
8.2. Análisis de Herramientas de Análisis de Rendimiento	66
8.2.1. Timemory	67
8.2.2. Gprof	68
8.2.3. Valgrind	70
8.2.4. Insure++	72
8.2.5. Instruments (macOS)	74
8.2.6. Tabla Comparativa	76
8.3. Análisis de Lambert Battin	78
8.3.1. Análisis de Rendimiento y Memoria Usando Valgrind	78
8.3.2. Análisis del Output de Gprof Usando Gprof	80

1 Introducción

En este portafolios se recopilan las tareas realizadas durante el curso, indicando cada tarea junto a su estimacion de tiempo y tiempo necesitado para realizarlo.

Junto a este portafolios estaran adjuntos los recursos nombrados durante las practicas.

Tambien accesibles desde [Github](#).

2 Practica I

2.1 Criterios para Evaluar Código Fuente

Tarea	Tiempo estimado	Tiempo real
Definir criterios que seguir para analizar el código	5 minutos	15 minutos

Tiempo estimado basado en experiencia previa en búsqueda de datos en internet.

2.1.1. Calidad del Código (Buenas Prácticas, Estructura y Organización)

Evalúa la claridad y organización del código, aspectos clave para su comprensión y mantenimiento: * Claridad en la jerarquía de carpetas, archivos y módulos. * Consistencia en la sintaxis y convenciones. * Uso adecuado de nombres descriptivos y comentarios relevantes.

2.1.2. Reutilización y Modularidad

Mide la capacidad del código para ser reutilizado y adaptado en diferentes contextos:

- Separación del código en componentes independientes.
- Reducción de dependencias innecesarias.
- Facilidad para integrar módulos en nuevos proyectos.

2.1.3. Gestión de Errores y Excepciones

- Implementación de bloques try-catch o equivalentes.
- Notificación comprensible de fallos.
- Continuidad de la ejecución ante errores menores.

2.1.4. Pruebas y Cobertura de Pruebas

- Validación de componentes individuales.
- Comprobación del trabajo conjunto de módulos.
- Porcentaje de código cubierto por pruebas.

2.1.5. Documentación y Comentarios

- Explicaciones concisas de secciones complejas.
- Guías de uso, instalación y configuración.
- Representaciones gráficas de la arquitectura y flujos.

2.1.6. Seguridad

- Prevención de inyecciones y ataques.
- Protección de datos sensibles.
- Control de accesos.

2.1.7. Eficiencia y Rendimiento

Evalúa el uso óptimo de recursos, como memoria y tiempo de ejecución:

- Selección de estructuras de datos y algoritmos adecuados.
- Reducción de tiempos de respuesta y consumo de memoria.
- Inclusión de librerías necesarias y actualizadas.
- Prevención de bibliotecas innecesarias.
- Verificación de versiones y licencias.

2.2 SGP4DC

SGP4DC es un código de correcciones diferenciales que utiliza el modelo de propagación analítica SGP4 para el refinamiento orbital. Implementa técnicas de ajuste por mínimos cuadrados usando descomposición de valores singulares (SVD) para mejorar la precisión de los elementos orbitales TLE mediante observaciones reales.

2.3 Informe crítico sobre el código SGP4DC

Tarea	Tiempo estimado	Tiempo real
Redactar informe critico	90 minutos	125 minutos

Tiempo estimado basado en experiencia previa en sintexis de información recopilada y redacción de informes.

2.3.1. Calidad del Código:

- Inconsistencias severas en formato: algunos archivos usan espacios, otros tabs. No importaria mucho si no fuese porque en algunos usan identacion con espacios impares... Tambien identacion no homogenea.

- Falta de convenciones consistentes para nombres de funciones (`iau76fk5allitrfgcrf`, `twoline2rv`)
- Una sola palabra: `"dsvbksb"`... (me gustaría dejarlo así porque es autoexplicativo, pero por si acaso: nombres de funciones muy concentradas generadas a partir de acortaciones pegadas a la fuerza, como `dsvdemp`. Si andas dentro del contexto bien, pero si sin leer el código ni saber contexto sabes que es un `dsvbksb`, llama una ambulancia, estas teniendo un ictus). Ni siquiera es el peor ejemplo

2.3.2. Reutilización y Modularidad:

- Muchas dependencias cruzadas entre módulos. `SGP4DC.h` se incluye a si mismo...
- Funciones demasiado largas con múltiples responsabilidades. Por ejemplo, en `findatwaatwb` puedo contar hasta 8 responsabilidades: Acumulación de matrices AT-WA/ATWB, cálculos matriciales y pesos, diferenciación finita, manejo de diferentes tipos de observaciones, transformaciones de tiempo, propagación orbital SGP4, y la configuración inicial junto al loop principal de observaciones.

2.3.3. Gestión de Errores:

- Inexistente: no hay try-catch ni validaciones (personalmente, no creo en el try-catch). Por ejemplo en `finitediff` no verifica el array `xnom`, ni `state2satrec` si falló, etc. En `pronttle` hay operaciones de formato sin validación.
- Funciones tipo void que no retornan códigos de error (`state2satrec`, `finitediff`...).
- No hay verificación de punteros nulos antes de usar archivos

2.3.4. Documentación:

- Comentarios poco descriptivos en cuanto a procedimientos. Aunque esta bien indicado inputs y outputs.
- Código de debug mezclado con producción sin explicación (`SGP4DC.cpp`, función `state2satrec`, `leastquares`)

2.3.5. Seguridad:

- Uso inseguro de funciones C (`strcpy`, `printf` con format strings, `gets`) (!!!!)
- No hay validación de entrada de datos (en `dsvbksb` no se validan las dimensiones de matrices, en `findatwaatwb` se acceden a arrays sin verificación)
- Posibles buffer overflows en manejo de strings (`pronttle`: buffers de tamaño fijo vulnerables)
- Variables globales accesibles sin control (los files en `testdc`, cualquier función puede modificarlos; globales sin protección y algunas mal definidas en `SGP4DC`)
- Ojala los del banco santander programasen también así

2.3.6. Eficiencia:

- Calculos repetitivos sin cache (Propagacion repetida en finndatwaatwb; operaciones trigonométricas repetidas en state2satrec; calculos ESTATICOS repetidos como en printtle, finitediff o leastsquares)
- Uso excesivo de printf para debug que impacta el rendimiento del programa
- No hay optimizacion de estructuras de datos (matrices ineficientes std::vector de std::vector HORRIBLE iugh, y encima creadas en cada llamada en findatwaatwb; retamñificacion de vector ineficiente, es MASIVA)

2.4 Mejoras necesarias en el código

Tarea	Tiempo estimado	Tiempo real
Enumerar mejoras necesarias	15 minutos	30 minutos

Tiempo estimado basado en experiencia previa en analisis de código.

2.4.1. Refactoring:

- Usar clang-tidy para detectar problemas automaticamente
- Implementar clang-format para unificar el estilo de codigo PORFAVOR
- Separar codigo de debug en builds condicionales (#ifdef DEBUG)
- Separar responsabilidades
- Eliminar dependencias circulares

2.4.2. Mejoras estructurales:

- Usar smart pointers en lugar de punteros raw
- Crear clases wrapper para operaciones de archivo seguras
- Reemplazar la ABOMINACIÓN de std::vector std::vector double con Eigen matrices
- Implementar cache para calculos trigonometricos repetitivos y memory pools para objetos temporales.
- Variables globales → Dependency Injection con interfaces
- Async processing con std::async para cálculos paralelos

2.4.3. Gestión de errores

- Migrar funciones void a tipos Result T, Error o excepciones C++
- Implementar logging con spdlog
- Validar entrada de datos con contracts o asserts
- Usar AddressSanitizer para detectar buffer overflows

2.4.4. Testing y validacion:

- Catch2 para unit testing de funciones matemáticas
- Google Benchmark para performance testing
- PVS-Studio para static analysis crítico
- Valgrind para memory leak detection

2.4.5. Documentacion:

- Doxygen para generar documentacion
- Diagramas UML con PlantUML para diagramas de flujo (lo siento pero no puedo usar otro que no sea PlantUML, ya estoy casado con ello, es el unico que se traga bien el chatg...)

2.4.6. Justificacion tecnologica:

Elijo estas herramientas porque:

- PVS-Studio detecta exactamente los bugs encontrados
- Eigen optimiza operaciones matriciales 10-100x vs std::vector de std::vector
- Catch2 permite testing de algoritmos numéricos con tolerancias apropiadas
- spdlog es thread-safe y eficiente para aplicaciones críticas

2.5 Informe de compilacion del codigo SGP4DC

Tarea	Tiempo estimado	Tiempo real
Documentar intento de compilación	10 minutos	45 minutos

Tiempo estimado basado en experiencia previa.

Tiempo real notablemente diferente al tiempo estimado debido a no recordar como se compilaba... Ya se, debería saber compilar eso como respirar a este punto, pero se me olvidó.

2.5.1. Constructor de Vector Bidimensional Inválido (Error Principal)

Este es el error más crítico y frecuente en toda la base de código. Aparece unas 20-30 veces entre múltiples archivos:

```

1      astiod.cpp:195:48: error: no matching function for call to 'std
2      ::vector<std::vector<double> >::vector(int, int)'
3      195 |     std::vector< std::vector<double> > lmatii(3,3), cmat
         |     (3,3), rhomat(3,3),
         |

```

```

1  astmath.cpp:1015:60: error: no matching function for call to '
std::vector<std::vector<double> >::vector(int, int)'
2  1015 |         std::vector< std::vector<double> > lu(order+1,
order+1);
3      |

```

El problema radica en que no existe un constructor de 'std::vector' que tome dos enteros para crear directamente una matriz. La sintaxis correcta requiere usar 'resize()' o inicialización apropiada.

2.5.2. Headers Faltantes

Funciones de manipulación de cadenas no están disponibles por falta del header correspondiente:

```

1  asttime.cpp:46:6: error: 'strcpy' was not declared in this scope
2  46 |         strcpy(monstr[1], "Jan");
3      |         ~~~~~
4  asttime.cpp:33:1: note: 'strcpy' is defined in header '<cstring>';
    did you forget to '#include <cstring>'?

```

```

1  asttime.cpp:60:14: error: 'strcmp' was not declared in this scope
2  60 |         while ((strcmp(instr, monstr[ktr])!=0) && (ktr <=12))
3      |

```

La solución es agregar '#include <cstring>' en 'asttime.cpp'.

2.5.3. Variables No Declaradas

Variables específicas del sistema que no están definidas en el contexto actual:

```

1  testdc.cpp:58:24: error: '_argc' was not declared in this scope
2  58 |         cout << "argc= " << _argc << endl;
3      |         ~~~~~
4  testdc.cpp:61:15: error: '_argv' was not declared in this scope
5  61 |         cout << _argv[i] << endl;
6      |         ~~~~~

```

Probablemente faltan includes específicos del sistema o estas variables necesitan ser definidas de manera diferente.

2.5.4. Directivas de Preprocesador Malformadas

Token extra al final de la directiva include:

```

1 sgp4dc.h:37:75: warning: extra tokens at end of #include directive
2 37 | #include "astiod.h"
      |
3     |
      ^

```

Hay un carácter backtick (`) innecesario al final de la línea que causa el warning.

2.5.5. Errores de Switch-Case

Sintaxis incorrecta en declaración de casos múltiples:

```

1 sgp4dc.cpp:817:16: error: expected ':' before ',' token
2 817 |         case 1,3 : indobs= 2;
      |                ^
3     |
4     |                :
5 sgp4dc.cpp:817:16: error: expected primary-expression before ','
   token

```

La sintaxis 'case 1,3:' es inválida. Debería ser 'case 1: case 3:' para casos múltiples.

2.5.6. Shadowing de Variables

Variables locales que ocultan parámetros de función:

```

1 coordfk5.cpp:375:15: error: declaration of 'double rpef [3]'
   shadows a parameter
2 375 |         rpef[3], vpef[3], apef[3], omgxv[3], tempvec1
      |         [3], tempvec[3];
      |         ~~~~
3     |
4 coordfk5.cpp:357:15: note: 'double* rpef' previously declared here
5 357 |         double rpef[3], double vpef[3], double apef[3],
      |         ~~~~~~
6     |

```

Se están redeclarando variables que ya existen como parámetros de la función.

2.5.7. Warnings de Formato

Inconsistencias entre especificadores de formato y tipos de datos:

```

1 testdc.cpp:257:52: warning: format '%d' expects argument of type '
   int*', but argument 4 has type 'long int*' [-Wformat=]
2 257 |         sscanf(longstr2,"%d %d %d %d %d
      |         %d %d %d %lf ",
3     |
      ~

```

```

4 |
5 |
6 |
int*
%ld

testdc.cpp:1163:37: warning: format '%d' expects argument of type '
int', but argument 3 has type 'long int' [-Wformat=]
1163 |
      fprintf(outfile1, " %8d difference %11lf %11
lf %11lf km %12lf m \n\n", satrec.satnum, dr[0], dr[1], dr[2],
      1000.0 * mag(dr) );
3 |
      ~~~
      ~~~~~~
4 |
      |
5 |
      int
      long int

```

El especificador %d espera int pero recibe long int. Debería usarse %ld.

3 Practica II

3.1 Clasificación de lenguajes de programación

Tarea	Tiempo estimado	Tiempo real
Investigación y clasificación de lenguajes de programación	30 minutos	55 minutos

Tiempo estimado basado en experiencia previa en investigación y sintesis de información.

3.1.1. Clasificación

Existen miles de lenguajes de programación, pero elegir el lenguaje correcto para la tarea a realizar es critico. Elegir el lenguaje correcto para una tarea en concreto, de entre los miles que existen, es critico. No todos los lenguajes son recomendados para todo. Usar python para rpogramar un microcontrolador sería como usar cizañas para circuncidar una pulga. Cada lenguaje fue diseñado con ciertos objetivos en mente, y entenderlos ayuda a la selección correcta de este.

Por nivel de abstracción

Lenguajes de bajo nivel

- **Lenguaje de máquina:** Solo 0s y 1s
- **Assembly:** Mnemonicos para instrucciones de maquina

Lenguajes de nivel medio

- **C:** Control total sobre memoria y hardware.
- **C++:** C, pero ahora se identifica como orientado/objetos. Java con sabor windows.
- **Rust:** Seguridad de memoria sin perder rendimiento.

Lenguajes de alto nivel

- **Python:** Sintaxis casi como inglés, muy legible. Le escupes y aun compila (aunque sea interpretado; era una forma de hablar).
- **Java:** .^{Es}cribe una vez, ejecuta en cualquier lugar"
- **JavaScript:** Interpretado, dinámico, horrible.

Por paradigma de programación

Paradigma imperativo

- *Procedural:* C, Pascal, COBOL, Fortran
- *Orientado a objetos:* Java, C++, C#, Smalltalk

Paradigma declarativo

- *Funcional:* Haskell, Lisp, Erlang, F#, Clojure
- *Lógico:* Prolog, Mercury

Paradigma multiparadigma

- Python, JavaScript, Scala, Ruby, Swift, Kotlin

Por modelo de ejecución

Compilados

- C, C++, Rust, Go, Haskell
- Se traducen completamente antes de ejecutar

Interpretados

- Python, Ruby, PHP, JavaScript
- Se ejecutan linea por linea en tiempo real

Híbridos (bytecode)

- Java (JVM), C# (.NET), Python (también puede ser)
- Compilan a código intermedio primero

Por sistema de tipos

Tipado estático

- C, C++, Java, C#, Rust, Haskell
- Los tipos se comprueban antes de ejecutar

Tipado dinámico

- Python, JavaScript, Ruby, PHP
- Los tipos se comprueban durante la ejecución

Tipado fuerte vs débil

- *Fuerte*: Python, Java (no permite conversiones automáticas peligrosas)
- *Débil*: C, JavaScript (permite más conversiones automáticas) (recomendado ver como se calcula el 1/sqrt rapido, ufffff eso es +18, incluso +84)

Por dominio específico

Lenguajes de proposito general

- Python, Java, C++, JavaScript, C#

Lenguajes de dominio específico (DSL)

- SQL: bases de datos
- HTML/CSS: estructura y diseño web
- R: análisis estadístico
- MATLAB: computación numérica
- Verilog/VHDL: diseño de hardware
- LaTeX: composición de documentos, y para que te miren mal en publico cuando se te olvida poner overleaf despues de latex en google

Otros:**Por concurrencia**

- *Actor model*: Erlang, Elixir
- *CSP (Communicating Sequential Processes)*: Go
- *Threads tradicionales*: Java, C++
- *Event-driven*: JavaScript, Node.js

Por gestión de memoria

- *Manual*: C, C++
- *Garbage collection*: Java, C#, Python, JavaScript
- *Ownership*: Rust (Odio, pero amo. Iugh)
- *Reference counting*: Swift, Python (parcialmente)

Por plataforma objetivo

- *Sistemas*: C, C++, Rust, Assembly
- *Web*: JavaScript, TypeScript, PHP, Python
- *Móvil*: Swift, Kotlin, Dart, Objective-C
- *Científico*: Python, R, Julia, MATLAB
- *Empresarial*: Java, C#, COBOL

3.1.2. Criterios para elegir un lenguaje**Tipo de aplicación****Aplicaciones web**

- Frontend: JavaScript es practicamente obligatorio, TypeScript si no tuviste caida neonatal.
- Backend: Python (Django/Flask), Node.js, Java, Go

Sistemas operativos y drivers

- C y C++ son los reyes aquí
- Rust está ganando terreno por su seguridad. Guerra civil en linux por esto me suena que escuché.

Aplicaciones móviles

- iOS: Swift (moderno) u Objective-C (legacy)
- Android: Kotlin (preferido) o Java
- Multiplataforma: Flutter (Dart) o React Native (JavaScript)

Inteligencia artificial y datos

- Python domina completamente (TensorFlow, PyTorch, pandas)
- R para estadísticas complejas (prefiero que me y con una farola antes de usarlo un segundo más)
- Julia para computación científica de alto rendimiento

Videojuegos

- C# con Unity para indie games
- JavaScript para juegos web

Rendimiento requerido

Muy alta velocidad: C, C++, Rust

- Trading de alta frecuencia
- Sistemas en tiempo real
- Controladores de hardware

Velocidad buena: Go, Java, C

- APIs con mucho tráfico
- Aplicaciones empresariales
- Servicios web escalables

Velocidad moderada: Python, JavaScript, Ruby

- Prototipos rápidos
- Scripts de automatización
- Aplicaciones internas

Facilidad de desarrollo

Muy fáciles de aprender

- Python: sintaxis casi como inglés
- JavaScript: muchos recursos y tutoriales

Dificultad media

- Java: verboso pero estructurado
- C#: similar a Java pero similar a Java
- Go: simple pero con conceptos nuevos (otra vez java)

Difíciles

- C++: mucha complejidad y detalles (personalmente lo ponía en media, pero internet le tiene trauma)
- Rust: conceptos de ownership
- Haskell: paradigma muy diferente

Ecosistema y librerías**Ecosistemas masivos**

- JavaScript: npm tiene cientos de miles de paquetes. Es como el chino, hay de to', y en verano es navidad.
- Python: pip, conda, todo tipo de librerías científicas
- Java: Maven Central, frameworks empresariales maduros

Ecosistemas especializados

- R: CRAN para estadísticas
- Swift: principalmente para iOS/macOS
- MATLAB: computación numérica

3.2 Evaluación de GraalVM

Tarea	Tiempo estimado	Tiempo real
Evaluación de GraalVM	60 minutos	240 minutos

Tiempo estimado basado en experiencia previa en tareas similares.

Tiempo real notablemente diferente al tiempo estimado debido a problemas técnicos

Recomendado para

- Microservicios
- Aplicaciones cloud-native
- Apps empresariales Java existentes
- Proyectos políglotas

3.2.1. Por nivel de abstracción

Nivel: Medio-Alto (híbrido)

- **Como JIT:** Alto nivel, se comporta como cualquier JVM tradicional
- **Como Native Image:** Nivel medio, genera código nativo pero mantiene abstracciones Java

3.2.2. Por paradigma de programación

Paradigma: Multiparadigma extremo

- Soporta todos los paradigmas que soporta Java (OOP, funcional, etc.)
- Además permite ejecutar JavaScript, Python, Ruby, R, C/C++ en la misma aplicación

3.2.3. Por modelo de ejecución

Categoría: Híbrido avanzado (nueva categoría)

- **JIT mode:** Compilación just-in-time
- **Native Image:** Compilación ahead-of-time

3.2.4. Por sistema de tipos

Tipado: Depende del lenguaje huésped

- Java en GraalVM mantiene tipado estático fuerte
- JavaScript, Python, Ruby mantienen su sistema de tipos original pero con optimizaciones JIT

3.2.5. Por dominio específico

Clasificación: Multipropósito con especialización cloud-native

- **Propósito general:** Compatible con frameworks como Helidon, Micronaut, Quarkus, Spring Boot

3.2.6. Por concurrencia

Modelo: Hereda modelos de lenguajes huésped

- Java: threads tradicionales, Fork/Join
- JavaScript: event-driven

3.2.7. Por gestión de memoria

Tipo: Garbage collection avanzado + manual

- **JIT mode:** GC optimizado con tiempos de pausa significativamente menores
- **Native Image:** Substrate VM con GC embebido más eficiente

3.2.8. Por plataforma objetivo

Targets: Multiplataforma cloud-first

- **Sistemas:** Linux, macOS, Windows (x64, AArch64)
- **Cloud:** Optimizado para Kubernetes, OpenShift, entornos serverless

3.3 QuickSort en diferentes lenguajes de programación

Tarea	Tiempo estimado	Tiempo real
Implementación de QuickSort en diferentes lenguajes de programación	15 minutos	10 minutos

Tiempo estimado basado en experiencia previa en tareas similares.

Se implemento Quicksort en C, Cpp, java y python. Se encuentran adjuntos dentro de "P2_quicksort".

3.4 Eficiencia de Quicksort en diferentes lenguajes de programación (version nativa y manual)

Tarea	Tiempo estimado	Tiempo real
Evaluación y comparación de la eficiencia del algoritmo en diferentes lenguajes de programación	30 minutos	50 minutos

Tiempo estimado basado en experiencia previa en tareas similares.

Tarea	Tiempo estimado	Tiempo real
Ampliar comparación con versiones nativas del Quicksort en los diferentes lenguajes de programación	10 minutos	25 minutos

Tiempo estimado basado en experiencia previa en tareas similares.

Se evaluará dicha eficiencia en C, Cpp, java y python.

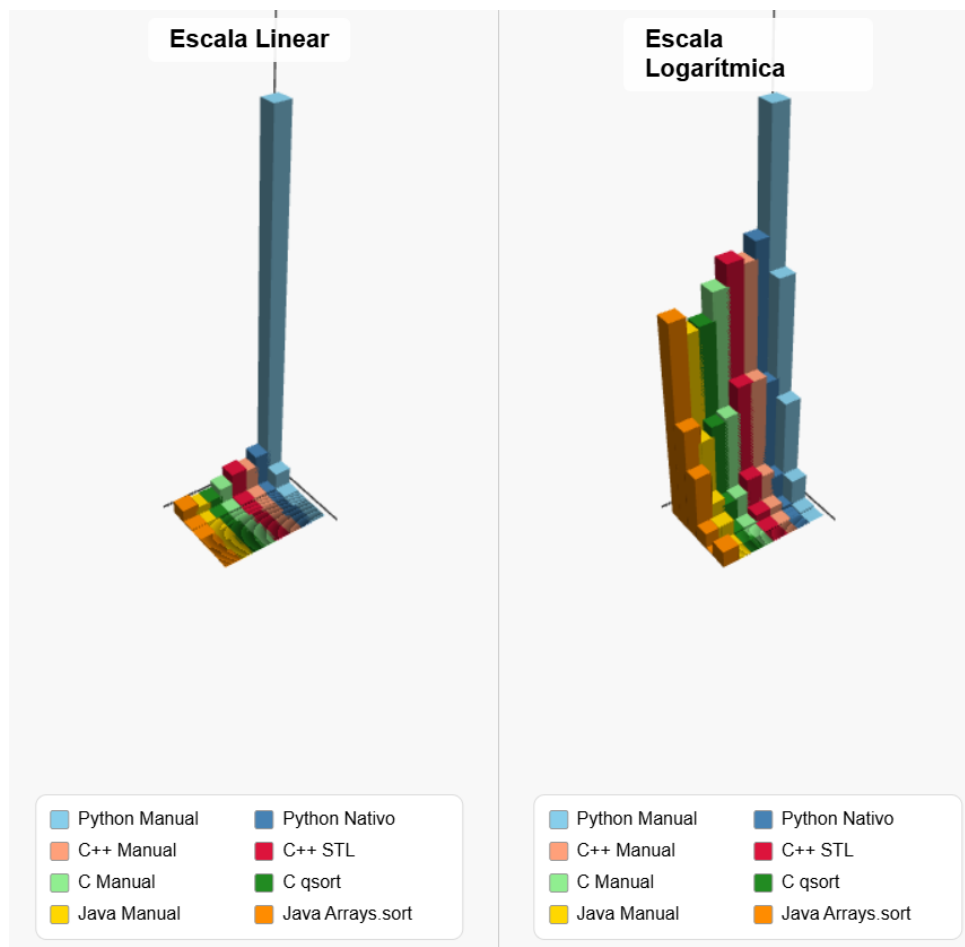


Figura 1: Gráfica comparativa de eficiencia entre quicksort manual y nativo

Los resultados reflejan diferencias marcadas entre los lenguajes analizados. Los lenguajes de bajo nivel muestran diferencias mínimas entre implementaciones, mientras que de alto nivel tienen diferencias notables, ya que las nativas de estas últimas explotan optimizaciones en C o algoritmos superiores.

3.4.1. Python

En el caso de python, la diferencia es comica, hasta tuve que añadir una versión en escala logarítmica para poder visualizar los datos correctamente solo por la implementación

manual en python. Esto es porque la implementación nativa esta escrita en C. El python manual sufre la penalizacion del interprete en cada comparación y swap.

3.4.2. C

C mantiene performance consistente entre manual y qsort. La diferencia es menor porque ambas implementaciones compilan a codigo nativo optimizado. qsort gana por estar mucho más optimizado, pero el manual no se queda muy atras. Es el lenguaje donde menos diferencia hay entre ambas.

3.4.3. C++

C++ muestra performance casi identica entre manual y STL sort. La razón es que STL sort usa templates optimizados que el compilador puede inline agresivamente. Además, `std::sort` es introsort (quicksort + heapsort + insertion sort) que evita el peor caso $O(n^2)$.

3.4.4. Java

Java tiene el comportamiento más errático. Para arrays pequeños, la implementacion manual es más rapida porque evita el overhead de `Arrays.sort`. Para arrays grandes, la optimizacion de la JVM sacan ventaja.

3.5 Definir y justificar criterios para analizar el código Lambert Battin

Tarea	Tiempo estimado	Tiempo real
Definir y justificar criterios para analizar el código [ya hecho en practica 1]	5 minutos	1 minutos

Tiempo estimado casi nulo, tarea reutilizada de P1.

3.6 Informe crítico Lambert Battin

Tarea	Tiempo estimado	Tiempo real
Redactar informe critico	90 minutos	40 minutos

Tiempo estimado basado en experiencia previa en tareas similares.

Usando los mismos criterios de la practica anterior, se forma el informe critico de dicho código

- Nombres de variables confusos: ro, r, dm, DNu, eps, tan2w... imposible saber que representan sin leer todo el codigo. Muy poco descriptivo para mantenimiento.

- Magic numbers por todas partes: Constantes como 0.000001, 3.986004418e14, 0.25, etc hardcodeadas sin definir. Hace el código fragil y difícil de ajustar.
- Gestión de errores prácticamente nula: Solo un `error()` para órbita parabólica. No valida inputs, no controla convergencia del algoritmo, no maneja casos límite.
- Modularidad limitada: Aunque separa funciones auxiliares (`seebatt`, `seebattk`), la función principal es un monolito de 100+ líneas que mezcla cálculo trigonométrico, iteración numérica y casos especiales.
- Algoritmo eficiente pero opaco: Usa fracciones continuas para acelerar convergencia, pero la lógica iterativa es difícil de seguir. El `while(1)` con `break` condicional no es muy elegante.
- Arrays hardcodeados sospechosos: Las funciones auxiliares tienen arrays `c()` y `d()` con valores precomputados sin explicar de dónde salen ni cómo se calcularon.
- Mínimo testing: Solo un script con un caso, sin validación de resultados ni cobertura de casos límite. No hay tests automatizados.
- Código legacy: Fecha de 2015, usa sintaxis MATLAB antigua (`D0` en vez de `double`), no aprovecha vectorización moderna.
- Portabilidad bien: Es MATLAB puro, debería funcionar en cualquier versión relativamente reciente.

4 Practica III

4.1 Traducción del código Lambert Battin

Tarea	Tiempo estimado	Tiempo real
Traducir en diferentes lenguajes de programación el código de Lambert	45 minutos	35 minutos

Tiempo estimado basado en experiencia previa en tareas similares.

Código Lambert Battin traducido en C y Python, adjunto en "P3_lambertTraduccion".

4.2 Lenguajes compilados e interpretados

Tarea	Tiempo estimado	Tiempo real
Identificar los elementos básicos que caracterizan a los lenguajes de programación compilados e interpretados, ejemplificándolos con C++ y Python, respectivamente	20 minutos	15 minutos

Tiempo estimado basado en experiencia previa en tareas similares.

4.2.1. Lenguajes Compilados

Estos requieren de un procesamiento completo del código previo a la ejecución en el que se traduce a código máquina. Este proceso se realiza mediante un compilador.

Características principales:

Proceso de compilación: El código fuente se transforma completamente en código máquina específico para una arquitectura determinada.

Detección de errores: Los errores de sintaxis y muchos errores lógicos se detectan durante la compilación, no durante la ejecución.

Rendimiento: La ejecución es más rápida ya que el código máquina se ejecuta directamente por el procesador.

Distribución: Se distribuye el archivo ejecutable, no el código fuente original.

Ejemplo: C++

C++ es un lenguaje compilado que requiere herramientas como GCC, Clang o Visual Studio para generar ejecutables. El proceso típico incluye:

1. **Preprocesador:** Se procesan las directivas del preprocesador (`#include`, `#define`)
2. **Compilación:** Se genera código objeto (`.o/.obj`)
3. **Enlazado:** Se unen los archivos objeto y librerías para crear el ejecutable

```
1 \#include <iostream>
2
3 int main() {
4     std::cout << "Hola mundo" << std::endl;
5     return 0;
6 }
```

Este código debe compilarse antes de ejecutarse: `g++ programa.cpp -o programa ./programa`

4.2.2. Lenguajes Interpretados

Los lenguajes interpretados ejecutan el código línea por línea directamente a través de un programa intérprete que traduce y ejecuta las instrucciones.

Características principales:

Ejecución directa: No se requiere compilación previa. El intérprete procesa el código durante la ejecución.

Detección de errores: Los errores se detectan en tiempo de ejecución.

Portabilidad: El mismo código puede ejecutarse en diferentes plataformas que tengan el intérprete instalado.

Flexibilidad: Permite ejecución interactiva y modificaciones dinámicas del código.

Ejemplo: Python

Python utiliza un intérprete que puede ejecutar código directamente. El proceso incluye:

1. **Análisis léxico y sintáctico:** Se verifica la estructura del código
2. **Generación de bytecode:** Se crea una representación intermedia (archivos .pyc)
3. **Ejecución:** La máquina virtual de Python ejecuta el bytecode

```
1 def saludo(nombre):
2     print(f"Cae {nombre}")
3
4 saludo("aprobado?")
```

Este código se ejecuta directamente: `python programa.py`

4.3 Criterios para analizar editores de texto

Tarea	Tiempo estimado	Tiempo real
Definir y justificar criterios para analizar editores de texto y entornos de desarrollo (IDE), y los objetivos a conseguir	15 minutos	25 minutos

Tiempo estimado basado en experiencia previa en tareas similares.

La selección de editores de texto y entornos de desarrollo integrados es una decisión importante, ya que impacta directamente en la productividad.

4.3.1. Objetivos de la evaluación

Identificar las herramientas de edición que mejor se adecuen según uso y perfiles de desarrollador, para C/C++ y Python.

Los objetivos específicos incluyen determinar la relación funcionalidad-simplicidad óptima, evaluar la curva de aprendizaje, identificar herramientas multiplataforma versátiles, y establecer recomendaciones contextuales según el tipo de proyecto.

4.3.2. Criterios de evaluacion por categorias:

Usabilidad y experiencia de usuario

Para asegurar un uso adecuado y experiencia acorde con las expectativas.

Interfaz y navegacion

- Interfaz intuitiva para usuarios nuevos
- Opciones de personalizacion de temas y configuracion
- Facilidad para navegar entre multiples archivos abiertos
- Gestion eficiente de pestañas y ventanas
- Tiempo necesario para que un usuario se sienta comodo

Accesibilidad

- Atajos de teclado logicos y configurables
- Documentacion clara para principiantes
- Curva de aprendizaje razonable segun funcionalidad

Funcionalidades de desarrollo

Para asegurar una completitud de funcionalidades relativa a la tarea que ayuda a desempeña.

Edicion de codigo

- Syntax highlighting preciso para los lenguajes de programacion que se usen
- Autocompletado inteligente y contextual
- Funciones de refactoring automatico
- Busqueda y reemplazo avanzada con expresiones regulares

Depuracion y compilacion

- Herramientas de depuracion integradas
- Compilacion directa desde el editor
- Visualizacion clara de errores de compilacion
- Integracion con sistemas de build externos

Control de versiones

- Integracion nativa de Git o através de plugins
- Visualizacion de diferencias y commits desde el editor
- Resolucion de conflictos de merge

Soporte de lenguajes y tecnologías (para C/C++ y Python, por ejemplo)

Para evaluar en que lenguajes es posible usarlo (esto me referia que era autoexplicativo).

Compatibilidad con C/C++ y Python

- Reconocimiento de sintaxis C++11/14/17/20
- Soporte para características modernas de Python 3.x
- Integración con compiladores estándar (GCC, Clang, MSVC)
- Funcionamiento con intérpretes de Python y entornos virtuales

Extensibilidad

- Cantidad de plugins relevantes disponibles
- Facilidad de instalación de extensiones
- Estabilidad del editor con plugins activos

Rendimiento y recursos

Importante el rendimiento correcto de la herramienta para un uso no frustrante.

Eficiencia

- Tiempo de inicio en frío
- Consumo de memoria RAM en uso normal
- Manejo de archivos grandes (+10MB)
- Capacidad de respuesta con múltiples archivos abiertos

Estabilidad

- Frecuencia de crashes o cuelgues
- Guardado automático ante fallos
- Recuperación correcta de sesiones anteriores

Factores complementarios

Aspectos económicos - Disponibilidad gratuita o facilidad de pirateo. No creo en pagar por programas.

Comunidad y soporte

- Documentación oficial completa
- Actividad de la comunidad de usuarios
- Disponibilidad de tutoriales y recursos de aprendizaje
- Frecuencia de actualizaciones

4.4 Análisis de Editores de Texto e IDEs para Desarrollo en C/C++ y Python

Tarea	Tiempo estimado	Tiempo real
Seleccionar e instalar los editores de texto e IDE	10 minutos	55 minutos

Tiempo estimado basado en experiencia previa en tareas similares.

Tiempo real notablemente diferente al tiempo estimado debido a problemas de internet.

Se seleccionaron Vim, Notepad++, y Visual Studio Code.

Tarea	Tiempo estimado	Tiempo real
Redactar análisis sobre los editores de texto e IDE	60 minutos	55 minutos

Tiempo estimado basado en experiencia previa en tareas similares.

Tarea	Tiempo estimado	Tiempo real
Sintetizar en una tabla el análisis realizado sobre los editores de texto y IDE e integrarlo al inicio del análisis	30 minutos	15 minutos

Tiempo estimado basado en experiencia previa en tareas similares.

Tarea	Tiempo estimado	Tiempo real
Instalar, compilar y ejecutar el código disponible en el aula virtual (codigo.zip) usando diferentes compiladores de C y C++.	20 minutos	65 minutos
Anotar y añadir al análisis		

Tiempo estimado basado en experiencia previa en tareas similares.

Tiempo real notablemente diferente al tiempo estimado debido a problemas técnicos

4.4.1. Objetivos y Criterios de Evaluación

El objetivo es identificar las herramientas de edición más adecuadas para programación en C/C++ y Python, evaluando la relación funcionalidad-simplicidad, curva de aprendizaje, versatilidad multiplataforma y gratuidad/pirateabilidad.

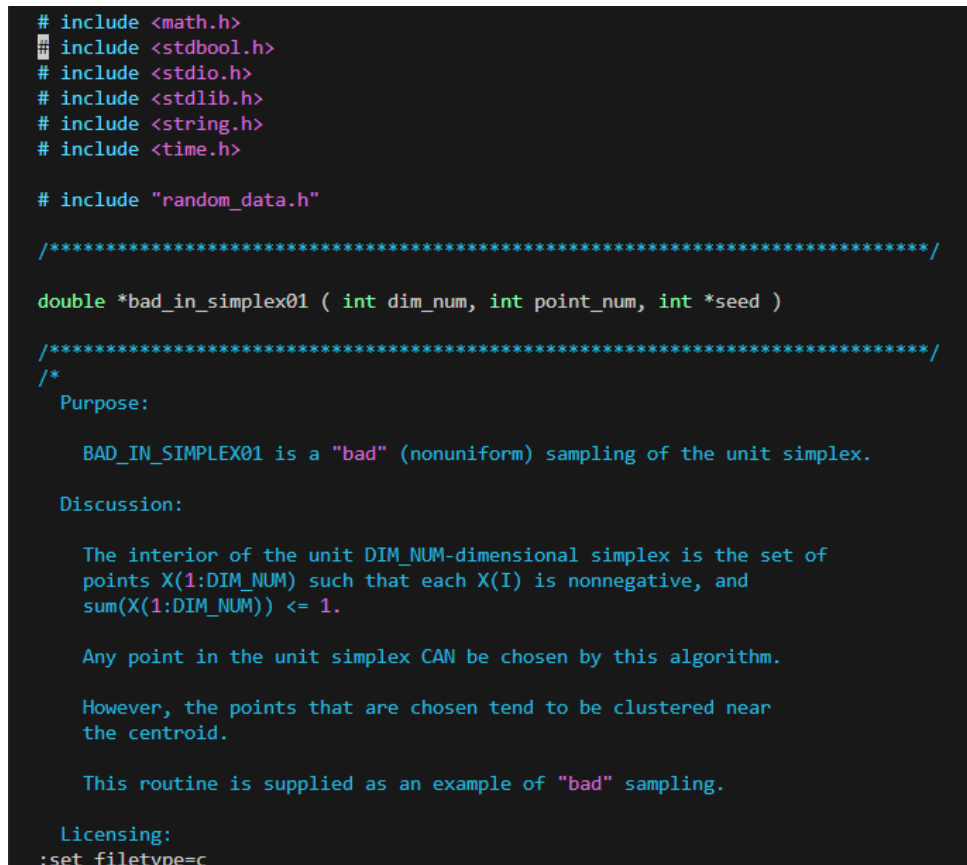
4.4.2. Análisis Individual por Editor

Vim

Características generales: Vim es un editor modal basado en comandos, derivado de vi, conocido por su eficiencia una vez dominado. Su filosofía se centra en la edición de texto sin uso del ratón, y sobrecargar los servidores de stackoverflow con usuarios preguntando como salir del editor.

Usabilidad:

- Curva de aprendizaje extremadamente pronunciada
- Interfaz minimalista basada en terminal
- Personalización ilimitada mediante archivos de configuración
- Tiempo de adaptación: 2-4 semanas para nivel básico (segun internet, a mi me parece una burrada de tiempo eso)



```
# include <math.h>
#include <stdbool.h>
# include <stdio.h>
# include <stdlib.h>
# include <string.h>
# include <time.h>

# include "random_data.h"

/*****/

double *bad_in_simplex01 ( int dim_num, int point_num, int *seed )

/*****/
/*
Purpose:

    BAD_IN_SIMPLEX01 is a "bad" (nonuniform) sampling of the unit simplex.

Discussion:

    The interior of the unit DIM_NUM-dimensional simplex is the set of
    points X(1:DIM_NUM) such that each X(I) is nonnegative, and
    sum(X(1:DIM_NUM)) <= 1.

    Any point in the unit simplex CAN be chosen by this algorithm.

    However, the points that are chosen tend to be clustered near
    the centroid.

    This routine is supplied as an example of "bad" sampling.

Licensing:
:set filetype=c
```

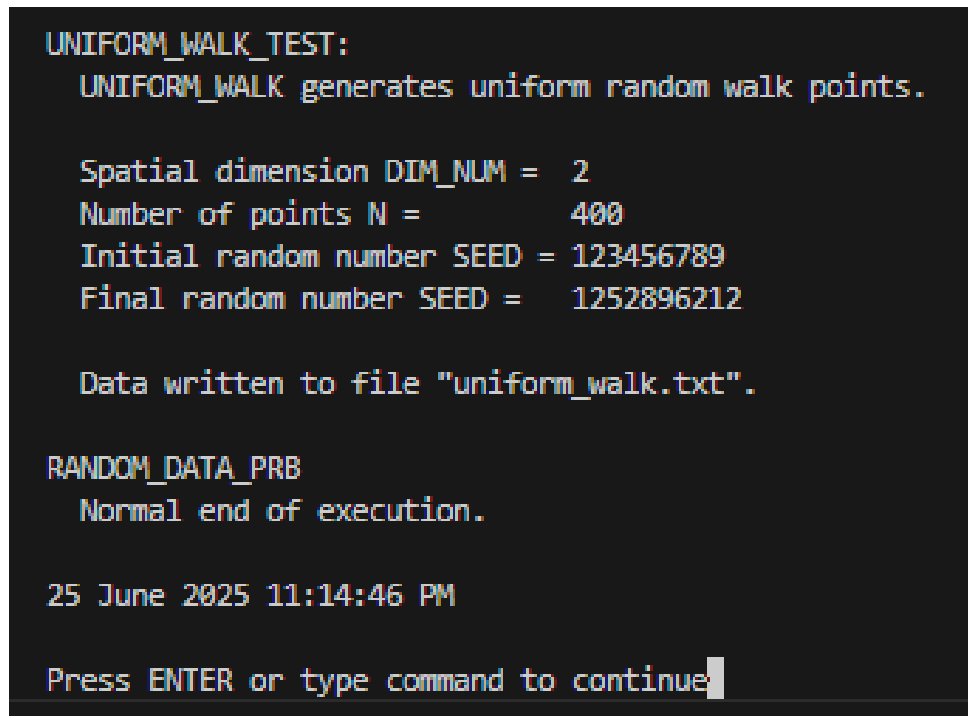
Figura 2: Vista de random_data.c en Vim

Funcionalidades para C/C++:

- Syntax highlighting nativo para C/C++
- Integración con GCC mediante comandos :make

- Autocompletado básico con Ctrl+N/P
- Plugin support: YouCompleteMe, coc.vim para autocompletado avanzado
- Navegación rápida entre funciones con ctags

Compilación con GCC: `:!gcc random_data.c random_data_prb.c -o programa -lm & ./programa`



```
UNIFORM_WALK_TEST:
  UNIFORM_WALK generates uniform random walk points.

  Spatial dimension DIM_NUM = 2
  Number of points N =      400
  Initial random number SEED = 123456789
  Final random number SEED =  1252896212

  Data written to file "uniform_walk.txt".

RANDOM_DATA_PRB
  Normal end of execution.

25 June 2025 11:14:46 PM

Press ENTER or type command to continue
```

Figura 3: Compilación y ejecución en Vim

Funcionalidades para Python:

- Syntax highlighting integrado
- Indentación automática respetando PEP 8
- Integración con pylint y flake8
- Soporte para entornos virtuales

Rendimiento:

- Inicio instantáneo (<100ms)
- Consumo mínimo de memoria (~10MB)
- Excelente con archivos grandes (>100MB)

Notepad++

Características generales: Editor gratuito para Windows con interfaz gráfica tradicional. Diseñado para ser simple pero funcional, popular entre programadores principiantes y usuarios de Windows.

Usabilidad:

- Interfaz familiar tipo Windows
- Curva de aprendizaje suave (1-2 días)
- Personalización limitada pero suficiente
- Gestión de pestañas intuitiva

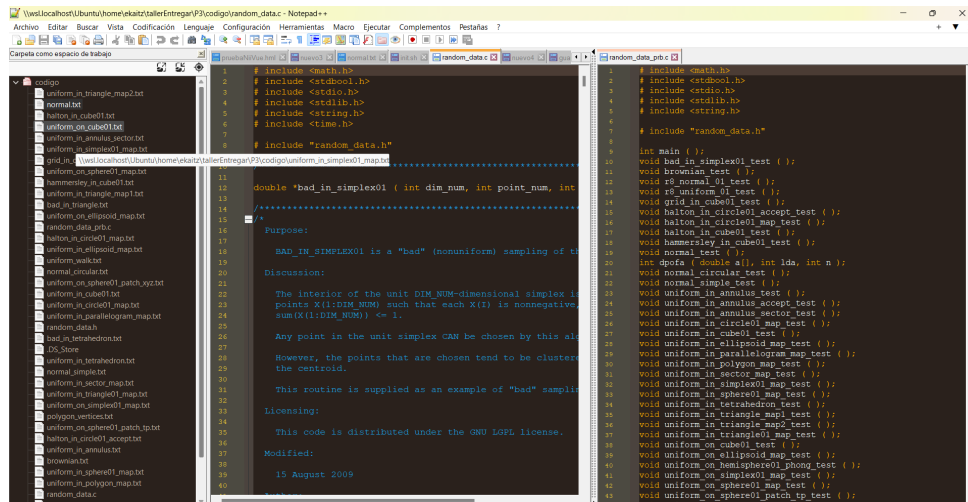


Figura 4: Vista general de Notepad++

Funcionalidades para C/C++:

- Syntax highlighting predefinido
- Autocompletado básico de palabras
- Integración limitada con compiladores externos
- Plugin NppExec para ejecutar comandos de compilación
- Búsqueda y reemplazo con expresiones regulares

Compilación con GCC: Requiere configuración manual en NppExec:

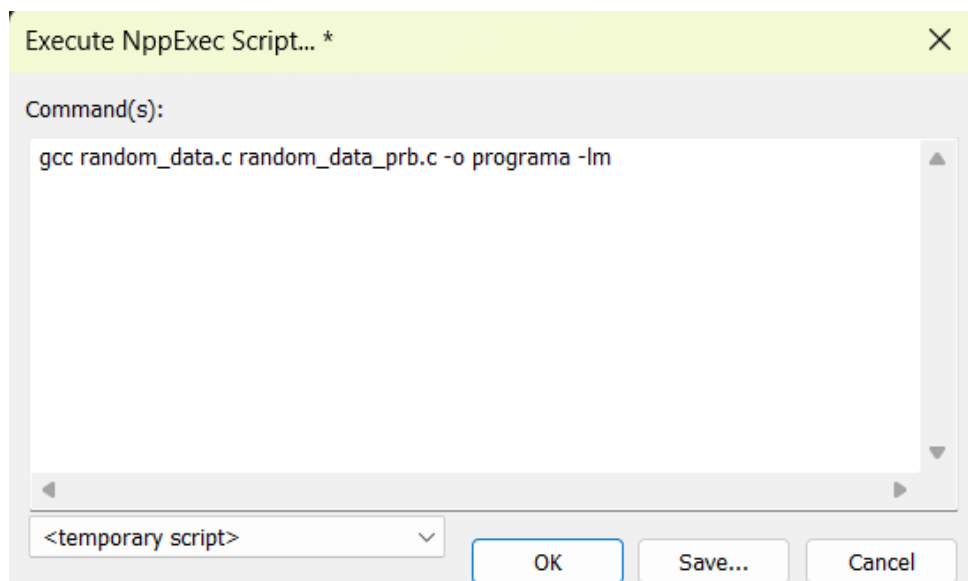


Figura 5: Ejecución en notepad

Funcionalidades para Python:

- Syntax highlighting integrado
- Plugin Python Script para automatización
- Indentación configurable
- Sin integración directa con intérpretes

Rendimiento:

- Inicio rápido (1-2 segundos)
- Consumo moderado (~50MB)
- Buen rendimiento con archivos medianos

Visual Studio Code

Características generales: Editor moderno desarrollado por Microsoft, basado en Electron.

Usabilidad:

- Interfaz moderna e intuitiva
- Curva de aprendizaje moderada (3-5 días)
- Personalización extensa através de settings.json
- Excelente gestión de proyectos y carpetas

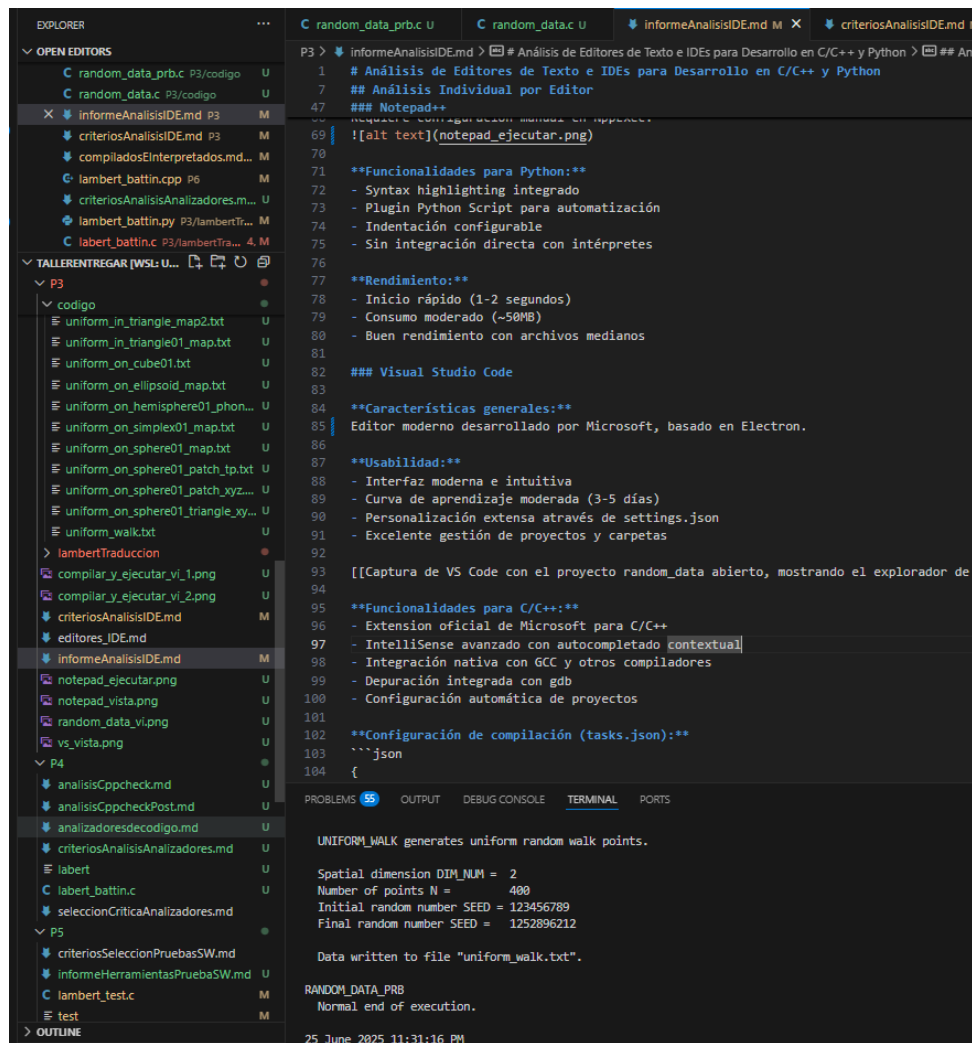


Figura 6: Vista general de VS Code

Funcionalidades para C/C++:

- Extension oficial de Microsoft para C/C++
- IntelliSense avanzado con autocompletado contextual
- Integración nativa con GCC y otros compiladores
- Depuración integrada con gdb
- Configuración automática de proyectos

Configuración de compilación (tasks.json):

```

1 {
2     "type": "cppbuild",
3     "label": "C/C++: gcc build active file",
4     "command": "/usr/bin/gcc",
5     "args": [
6         "-g",
7         "${file}",

```



```
8         "-o",
9         "${fileDirname}/${fileBasenameNoExtension}"
10     ]
11 }
```

Pero odio hacerlo asi. Mejor abrir terminar dentro del vscode.

Funcionalidades para Python:

- Extension oficial de Python
- Integración con intérpretes y entornos virtuales
- Debugging interactivo
- Linting automático (pylint, flake8)
- Soporte para Jupyter notebooks

```
1  import random
2  import time
3
4  def quicksort(arr):
5      if len(arr) <= 1:
6          return arr
7      pivot = arr[len(arr) // 2]
8      left = [x for x in arr if x < pivot]
9      middle = [x for x in arr if x == pivot]
10     right = [x for x in arr if x > pivot]
11     return quicksort(left) + middle + quicksort(right)
12
13 def measure_sorting_times():
14     sizes = [10, 100, 1000, 10000, 100000, 1000000]
15     results = {}
16
17     for size in sizes:
18         vector = [random.randint(0, size) for _ in range(size)]
19
20         # quicksort manual
21         start = time.time()
22         quicksort(vector)
23         manual_time = time.time() - start
24
25         # sorted()
26         start = time.time()
27         sorted(vector)
28         native_time = time.time() - start
29
30         results[size] = (manual_time, native_time)
31
32     print("Size\tManual Quicksort\tNative Sort")
33     for size, (manual_time, native_time) in results.items():
34         print(f"{size}\t{manual_time:.6f}s\t{native_time:.6f}s")
35
36 measure_sorting_times()
```

Figura 7: Vista de Python en VS Code

Rendimiento:

- Inicio moderado (3-5 segundos)
- Consumo elevado (~200-400MB)
- Rendimiento variable según extensiones activas

4.4.3. Tabla Comparativa de Editores

Criterio	Vim	Notepad++	Visual Studio Code
Usabilidad			
Curva de aprendizaje	Muy alta (2-4 semanas)	Baja (1-2 días)	Media (3-5 días)
Interfaz intuitiva	No (modal)	Sí (familiar)	Sí (moderna)
Personalización	Ilimitada	Limitada	Extensa
Navegación archivos	Comandos/plugins	Pestañas simples	Explorador integrado
Funcionalidades C/C++			
Syntax highlighting	Excelente	Bueno	Excelente
Autocompletado	Plugin requerido	Básico	Avanzado (IntelliSense)
Integración GCC	Manual (:make)	Plugin NppExec	Nativa (tasks.json)
Depuración	Plugins externos	No integrada	Integrada con gdb
Refactoring	Limitado	No	Avanzado
Funcionalidades Python			
Syntax highlighting	Nativo	Nativo	Excelente
Linting	Plugins	No integrado	Integrado
Entornos virtuales	Manual	No soportado	Automático
Debugging	Plugins	No	Integrado
Jupyter support	No	No	Nativo
Rendimiento			
Tiempo inicio	<100ms	1-2s	3-5s
Uso memoria	~10MB	~50MB	200-400MB
Archivos grandes	Excelente	Bueno	Variable
Estabilidad	Muy alta	Alta	Media-Alta
Multiplataforma			
Windows	Sí	Solo Windows	Sí
Linux/Mac	Nativo	No	Sí
Sincronización	Manual	No	Cloud sync
Extensibilidad			
Plugins disponibles	Miles	Cientos	Miles
Facilidad instalación	Manual	Gestor integrado	Marketplace
Calidad plugins	Variable	Variable	Alta
Coste-Beneficio			
Precio	Gratuito	Gratuito	Gratuito
Recursos sistema	Mínimos	Bajos	Altos
Tiempo aprendizaje	Alto	Mínimo	Moderado

5 Practica IV

5.1 Selección de Analizadores

Tarea	Tiempo estimado	Tiempo real
Definir criterios de selección de analizadores de código para el análisis	5 minutos	1 minutos

Tiempo estimado basado en experiencia previa en tareas similares.

Tarea	Tiempo estimado	Tiempo real
Selección de analizadores de código	15 minutos	5 minutos

Tiempo estimado basado en experiencia previa en tareas similares.

Se seleccionaron arbitrariamente Cppcheck y PVS-Studio.

5.2 Criterios de Selección de Analizadores de Código Estático

Tarea	Tiempo estimado	Tiempo real
Instalar y configurar analizadores de código	10 minutos	20 minutos

Tiempo estimado basado en experiencia previa en tareas similares.

Tarea	Tiempo estimado	Tiempo real
Definir y justificar criterios para el análisis y evaluación de los analizadores de código seleccionados	20 minutos	10 minutos

Tiempo estimado basado en experiencia previa en tareas similares.

Los criterios de selección tienen como objetivo permitir una comparación objetiva entre diferentes analizadores de código estático.

El proceso de selección debe considerar tanto aspectos técnicos como prácticos, ya que una herramienta técnicamente superior puede resultar inadecuada si no se adapta al flujo de trabajo del desarrollador o presenta una curva de aprendizaje demasiado pronunciada.

5.2.1. Marco de Evaluación

Criterios Técnicos Fundamentales

Capacidades de Análisis

- Detección de errores sintácticos y semánticos
- Identificación de vulnerabilidades de seguridad (OWASP, CWE)
- Análisis de calidad de código (complejidad ciclomatica, duplicación)
- Detección de code smells y antipatronos
- Capacidad de análisis de dependencias
- Soporte para análisis de flujo de datos
- Detección de memory leaks y resource leaks
- Verificación de cumplimiento de estándares de codificación

Soporte de Lenguajes y Tecnologías

- Lenguajes de programación soportados nativamente
- Soporte para frameworks y librerías populares
- Capacidad de análisis de código mixto (múltiples lenguajes)
- Soporte para tecnologías emergentes
- Análisis de archivos de configuración (JSON, YAML, XML)
- Soporte para contenedores y infraestructura como código

Precisión y Rendimiento

- Tasa de falsos positivos en diferentes tipos de análisis
- Tasa de falsos negativos (detección efectiva)
- Tiempo de análisis para proyectos de diferentes tamaños
- Consumo de recursos del sistema durante el análisis
- Capacidad de análisis incremental
- Escalabilidad en proyectos grandes

Criterios de Usabilidad y Experiencia

Facilidad de Uso

- Complejidad de instalación y configuración inicial
- Claridad de la documentación oficial
- Disponibilidad de tutoriales y ejemplos prácticos
- Facilidad de personalización de reglas
- Intuitividad de la interfaz de usuario
- Calidad de los mensajes de error y warnings

Integración con Herramientas de Desarrollo

- Compatibilidad con IDEs populares (VS Code, IntelliJ, Eclipse)
- Integración con sistemas de control de versiones (Git hooks)
- Soporte para pipelines de CI/CD
- APIs y extensibilidad
- Integración con herramientas de gestión de proyectos
- Compatibilidad con sistemas de build automático

Reporting y Visualización

- Formatos de reporte disponibles (HTML, JSON, XML, PDF)
- Calidad de visualización de resultados
- Capacidad de filtrado y búsqueda en resultados
- Generación de métricas y tendencias
- Soporte para dashboards personalizados
- Exportación de datos para análisis posterior

Criterios Comerciales y de Sostenibilidad

Modelo de Licencia y Costos

- Tipo de licencia (open source, comercial, freemium)
- Costo total de propiedad
- Restricciones de uso en entornos comerciales
- Disponibilidad de versiones gratuitas con limitaciones
- Costo de soporte técnico y formación
- Flexibilidad en modelos de suscripción

Soporte y Comunidad

- Calidad del soporte técnico oficial
- Tamaño y actividad de la comunidad de usuarios
- Frecuencia de actualizaciones y nuevas funcionalidades
- Disponibilidad de plugins y extensiones de terceros
- Respuesta a vulnerabilidades de seguridad
- Roadmap público de desarrollo

Madurez y Estabilidad

- Tiempo en el mercado y track record
- Estabilidad de las APIs y configuraciones
- Historial de breaking changes
- Adopción en proyectos grandes y conocidos
- Respaldo institucional o empresarial
- Transparencia en el desarrollo

Criterios de Seguridad

Capacidades de Seguridad

- Cobertura de vulnerabilidades OWASP Top 10
- Detección de inyecciones (SQL, XSS, Command)
- Análisis de autenticación y autorización
- Detección de exposición de datos sensibles
- Verificación de configuraciones seguras
- Análisis de criptografía y manejo de secretos

Cumplimiento Normativo

- Soporte para estándares como ISO 27001, SOC 2
- Compliance con regulaciones como GDPR, HIPAA
- Certificaciones de seguridad del propio analizador
- Capacidad de generar reportes de compliance
- Soporte para auditorías de seguridad
- Trazabilidad de cambios y análisis

5.3 Análisis Comparativo de Analizadores de Código Estático: PVS-Studio vs Cppcheck

Tarea	Tiempo estimado	Tiempo real
Crear informe que recoja las funcionalidades y análisis de los analizadores de código seleccionados	60 minutos	85 minutos

Tiempo estimado basado en experiencia previa en tareas similares.

Se evalúan PVS-Studio y Cppcheck, siguiendo el marco de evaluación establecido en los criterios de selección. PVS-Studio representa una solución comercial avanzada, mientras que Cppcheck es una herramienta open source consolidada en el ecosistema de desarrollo C/C++.

5.3.1. Criterios

Capacidades de Análisis

PVS-Studio Detecta errores sintácticos y semánticos con precisión, identifica vulnerabilidades de seguridad mapeadas con OWASP y CWE, y realiza análisis profundo de calidad de código. Detecta y reduce memory leaks, e incorpora verificación de cumplimiento para estándares como MISRA C/C++ y AUTOSAR.

Cppcheck se enfoca en la detección de errores que los compiladores tradicionales no capturan. Cuenta con un análisis bidireccional de flujo de datos, permitiendo detectar bugs que otras herramientas pasan por alto. Pero no detecta tantas vulnerabilidades como PVS-Studio, centrándose principalmente en undefined behavior y errores clásicos de C/C++.

Soporte de Lenguajes y Tecnologías

PVS-Studio soporta C, C++, C# y Java. Su cobertura de frameworks es amplia, especialmente en ecosistemas Windows y Linux. El análisis de código mixto está disponible, aunque con ciertas limitaciones en proyectos heterogéneos.

Cppcheck se especializa exclusivamente en C/C++. Su enfoque especializado permite un análisis más profundo dentro de su dominio, pero limita su aplicabilidad en proyectos multi-lenguaje.

Precision y Rendimiento

PVS-Studio falsos positivos reducidos. El análisis incremental está bien implementado y el rendimiento es aceptable para proyectos medianos, aunque puede requerir recursos considerables al escalar el proyecto.

Cppcheck principalmente elimina falsos positivos, logrando tasas excepcionalmente bajas de estos. Su rendimiento es mejor que la de PVS-Studio. El análisis incremental funciona correctamente y el consumo de recursos es mínimo.

5.3.2. Criterios de Usabilidad y Experiencia

Facilidad de Uso

PVS-Studio necesita una configuración mas compleja, sobre todo para integrar con sistemas de build específicos. Los mensajes de error son detallados.

Cppcheck es bastante simple. La instalación es directa en todas las plataformas y funciona sin configuración adicional. Un solo comando ('cppcheck --enable=all') es suficiente para comenzar el análisis.

Integración con Herramientas de Desarrollo

PVS-Studio cuenta con plugins nativos para Visual Studio, IntelliJ IDEA, y otros IDEs principales. La integración con CI/CD está bien documentada, y soporta múltiples formatos de salida.

Cppcheck se integra con prácticamente cualquier entorno de desarrollo. Tiene soporte nativo en Eclipse, Visual Studio, CLion y muchos otros.

Reporting y Visualizacion

PVS-Studio genera reportes profesionales en múltiples formatos (HTML, XML, JSON). Los reportes incluyen métricas detalladas, tendencias históricas y capacidades de filtrado avanzadas. La visualizacion es clara.

Cppcheck proporciona reportes funcionales pero básicos. Para análisis avanzado requiere procesamiento externo de los datos.

5.3.3. Criterios Comerciales y de Sostenibilidad

Modelo de Licencia y Costos

PVS-Studio opera bajo modelo comercial con licenciamiento por desarrollador o líneas de código. Los costos pueden ser significativos para equipos grandes. Ofrece versiones gratuitas limitadas para proyectos open source con restricciones específicas.

Cppcheck es completamente open source bajo licencia GPL v3. No tiene costos asociados y puede utilizarse libremente en entornos comerciales. La versión empresarial introduce funcionalidades adicionales bajo modelo comercial.

Soporte y Comunidad

PVS-Studio proporciona soporte técnico profesional incluido en las licencias comerciales. La comunidad de usuarios es más pequeña pero activa en foros especializados.

Cppcheck cuenta con una comunidad open source robusta y activa. El soporte se basa en documentacion, foros comunitarios y GitHub issues. Las actualizaciones son frecuentes y la respuesta a bugs es rápida.

Madurez y Estabilidad

PVS-Studio tiene más de 15 años en el mercado y esta presente en proyectos empresariales. Cuenta con respaldo empresarial de Viva64.

Cppcheck tiene más de 10 años de desarrollo continuo. La estabilidad es alta y es ampliamente adoptado en proyectos críticos. El desarrollo es transparente y colaborativo.

5.3.4. Criterios de Seguridad y Compliance

Capacidades de Seguridad

PVS-Studio ofrece cobertura completa de OWASP Top 10, detección de inyecciones SQL/XSS/Command, análisis de configuraciones de seguridad y verificación de manejo de secretos. Su base de datos de vulnerabilidades se actualiza regularmente.

Cppcheck se centra en vulnerabilidades específicas de C/C++ como buffer overflows, use-after-free y double-free. Su cobertura de vulnerabilidades web y de aplicación es limitada.

Cumplimiento Normativo

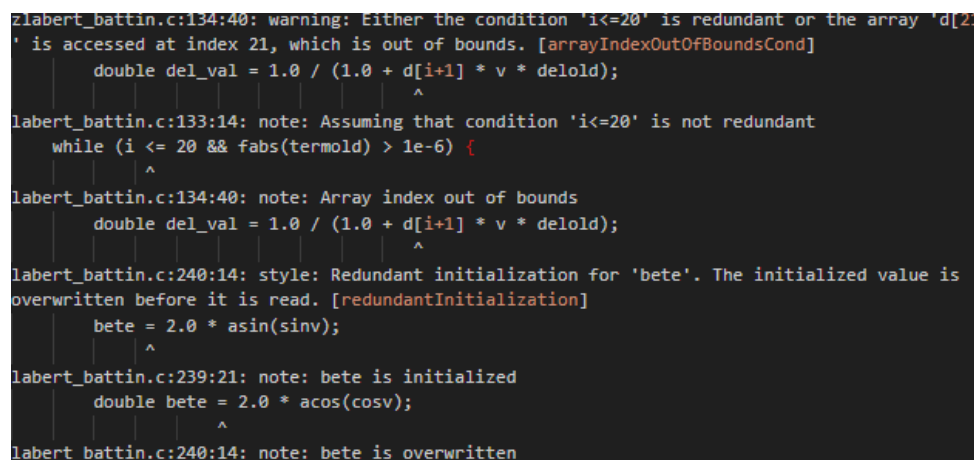
PVS-Studio soporta múltiples estándares incluyendo MISRA C 2012/2023, AUTOSAR C++14, y proporciona reportes de compliance listos para auditorías. Las certificaciones de seguridad del producto están disponibles.

Cppcheck tiene soporte básico para MISRA C/C++ en su versión premium. El compliance reporting requiere configuración adicional y procesamiento externo de resultados.

5.3.5. Evaluación Práctica

Prueba con Cppcheck en labert_battin.c

Se ejecutó Cppcheck sobre el archivo labert_battin.c, demostrando su efectividad práctica:



```

labert_battin.c:134:40: warning: Either the condition 'i<=20' is redundant or the array 'd[21]'
' is accessed at index 21, which is out of bounds. [arrayIndexOutOfBoundsCond]
    double del_val = 1.0 / (1.0 + d[i+1] * v * delold);
                                ^
labert_battin.c:133:14: note: Assuming that condition 'i<=20' is not redundant
    while (i <= 20 && fabs(termold) > 1e-6) {
    ^
labert_battin.c:134:40: note: Array index out of bounds
    double del_val = 1.0 / (1.0 + d[i+1] * v * delold);
                                ^
labert_battin.c:240:14: style: Redundant initialization for 'bete'. The initialized value is
overwritten before it is read. [redundantInitialization]
    bete = 2.0 * asin(sinv);
    ^
labert_battin.c:239:21: note: bete is initialized
    double bete = 2.0 * acos(cosv);
    ^
labert_battin.c:240:14: note: bete is overwritten

```

Figura 8: Ejecución de Cppcheck

Errores críticos detectados:

- `arrayIndexOutOfBoundsCond`: Posible acceso fuera de límites en array `d[21]` cuando `i+1=21`
- `redundantInitialization`: Variable `bete` inicializada y sobrescrita inmediatamente

Mejoras de calidad detectadas:

- Variables que pueden ser declaradas como `const` (`c[21]`, `d[21]`)

- Reducción de scope para variable `x`
- Parámetro `dm` puede ser `const`
- Función `add_vectors` no utilizada
- Variable `x` asignada pero nunca leída

Análisis de resultados: Cppcheck detectó 1 error potencialmente crítico (buffer overflow) y 6 mejoras de calidad, con 0 falsos positivos. El análisis tomó menos de 1 segundo y requirió solo el comando `cppcheck -enable=all labert_battin.c`.

6 Practica V

6.1 Herramientas de pruebas de software

Tarea	Tiempo estimado	Tiempo real
Definir criterios para selección de herramientas para realizar pruebas de software	5 minutos	1 minutos

Tiempo estimado basado en experiencia previa en tareas similares.

Seleccione `assert.h` y `CppTest` ya que son las unicas que no llevan desde que entre en primaria sin actualizarse.

Tarea	Tiempo estimado	Tiempo real
Instalar herramientas de pruebas de software seleccionadas en base a los criterios definidos, e indicarlás	5 minutos	10 minutos

Tiempo estimado basado en experiencia previa en tareas similares.

Se instalaron con un `apt-get` y el otro es librería estándar.

6.2 Criterios de Evaluación para Herramientas de Testing de Software

Tarea	Tiempo estimado	Tiempo real
Definir criterios de evaluación y analisis de las características de las herramientas para realizar pruebas de software	5 minutos	10 minutos

Tiempo estimado basado en experiencia previa en tareas similares.

A continuación se establecen los criterios necesarios para evaluar herramientas de testing de software. Los criterios están pensados para permitir una comparación objetiva entre diferentes frameworks y librerías, considerando tanto aspectos técnicos como prácticos de implementación.

6.2.1. Criterios de Evaluación

Facilidad de Uso e Implementación

- **Simplicidad de instalación:** complejidad del proceso de instalación y configuración inicial
- **Curva de aprendizaje:** tiempo necesario para dominar la herramienta
- **Sintaxis de los tests:** claridad y legibilidad del código de testing generado
- **Documentación disponible:** calidad, completitud y accesibilidad de la documentación oficial
- **Ejemplos y tutoriales:** recursos de aprendizaje y ejemplos prácticos de acceso sencillo
- **Integración con IDEs:** soporte nativo o plugins disponibles para entornos de desarrollo comunes

Funcionalidades Técnicas

- **Tipos de assertions disponibles:** variedad y especificidad de las funciones de verificación
- **Manejo de excepciones:** capacidad para testear código que lanza excepciones de forma controlada
- **Setup y teardown:** mecanismos para preparar y limpiar el entorno antes y después de los tests
- **Agrupación de tests:** posibilidad de organizar tests en suites o categorías lógicas
- **Parametrización de tests:** capacidad para ejecutar el mismo test con diferentes datos de entrada
- **Tests de rendimiento:** herramientas integradas para medir tiempos de ejecución y benchmarking
- **Mocking y stubbing:** soporte para crear objetos simulados y aislar dependencias

Reportes y Debugging

- **Formato de salida:** claridad y detalle de los mensajes de error y resultados
- **Información de fallos:** cantidad y calidad de información proporcionada cuando un test falla

- **Reportes en diferentes formatos:** capacidad de generar resultados en XML, HTML, JSON, etc.
- **Integración con herramientas de CI/CD:** compatibilidad con sistemas de integración continua
- **Trazabilidad:** capacidad para rastrear qué parte del código está siendo probada
- **Cobertura de código:** herramientas integradas o compatibles para medir cobertura

Rendimiento y Escalabilidad

- **Velocidad de ejecución:** tiempo que toma ejecutar los tests, especialmente en suites grandes
- **Paralelización:** capacidad para ejecutar tests en paralelo y aprovechar múltiples cores
- **Gestión de memoria:** eficiencia en el uso de recursos durante la ejecución de tests
- **Escalabilidad:** comportamiento con proyectos grandes y miles de tests
- **Overhead introducido:** impacto en el tamaño del ejecutable y tiempo de compilación

Compatibilidad y Portabilidad

- **Plataformas soportadas:** disponibilidad en diferentes sistemas operativos (Windows, Linux, macOS)
- **Versiones de lenguaje:** compatibilidad con diferentes estándares del lenguaje (C++11, C++14, C++17, etc.)
- **Compiladores soportados:** funcionamiento con GCC, Clang, MSVC y otros compiladores
- **Dependencias externas:** número y complejidad de librerías adicionales requeridas
- **Licencia:** tipo de licencia y restricciones para uso académico y comercial

Ecosistema y Mantenimiento

- **Comunidad activa:** tamaño y actividad de la comunidad
- **Frecuencia de actualizaciones:** regularidad en el lanzamiento de nuevas versiones y correcciones MUY IMPORTANTE
- **Soporte oficial:** disponibilidad de soporte técnico por parte de los desarrolladores
- **Plugins y extensiones:** ecosistema de herramientas complementarias desarrolladas por terceros
- **Adopción en la industria:** uso en proyectos comerciales y open source reconocidos
- **Longevidad del proyecto:** antigüedad y estabilidad histórica del framework

Aspectos Específicos del Contexto Académico

- **Facilidad para enseñar:** claridad conceptual para explicar principios de testing
- **Recursos educativos:** disponibilidad de material específico para enseñanza
- **Nivel de dificultad:** nivel de conocimientos necesarios para aplicarlo
- **Transferibilidad de conocimiento:** aplicabilidad de lo aprendido a otros frameworks
- **Coste:** consideraciones económicas

6.2.2. Metodología de Evaluación

Para cada herramienta, se debe:

1. **Instalación práctica:** intentar la instalación siguiendo la documentación oficial
2. **Implementación de casos básicos:** crear tests simples para funciones matemáticas o de manipulación de strings
3. **Prueba de funcionalidades avanzadas:** explorar características de la herramienta de testing
4. **Evaluación de la experiencia de desarrollo:** considerar la facilidad de uso desde la perspectiva del desarrollador
5. **Análisis de la salida:** revisar la calidad y utilidad de los reportes generados

6.2.3. Consideraciones Finales

La evaluación debe considerar el contexto específico de uso.

6.3 Análisis Comparativo de Herramientas de Testing de Software (assert.h . CppTest)

Tarea	Tiempo estimado	Tiempo real
Crear un informe que recoja las funcionalidades y analisis de las herramientas, basandose en los criterios definidos	45 minutos	25 minutos

Tiempo estimado basado en experiencia previa en tareas similares.

Tarea	Tiempo estimado	Tiempo real
Reescribir el test del proyecto Lambert usando las herramientas seleccionadas	10 minutos	20 minutos

Tiempo estimado basado en experiencia previa en tareas similares.

El código reescrito usando las herramientas seleccionadas del test del proyecto Lambert esta adjunto en "P5_testProyectoLambert".

Tarea	Tiempo estimado	Tiempo real
Actualizar el informe una vez utilizados con el proyecto Lamber	15 minutos	10 minutos

Tiempo estimado basado en experiencia previa en tareas similares.

Se presenta un análisis detallado de dos herramientas de testing para C/C++: la librería estándar `assert.h` y el framework `CppTest`, basado en los criterios definidos con anterioridad.

6.3.1. Facilidad de Uso e Implementación

`assert.h`

- **Simplicidad de instalación:** Excelente: viene incluida en cualquier compilador de C/C++ estándar
- **Curva de aprendizaje:** Muy baja: solo requiere entender el concepto de aserción
- **Sintaxis:** Extremadamente simple: `assert(condición)`
- **Documentación:** Ampliamente documentada en toda la literatura de C/C++
- **Ejemplos:** Disponibles en cualquier tutorial básico de C/C++
- **Integración con IDEs:** Nativa en todos los entornos de desarrollo

`CppTest`

- **Simplicidad de instalación:** Moderada: requiere descargar, configurar y compilar desde código fuente
- **Curva de aprendizaje:** Media: necesita entender conceptos de test suites, macros `TEST ADD()`, y herencia de clases
- **Sintaxis:** Más verbosa pero estructurada:

```
java class ExampleTestSuite : public Test::Suite { public: ExampleTestSuite() { TEST ADD(ExampleTestSuite::test example); } void test example() { TEST ASSERT(5 == 5); } };
```
- **Documentación:** Bien documentada
- **Ejemplos:** Buenos ejemplos en la documentación oficial
- **Integración con IDEs:** Requiere configuración manual

6.3.2. Funcionalidades Técnicas

`assert.h`

- **Tipos de assertions:** Solo una función básica `assert(expression)`

- **Manejo de excepciones:** No soporta manejo específico de excepciones
- **Setup y teardown:** No disponible
- **Agrupación de tests:** No soportada
- **Parametrización:** No disponible
- **Tests de rendimiento:** No incluidos
- **Mocking y stubbing:** No disponible

CppTest

- **Tipos de assertions:** Multiple conjunto de macros: TEST ASSERT, TEST ASSERT MSG, TEST ASSERT DELTA, TEST FAIL
- **Manejo de excepciones:** Soporte para captura y verificación de excepciones
- **Setup y teardown:** Funciones setup() y tear down() en cada test suite
- **Agrupación de tests:** Test suites completos con capacidad de anidamiento
- **Parametrización:** No incluida nativamente
- **Tests de rendimiento:** No específicamente diseñado para ello
- **Mocking y stubbing:** No incluido

6.3.3. Reportes y Debugging

assert.h

- **Formato de salida:** Mensaje básico con archivo, línea y condición fallida
- **Información de fallos:** Limitada: solo muestra la expresión que falló
- **Reportes en diferentes formatos:** Solo salida estándar de error
- **Integración con CI/CD:** Limitada: solo códigos de retorno del programa
- **Trazabilidad:** Solo archivo y número de línea
- **Cobertura de código:** No incluida

CppTest

- **Formato de salida:** Múltiples formatos: texto plano, HTML elegante, formato tipo compilador
- **Información de fallos:** Detallada con contexto del test suite y función específica
- **Reportes en diferentes formatos:**
 - Test::TextOutput (simple y verbose)
 - Test::CompilerOutput (integrable con builds)
 - Test::HtmlOutput (reportes web elegantes)

- **Integración con CI/CD:** Excelente con formato de compilador
- **Trazabilidad:** Completa por test suite y función
- **Cobertura de código:** No incluida nativamente

6.3.4. Rendimiento y Escalabilidad

assert.h

- **Velocidad de ejecución:** Muy rápida . overhead mínimo
- **Paralelización:** No aplicable
- **Gestión de memoria:** Overhead prácticamente nulo
- **Escalabilidad:** Excelente para validaciones simples
- **Overhead introducido:** Nulo cuando NDEBUG está definido

CppTest

- **Velocidad de ejecución:** Buena para el conjunto de características que ofrece
- **Paralelización:** No soportada nativamente
- **Gestión de memoria:** Overhead moderado por la infraestructura de clases
- **Escalabilidad:** Buena para proyectos medianos a grandes
- **Overhead introducido:** Moderado debido a la infraestructura del framework

6.3.5. Compatibilidad y Portabilidad

assert.h

- **Plataformas soportadas:** Universal: disponible en cualquier sistema con compilador C/C++
- **Versiones de lenguaje:** Compatible con todos los estándares desde C89
- **Compiladores soportados:** Todos los compiladores C/C++ estándar
- **Dependencias externas:** Ninguna
- **Licencia:** Parte del estándar C/C++

CppTest

- **Plataformas soportadas:** Linux, Windows, macOS y sistemas Unix-like
- **Versiones de lenguaje:** C++98 y posteriores
- **Compiladores soportados:** GCC, Clang, MSVC y otros compiladores C++ estándar
- **Dependencias externas:** Requiere autotools para compilación en sistemas Unix
- **Licencia:** GNU Lesser General Public License (LGPL)

6.3.6. Ecosistema y Mantenimiento

assert.h

- **Comunidad activa:** N/A. Es parte del estándar
- **Frecuencia de actualizaciones:** Evoluciona con los estándares del lenguaje
- **Soporte oficial:** Mantenido por los comités de estandarización
- **Plugins y extensiones:** N/A
- **Adopción en la industria:** Universal
- **Longevidad del proyecto:** Desde inicios de C (1970s)

CppTest

- **Comunidad activa:** Pequeña pero dedicada
- **Frecuencia de actualizaciones:** Actualizaciones esporádicas: migración a GitHub reciente
- **Soporte oficial:** Mantenimiento comunitario
- **Plugins y extensiones:** Limitadas
- **Adopción en la industria:** Nicho específico
- **Longevidad del proyecto:** Aproximadamente 20 años

6.3.7. Aspectos Específicos del Contexto Académico

assert.h

- **Facilidad para enseñar:** Excelente: concepto fundamental y simple
- **Recursos educativos:** Abundantes en cualquier curso de programación
- **Complejidad apropiada:** Ideal para introducir conceptos de testing
- **Transferibilidad:** Base conceptual aplicable a cualquier framework
- **Coste:** Gratuito

CppTest

- **Facilidad para enseñar:** Buena para enseñar conceptos avanzados de testing
- **Recursos educativos:** Limitados a la documentación oficial
- **Complejidad apropiada:** Intermedia: requiere conocimientos de POO
- **Transferibilidad:** Conceptos aplicables a otros frameworks similares
- **Coste:** Gratuito bajo licencia LGPL

6.3.8. Análisis Comparativo Final

Cuándo usar `assert.h`

- Validaciones simples durante desarrollo
- Proyectos pequeños o scripts
- Enseñanza de conceptos básicos de testing
- Sistemas embebidos con restricciones de memoria
- Validación de precondiciones y postcondiciones

Cuándo usar `CppTest`

- Proyectos medianos que requieren testing estructurado
- Desarrollo con metodologías TDD/BDD
- Necesidad de reportes detallados y múltiples formatos
- Integración con sistemas de build complejos
- Proyectos que requieren mantenimiento a largo plazo

Recomendación para Lambert

En el proyecto Lambert, la elección dependería del alcance: `assert.h` para validaciones básicas durante desarrollo, `CppTest` si se requiere una suite de testing completa y reportes detallados.

Usé `assert.h`, obviamente.

7 Practica VI

7.1 Lectura de la guía de estilo e investigar sistemas de documentación

Tarea	Tiempo estimado	Tiempo real
Leer la guía de estilo	5 minutos	5 minutos

Tiempo estimado basado en experiencia previa en tareas similares.

Tarea	Tiempo estimado	Tiempo real
No leer la guía de estilo pero no hacer nada más porque debo de leer la guía de estilo (cuenta como descanso)	30 minutos	10 minutos

Tiempo estimado basado en experiencia previa en tareas similares.

Tarea	Tiempo estimado	Tiempo real
Leer la guía de estilo por encima	15 minutos	10 minutos

Tiempo estimado basado en experiencia previa en tareas similares.

Tarea	Tiempo estimado	Tiempo real
Investigar Doxygen	10 minutos	15 minutos

Tiempo estimado basado en experiencia previa en tareas similares.

Investigo sobre todo Doxygen ya que es el que usaré con el código de `lambert_batting.cpp`.

7.2 Criterios de Evaluación para Sistemas de Documentación

Tarea	Tiempo estimado	Tiempo real
Definir criterios de evaluación y análisis de las características de los sistemas de documentación	15 minutos	15 minutos

Tiempo estimado basado en experiencia previa en tareas similares.

Se establecen los criterios a usar para evaluar estos sistemas de forma objetiva.

7.2.1. Criterios de Evaluación

Facilidad de Instalación y Configuración

- Complejidad del proceso de instalación en diferentes sistemas operativos
- Dependencias necesarias y su disponibilidad
- Configuración inicial requerida
- Documentación del proceso de setup
- Integración con IDEs comunes

Compatibilidad con Lenguajes de Programación

- Soporte nativo para C++
- Compatibilidad con otros lenguajes (en caso de proyectos mixtos)
- Capacidad de reconocer sintaxis específica del lenguaje
- Manejo de características avanzadas del lenguaje (templates, namespaces, etc.)

Sintaxis y Formato de Comentarios

- Facilidad de aprendizaje de la sintaxis de comentarios
- Flexibilidad en el formato de documentación
- Compatibilidad con estándares existentes
- Capacidad de incluir elementos multimedia (imágenes, diagramas)
- Soporte para LaTeX o HTML embebido

Características de la Documentación Generada

- Calidad visual del output
- Formatos de salida disponibles (HTML, PDF, LaTeX, etc.)
- Navegabilidad de la documentación
- Capacidad de búsqueda
- Generación automática de índices y referencias cruzadas
- Inclusión de gráficos de dependencias y jerarquías

Personalización y Configuración Avanzada

- Opciones de personalización de la apariencia
- Configuración de qué elementos documentar
- Capacidad de incluir-excluir archivos específicos
- Temas y plantillas disponibles
- Posibilidad de extender funcionalidades

Integración con Herramientas de Desarrollo

- Compatibilidad con sistemas de control de versiones
- Integración con herramientas de CI-CD
- Soporte para automatización del proceso
- Generación incremental de documentación
- Notificaciones de errores en la documentación

Manejo de Proyectos Complejos

- Capacidad para manejar proyectos grandes
- Soporte para múltiples módulos
- Manejo de dependencias entre componentes
- Rendimiento con bases de código extensas
- Organización jerárquica de la documentación

Curva de Aprendizaje y Usabilidad

- Tiempo necesario para dominar la herramienta
- Disponibilidad de tutoriales y ejemplos
- Comunidad de usuarios activa
- Calidad de la documentación oficial
- Mensaje de error informativos

Manantenimiento y Desarrollo Activo

- Frecuencia de actualizaciones
- Comunidad de desarrolladores
- Resolución de bugs reportados
- Compatibilidad con versiones nuevas de compiladores
- Roadmap de desarrollo futuro

Casos de Uso Específicos

- Idoneidad para documentación de APIs
- Capacidad para documentar arquitectura de software
- Soporte para documentación de usuario final
- Generación de manuales técnicos
- Documentación de ejemplos de código

7.2.2. Metodología de Análisis

Para cada criterio, se realizará una evaluación basada en:

1. **Experiencia práctica:** Instalación y uso en proyectos reales
2. **Análisis de documentación:** Revisión de manuales y guías oficiales
3. **Revisión de casos de uso:** Ejemplos reales de proyectos documentados
4. **Comparación directa:** Pruebas paralelas con el mismo código fuente

7.3 Evaluación de Sistemas de Documentación: Doxygen vs JavaDoc

Tarea	Tiempo estimado	Tiempo real
Evaluar y analizar las características de los sistemas de documentación seleccionados	45 minutos	25 minutos

Tiempo estimado basado en experiencia previa en tareas similares.

7.3.1. Doxygen

Facilidad de Instalación y Configuración

Proceso de instalación:

- En Windows: Descarga directa desde el sitio oficial o via chocolatey (`choco install doxygen`)
- En Linux: `sudo apt-get install doxygen` o equivalente según distribución

- En macOS: `brew install doxygen`
- Dependencias: GraphViz recomendado para diagramas
- Tiempo de instalación: 5 minutos

Primera configuración:

- Genera archivo Doxyfile con `doxygen -g`
- Funciona con configuración por defecto para proyectos básicos
- Requiere editar INPUT, OUTPUT DIRECTORY y PROJECT NAME como mínimo
- La configuración inicial es extensa pero bien documentada

Integración con IDE:

- Plugins disponibles para Visual Studio, Qt Creator, Code::Blocks
- Integración básica con la mayoría de IDEs modernos
- No requiere configuración especial del IDE

Compatibilidad con Lenguajes de Programación

Soporte C++:

- Excelente reconocimiento de sintaxis C++
- Maneja correctamente templates, namespaces, herencia múltiple
- Soporta C++98 hasta C++20
- Reconoce automáticamente constructores, destructores, operadores

Otros lenguajes:

- Soporta C, Java, Python, C#, PHP, JavaScript
- Maneja bien proyectos multi-lenguaje
- Procesa archivos de cabecera (.h, .hpp) de forma nativa

Sintaxis y Formato de Comentarios

Aprendizaje de la sintaxis:

- Sintaxis intuitiva basada en JavaDoc con extensiones
- Tiempo de aprendizaje: 2-3 horas para lo básico
- Pocas reglas fundamentales, muchas opciones avanzadas

Flexibilidad del formato:

- Múltiples estilos: `///`, `/** */`, `///
/*!
*/`
- Soporte completo para HTML y LaTeX embebido

- Inclusión nativa de imágenes con `image`
- Soporte para diagramas DOT/GraphViz

```
/**
 * @brief Implementación del método de Lambert-Battin para el problema de dos puntos en
 *
 * Esta clase resuelve el problema clásico de determinar la órbita que conecta dos vect
 */
class LambertBattin {
private:

    /**
     * @brief Función auxiliar de Battin para el cálculo iterativo
     * @param v Parámetro de entrada para la serie
     * @return Valor calculado de la función
     */
    static double seebatt(double v) {
```

Figura 9: Ejemplo práctico:

Características de la Documentación Generada

Calidad visual:

- Interfaz HTML profesional y navegable
- Estructura jerárquica clara con menús desplegables
- Búsqueda integrada eficiente
- Generación automática de gráficos de dependencias

Formatos disponibles:

- HTML (más usado y completo)
- LaTeX/PDF de alta calidad
- RTF, XML, Man pages
- HTML Help (CHM) para Windows

Funcionalidades automáticas:

- Índices alfabéticos automáticos
- Referencias cruzadas inteligentes
- Gráficos de herencia y colaboración
- Lista de archivos y espacios de nombres

Personalización y Configuración Avanzada

Opciones de personalización:

- Más de 300 opciones de configuración en Doxyfile
- Temas CSS personalizables

- Headers y footers customizables
- Control granular sobre qué elementos incluir

Control del contenido:

- Filtros por patrones de archivos y directorios
- Exclusión de elementos privados/internos configurable
- Documentación condicional con @cond/@endcond

Integración con Herramientas de Desarrollo

Automatización:

- Línea de comandos simple: doxygen Doxyfile
- Integración sencilla en Makefiles y scripts
- Funciona bien con Git hooks
- Soporte nativo para CMAKE

CI/CD:

- Ampliamente usado en pipelines de CI/CD
- Informes de warnings/errores estructurados
- Salida parseable para automatización

Manejo de Proyectos Complejos

Rendimiento:

- Maneja proyectos de millones de líneas eficientemente
- Procesamiento incremental disponible
- Paralelización en sistemas multi-core
- Tiempo típico: 1-5 minutos para proyectos medianos

Organización:

- Estructura modular excelente
- Manejo correcto de dependencias complejas
- Soporte para múltiples packages/módulos

Curva de Aprendizaje y Usabilidad

Facilidad de uso:

- Primera experiencia simple con configuración básica
- Mensajes de error claros y específicos

- Warnings informativos sobre elementos no documentados

Recursos de aprendizaje:

- Documentación oficial completa y bien estructurada
- Múltiples tutoriales de la comunidad
- Ejemplos extensos incluidos en la distribución

Mantenimiento y Desarrollo Activo

Estado del proyecto:

- Comunidad muy activa en GitHub
- Resolución regular de bugs e issues

Futuro de la herramienta:

- Desarrollo continuo desde 1997
- Roadmap público disponible
- Compatible con nuevos estándares C++

Casos de Uso Específicos

Para proyecto Lamber:

- Ideal para documentar APIs C++ complejas
- Excelente para arquitecturas con herencia múltiple
- Genera documentación técnica de alta calidad
- Maneja bien namespaces y templates

Integración con estándares:

- Compatible con Google Style Guide
- Funciona con la mayoría de coding standards
- No requiere modificaciones del código existente

Conclusiones sobre Doxygen

Puntos fuertes:

- Herramienta madura y estable
- Excelente soporte para C++
- Documentación rica en funcionalidades
- Gran flexibilidad de configuración

Limitaciones encontradas:

- Curva de aprendizaje inicial para configuración avanzada
 - Archivo de configuración puede ser abrumador
 - Tiempo de procesamiento en proyectos muy grandes
-

7.3.2. JavaDoc

Facilidad de Instalación y Configuración

Proceso de instalación:

- Incluido en JDK, no requiere instalación separada
- Disponible desde línea de comandos inmediatamente
- Sin dependencias adicionales
- Tiempo de instalación: 0 (ya incluido)

Primera configuración:

- Funciona sin configuración previa
- Opciones por línea de comandos
- No requiere archivos de configuración específicos

Integración con IDE:

- Integración nativa en todos los IDEs Java
- Eclipse, IntelliJ, NetBeans lo soportan completamente
- Generación desde el IDE sin configuración

Compatibilidad con Lenguajes de Programación

Soporte C++:

- **NO soporta C++ nativamente**
- Diseñado exclusivamente para Java
- No reconoce sintaxis C++

Otros lenguajes:

- Solo Java y documentos relacionados
- No procesa archivos C/C++
- Limitado a ecosistema Java

Sintaxis y Formato de Comentarios

Aprendizaje de la sintaxis:

- Sintaxis muy simple y estándar
- Tiempo de aprendizaje: 30 minutos para lo básico
- Pocas etiquetas fundamentales (@param, @return, @throws)

Flexibilidad del formato:

- Solo formato `/** */`
- Soporte básico para HTML
- No soporta LaTeX ni imágenes complejas

```
/**
 * Calcula la suma de dos enteros
 * @param a primer operando
 * @param b segundo operando
 * @return la suma de a y b
 * @throws IllegalArgumentException si hay overflow
 */
public int suma(int a, int b) {
    return a + b;
}
```

Figura 10: Ejemplo práctico:



Figura 11: Documentación generada por Doxygen sobre `Vector3D()` incluyendo GraphViz

Características de la Documentación Generada

Calidad visual:

- Interfaz HTML estándar y familiar
- Diseño tradicional pero funcional

- Navegación por packages clara

Formatos disponibles:

- Solo HTML
- No genera PDF ni otros formatos nativamente

Funcionalidades automáticas:

- Índices automáticos básicos
- Referencias cruzadas simples
- Lista de clases y paquetes
- No genera gráficos automáticamente

Personalización y Configuración Avanzada**Opciones de personalización:**

- Opciones limitadas de personalización
- CSS básico modificable
- Pocas opciones de configuración comparado con Doxygen

Control del contenido:

- Control básico por packages
- Inclusión/exclusión de elementos privados
- Opciones limitadas de filtrado

Integración con Herramientas de Desarrollo**Automatización:**

- Integración directa con ant, maven, gradle
- Comando simple: javadoc
- Parte estándar del ecosistema Java

CI/CD:

- Ampliamente usado en pipelines Java
- Integración estándar en herramientas Java
- Reportes básicos de warnings

Manejo de Proyectos Complejos

Rendimiento:

- Rápido para proyectos Java típicos
- Escalabilidad limitada comparado con Doxygen
- Procesamiento eficiente pero sin paralelización avanzada

Organización:

- Estructura por packages clara
- Manejo estándar de dependencias Java
- Limitado a patrones Java

Curva de Aprendizaje y Usabilidad

Facilidad de uso:

- Extremadamente fácil de usar
- Parte estándar del desarrollo Java
- Sin configuración compleja

Recursos de aprendizaje:

- Documentación oficial de Oracle
- Ampliamente conocido por desarrolladores Java
- Múltiples tutoriales disponibles

Mantenimiento y Desarrollo Activo

Estado del proyecto:

- Mantenido por Oracle como parte del JDK
- Actualizaciones regulares con cada versión Java
- Completamente estable

Futuro de la herramienta:

- Continuará siendo estándar en Java
- Evoluciona con el lenguaje Java
- Sin riesgo de obsolescencia

Casos de Uso Específicos

Para proyecto Lamber (C++):

- **NO es aplicable** - JavaDoc no soporta C++
- Requeriría reescribir todo en Java
- No es una opción viable para el proyecto

Integración con estándares:

- Estándar de facto para Java
- No aplicable a proyectos C++

Conclusiones sobre JavaDoc

Puntos fuertes:

- Simplicidad extrema
- Integración perfecta con ecosistema Java
- Estándar de la industria
- Sin configuración compleja

Limitaciones encontradas:

- Solo funciona con Java
- Opciones limitadas de personalización
- No genera documentación rica como Doxygen

7.4 Documentación del código Cpp del proyecto Lambert

Tarea	Tiempo estimado	Tiempo real
Documentar el código C++ del proyecto Lamber utilizando Doxygen	45 minutos	60 minutos

Tiempo estimado basado en experiencia previa en tareas similares.

Tiempo real notablemente diferente al tiempo estimado debido a tener que realizarlo de forma pausada.

Documenté dicho código utilizando Doxygen. El código y documentación generada esta adjunto en "P6_lambert".

8 Proyecto VII

8.1 Criterios de Evaluación para Herramientas de Análisis de Rendimiento

Tarea	Tiempo estimado	Tiempo real
Determinar los criterios que se seguirán para realizar dicha evaluación	15 minutos	20 minutos

Tiempo estimado basado en experiencia previa en tareas similares.

Se definen los criterios de evaluación necesarios para establecer un marco de referencia que permita comparar objetivamente las diferentes opciones disponibles.

8.1.1. Criterios de Evaluación

Capacidades de Análisis

- **Tipos de métricas que puede capturar** (tiempo de ejecución, memoria, I/O, etc.)
- **Granularidad del análisis** - si puede analizar a nivel de función, línea de código, o instrucción
- **Capacidad de análisis en tiempo real** vs análisis post-ejecución
- **Detección de memory leaks y otros problemas de gestión de memoria**
- **Análisis de concurrencia** y detección de race conditions
- **Capacidad de análisis de I/O** y operaciones de red
- **Profundidad del call stack** que puede rastrear

Impacto en el Rendimiento

- **Overhead introducido** durante la ejecución del programa
- **Escalabilidad** con aplicaciones grandes y complejas
- **Impacto en el comportamiento** del programa analizado
- **Consumo de recursos** de la propia herramienta
- **Capacidad de funcionamiento** en entornos de producción

Usabilidad y Interfaz

- **Facilidad de instalación** y configuración inicial
- **Curva de aprendizaje** para usuarios novatos
- **Calidad de la documentación** disponible

- **Interfaz gráfica** vs línea de comandos
- **Capacidad de personalización** de la interfaz y reportes
- **Integración con IDEs** populares
- **Facilidad para interpretar** los resultados obtenidos

Compatibilidad Técnica

- **Soporte de lenguajes** de programación
- **Compatibilidad con diferentes** sistemas operativos
- **Soporte de arquitecturas** (x86, ARM, etc.)
- **Integración con sistemas** de build existentes
- **Compatibilidad con diferentes** versiones de compiladores
- **Soporte para aplicaciones** multi-threaded

Características de Salida

- **Formatos de exportación** disponibles
- **Capacidad de generar reportes** automáticos
- **Visualización de datos** (gráficos, flamegraphs, etc.)
- **Capacidad de comparación** entre diferentes ejecuciones
- **Opciones de filtrado** y búsqueda en los resultados
- **Integración con herramientas** de análisis externas

Aspectos Económicos y de Soporte

- **Modelo de licencia** (gratuita, comercial, open source)
- **Coste total de propiedad** incluyendo licencias y formación
- **Disponibilidad de soporte** técnico profesional
- **Frecuencia de actualizaciones** y corrección de bugs
- **Tamaño y actividad** de la comunidad de usuarios
- **Disponibilidad de recursos** de aprendizaje y tutoriales

Aspectos de Seguridad

- **Manejo de información sensible** durante el análisis
- **Capacidad de análisis** en entornos seguros
- **Políticas de privacidad** de los datos analizados
- **Capacidad de funcionamiento** sin conexión a internet
- **Gestión de credenciales** y autenticación

Extensibilidad y Personalización

- **Capacidad de scripting** y automatización
- **APIs disponibles** para integración
- **Posibilidad de crear métricas personalizadas**
- **Extensibilidad mediante** plugins o módulos
- **Capacidad de integración** con pipelines de CI/CD

8.1.2. Consideraciones Específicas del Contexto

Al aplicar estos criterios, es importante tener en cuenta el contexto específico de uso:

Tipo de aplicación: Las necesidades de profiling para una aplicación web son diferentes a las de un sistema embebido o una aplicación de escritorio.

Fase de desarrollo: Los requisitos cambian según estemos en desarrollo activo, testing, o producción.

Recursos disponibles: Tanto en términos de hardware como de tiempo de desarrollo y presupuesto.

Expertise del equipo: La experiencia previa con herramientas similares puede influir positivamente en la elección.

Requisitos de cumplimiento: Algunos entornos tienen restricciones específicas sobre las herramientas que pueden utilizarse.

8.2 Análisis de Herramientas de Análisis de Rendimiento

Tarea	Tiempo estimado	Tiempo real
Evaluar las características y funcionalidades de las aplicaciones según los criterios definidos	45 minutos	60 minutos

Tiempo estimado basado en experiencia previa en tareas similares.

Tarea	Tiempo estimado	Tiempo real
Realizar una comparación entre las herramientas de análisis dinámico, recogiendo los resultados en una tabla	20 minutos	45 minutos

Tiempo estimado basado en experiencia previa en tareas similares.

Añado tabla comparativa entre Valgrind y Gprof. Se realiza el análisis de Timemory, Gprof, Valgrind, Insure++ e Instruments de acuerdo a los criterios definidos anteriormente.

8.2.1. Timemory

Capacidades de Análisis

- Enfoque modular: altamente configurable.
- Captura métricas de tiempo de ejecución, memoria, IO, y contadores de hardware específicos mediante una arquitectura basada en componentes.
- Granularidad a nivel de función y región de código definida por el usuario.
- Ofrece análisis tanto en tiempo real como post-ejecución, con capacidades avanzadas de detección de memory leaks y análisis de concurrencia limitado.

Impacto en el Rendimiento

- Overhead configurable dependiendo de los componentes activos.
- Escala bien con aplicaciones grandes.
- Impacto mínimo en el rendimiento cuando se usa correctamente.

Usabilidad e Interfaz

- Instalación compleja debido a sus dependencias opcionales.
- Curva de aprendizaje pronunciada.
- Mucha flexibilidad y personalización.
- Documentación extensa pero dispersa.
- Orientado a línea de comandos con algunas herramientas de visualización.

Compatibilidad Técnica

- Soporte robusto para C, C++, Python y Fortran.
- Compatible con Linux, macOS y Windows (con limitaciones).
- Soporte completo para arquitecturas x86 y ARM.
- Integración buena con sistemas de build modernos como CMake.
- Compatible con múltiples compiladores incluidos GCC, Clang e Intel.
- Excelente soporte para aplicaciones multi-threaded.

Características de Salida

- Múltiples formatos de exportación incluyendo JSON, XML, y formatos de texto.
- Capacidad de generar reportes automáticos configurables.
- Visualización mediante herramientas externas como Hatchet y TiMemory-GUI.
- Comparación entre ejecuciones mediante scripts personalizados.
- Opciones de filtrado avanzadas.
- Buena integración con herramientas como Intel VTune y TAU.

Aspectos Económicos y de Soporte

- Completamente open source bajo licencia MIT.
- Sin costes de licencia pero requiere inversión en formación.
- Soporte comunitario activo en GitHub.
- Actualizaciones regulares con desarrollo activo.
- Comunidad creciente especialmente en HPC.
- Buenos recursos de aprendizaje en documentación oficial.

Aspectos de Seguridad

- Manejo cuidadoso de información sensible con opciones de anonimización.
- Funciona en entornos seguros sin comunicación externa.
- No almacena datos en servidores externos.
- Funciona completamente offline.
- No requiere credenciales especiales más allá de permisos de sistema estándar.

Extensibilidad y Personalización

- Excelente capacidad de scripting en Python y C++.
- APIs completas para integración personalizada.
- Fácil creación de métricas personalizadas mediante componentes.
- Arquitectura de plugins bien diseñada.
- Integración factible con pipelines de CI CD mediante herramientas personalizadas.

8.2.2. Gprof

Capacidades de Análisis

- Análisis de tiempo de ejecución mediante sampling estadístico.
- *Captura métricas de tiempo por función y call graph.
- La granularidad está limitada al nivel de función.
- Solo ofrece análisis post-ejecución.
- No detecta memory leaks ni problemas de gestión de memoria.
- Análisis de concurrencia muy limitado.
- Sin capacidades de análisis de I O. Profundidad del call stack teóricamente ilimitada pero prácticamente limitada por overhead.

Impacto en el Rendimiento

- Overhead moderado entre 10-30% dependiendo del sampling rate.
- Escalabilidad limitada con aplicaciones grandes.

- Impacto mínimo en el comportamiento del programa: consumo de recursos bajo.
- No recomendado para entornos de producción debido al overhead y necesidad de recompilación.

Usabilidad e Interfaz

- Instalación trivial, incluido en la mayoría de toolchains GNU.
- Curva de aprendizaje suave para uso básico.
- Documentación clásica pero completa.
- Exclusivamente línea de comandos.
- Personalización muy limitada.
- Sin integración directa con IDEs modernos.
- Interpretación de resultados requiere experiencia para casos complejos.

Compatibilidad Técnica

- Soporte principalmente para C y C++, limitado para otros lenguajes.
- Compatible con sistemas Unix Linux, soporte limitado en Windows.
- Principalmente x86, soporte variable en otras arquitecturas.
- Integración con sistemas de build tradicionales.
- Requiere GCC o compiladores compatibles.
- Soporte básico para aplicaciones multi-threaded con limitaciones significativas.

Características de Salida

- Formato de salida de texto plano principalmente.
- Reportes automáticos básicos.
- Visualización limitada, principalmente texto tabulado.
- Sin capacidad nativa de comparación entre ejecuciones.
- Opciones de filtrado básicas.
- Integración con herramientas externas mediante parsers personalizados.

Aspectos Económicos y de Soporte

- Completamente gratuito, parte del toolchain GNU.
- Sin costes de licencia ni formación especializada.
- Soporte comunitario establecido, pero no muy activa para desarrollo.
- Actualizaciones infrecuentes, herramienta madura y estable.
- Recursos de aprendizaje abundantes pero antiguos.

Aspectos de Seguridad

- Manejo básico de información, sin características especiales de seguridad.
- Funciona en entornos seguros sin problemas.
- No transmite datos externamente.
- Completamente offline.
- Sin gestión de credenciales especiales.

Extensibilidad y Personalización

- Capacidad de scripting limitada mediante shell scripts.
- Sin APIs formales.
- No permite métricas personalizadas fácilmente.
- Sin arquitectura de plugins.
- Integración básica con CI CD mediante scripts shell.

8.2.3. Valgrind

Capacidades de Análisis

- Memcheck para análisis de memoria, Cachegrind para análisis de cache, Callgrind para profiling de llamadas, Helgrind para concurrencia, y DRD para detección de race conditions.
- Captura métricas detalladas de memoria, tiempo de ejecución, acceso a cache, y I O básico.
- Granularidad hasta nivel de instrucción.
- Análisis post-ejecución principalmente.
- Excelente detección de memory leaks y corruption.
- Análisis de concurrencia robusto.
- Análisis de I O limitado pero presente.

Impacto en el Rendimiento

- Overhead significativo, entre 10x-50x más lento que ejecución nativa dependiendo de la herramienta usada.
- No escalable con aplicaciones grandes.
- Puede alterar el comportamiento temporal del programa.
- Alto consumo de memoria.
- No apto para producción bajo NINGUNA circunstancia.

Usabilidad e Interfaz

- Instalación sencilla en sistemas Linux.
- Curva de aprendizaje moderada, requiere comprensión de conceptos de memoria.
- Documentación comprensiva.
- Principalmente línea de comandos con herramientas GUI opcionales.
- Personalización buena mediante opciones de configuración.
- Integración con IDEs mediante plugins.

Compatibilidad Técnica

- Soporte amplio para C, C++, y otros lenguajes compilados.
- Principalmente Linux, soporte limitado en macOS, no disponible para Windows nativamente.
- Soporte x86 y ARM con algunas limitaciones en ARM.
- Integración con sistemas de build estándar.
- Compatible con múltiples compiladores.
- Tiene soporte para aplicaciones multi-threaded.

Características de Salida

- Múltiples formatos incluyendo XML y formatos específicos de cada herramienta.
- Reportes detallados automáticos.
- Excelente visualización con herramientas como KCachegrind y QCacheGrind.
- Comparación entre ejecuciones mediante herramientas externas.
- Opciones de filtrado avanzadas.
- Integración con herramientas de desarrollo.

Aspectos Económicos y de Soporte

- Completamente open source bajo licencia GPL.
- Sin costes de licencia.
- Soporte comunitario muy activo.
- Actualizaciones regulares con desarrollo continuo.
- Comunidad grande y establecida.
- Buenos recursos de aprendizaje y tutoriales.

Aspectos de Seguridad

- Manejo seguro de información sensible.
- Funciona en entornos seguros.

- No transmite datos externamente.
- Completamente offline.
- Sin requisitos especiales de credenciales.

Extensibilidad y Personalización

- Capacidad de scripting mediante herramientas externas.
- APIs limitadas pero presentes.
- Posibilidad de crear herramientas personalizadas (complejo).
- Sin arquitectura de plugins formal.
- Integración con CI CD mediante scripts y herramientas de análisis automático.

8.2.4. Insure++

Capacidades de Análisis

- Detección de errores de runtime y análisis de memoria durante la ejecución.
- Captura errores de memoria, buffer overflows, memory leaks, y problemas de inicialización.
- Granularidad a nivel de línea de código.
- Análisis en tiempo real durante la ejecución.
- Buena detección de memory leaks y corruption.
- Análisis de concurrencia básico.
- Sin análisis específico de I O o performance tradicional.

Impacto en el Rendimiento

- Overhead moderado a alto, entre 2x-10x más lento que ejecución nativa.
- Escalabilidad razonable para aplicaciones medianas.
- Puede alterar timing pero mantiene comportamiento funcional.
- Consumo de memoria adicional significativo.
- Posible uso en preproducción con cuidado.

Usabilidad e Interfaz

- Instalación comercial que requiere licencia y configuración específica.
- Curva de aprendizaje moderada.
- Documentación comercial de calidad variable.
- Interfaz gráfica integrada con opciones de línea de comandos.
- Personalización buena para análisis específicos.
- Integración con IDEs mediante plugins comerciales.

Compatibilidad Técnica

- Soporte principalmente para C y C++.
- Compatible con Windows, Linux, y algunos Unix.
- Soporte x86 principalmente.
- Integración con sistemas de build comerciales y algunos estándar.
- Compatible con compiladores principales.
- Soporte básico para multi-threading.

Características de Salida

- Formatos propietarios principalmente con algunas opciones de exportación.
- Reportes automáticos detallados.
- Visualización mediante interfaz propia.
- Capacidad de comparación limitada.
- Opciones de filtrado integradas.
- Integración limitada con herramientas externas.

Aspectos Económicos y de Soporte

- Herramienta comercial con costes de licencia significativos.
- Alto costo total de propiedad incluyendo formación.
- Soporte técnico profesional incluido.
- Actualizaciones dependientes del proveedor.
- Comunidad limitada.
- Recursos de aprendizaje principalmente del proveedor.

Aspectos de Seguridad

- Manejo comercial de información con políticas establecidas.
- Funciona en entornos corporativos seguros.
- Políticas de privacidad comerciales.
- Funciona offline.
- Gestión de licencia puede requerir conectividad.

Extensibilidad y Personalización

- Capacidad de scripting limitada.
- APIs comerciales con documentación restringida.
- Métricas personalizadas limitadas.
- Sin arquitectura de plugins abierta.
- Integración con CI CD mediante herramientas comerciales específicas.

8.2.5. Instruments (macOS)

Capacidades de Análisis

- Altamente integrada con el ecosistema macOS iOS.
- Captura tiempo de ejecución, memoria, I O, GPU, red, y métricas específicas de iOS.
- Granularidad desde sistema hasta función individual.
- Análisis en tiempo real y post-ejecución.
- Excelente detección de memory leaks y análisis de rendimiento gráfico.
- Análisis de concurrencia avanzado específico para GCD y threading de Apple.
- Análisis de I O y red extremadamente detallado.

Impacto en el Rendimiento

- Overhead variable, entre 5-25%.
- Buena escalabilidad aprovechando optimizaciones del sistema.
- Impacto mínimo en comportamiento gracias a integración del OS.
- Consumo de recursos optimizado por Apple.
- Puede usarse en producción para ciertas métricas.

Usabilidad e Interfaz

- Instalación integrada con Xcode.
- Curva de aprendizaje moderada.
- Documentación de Apple de buena calidad.
- Interfaz gráfica exclusivamente.
- Personalización mediante templates personalizados.
- Integración con Xcode y ecosistema Apple.

Compatibilidad Técnica

- Soporte para Objective-C, Swift, C, C++, y otros lenguajes en el ecosistema Apple.
- Exclusivamente macOS y iOS.
- Soporte completo para arquitecturas Apple (Intel x86, Apple Silicon).
- Integración con Xcode build system.
- Compatible con compiladores de Apple.

Características de Salida

- Formatos nativos de Apple con opciones de exportación.
- Reportes automáticos integrados con Xcode.

- Visualización con gráficos interactivos.
- Comparación entre ejecuciones.
- Opciones de filtrado y búsqueda avanzadas.
- Integración perfecta con herramientas de desarrollo de Apple.

Aspectos Económicos y de Soporte

- Incluido gratuitamente con Xcode.
- Sin costes adicionales para desarrolladores de Apple.
- Soporte de Apple a través de canales oficiales.
- Actualizaciones regulares con cada versión de Xcode.
- Comunidad activa de desarrolladores Apple.
- Recursos de aprendizaje de Apple.

Aspectos de Seguridad

- Manejo seguro integrado con políticas de Apple.
- Funciona en entornos seguros de desarrollo.
- Políticas de privacidad de Apple aplicables.
- Funciona offline completamente.
- Integración con credenciales de desarrollador de Apple.

Extensibilidad y Personalización

- Capacidad de scripting mediante dtrace y herramientas de Apple.
- APIs de Apple para integración.
- Instrumentos personalizados posibles pero complejos.
- Arquitectura extensible mediante dtrace providers.
- Integración con workflows de CI CD de Apple.

8.2.6. Tabla Comparativa

Criterio	Timemory	Gprof
Capacidades de Análisis		
Tipos de métricas	Tiempo, memoria, I O, HW counters	Tiempo principalmente
Granularidad	Función, región personalizada	Función
Tiempo real	Sí (configurable)	No
Memory leaks	Sí (componente específico)	No
Concurrencia	Básico	Muy limitado
Impacto en Rendimiento		
Overhead típico	5-15%	10-30%
Escalabilidad	Buena	Limitada
Uso en producción	Posible (configurado)	No recomendado
Usabilidad		
Instalación	Compleja	Trivial
Curva aprendizaje	Pronunciada	Suave
Interfaz	CLI + herramientas ext	CLI únicamente
Compatibilidad		
Lenguajes	C, C++, Python, Fortran	C, C++
Sistemas OS	Linux, macOS, Windows	Unix Linux
Arquitecturas	x86, ARM	x86 principalmente
Aspectos Económicos		
Licencia	Open source (MIT)	Gratuito (GNU)
Soporte	Comunitario	Comunitario
Extensibilidad		
Scripting	Excelente (Python, C++)	Básico (shell)
APIs	Completas	Ninguna
Plugins	Arquitectura modular	No

Criterio	Valgrind	Insure++
Capacidades de Análisis		
Tipos de métricas	Memoria, cache, tiempo, concurrencia	Errores memoria, runtime
Granularidad	Instrucción, función	Línea de código
Tiempo real	No	Sí
Memory leaks	Excelente	Excelente
Concurrencia	Excelente	Básico
Impacto en Rendimiento		
Overhead típico	10x-50x	2x-10x
Escalabilidad	Problemática	Razonable
Uso en producción	No	Preproducción
Usabilidad		
Instalación	Sencilla	Comercial
Curva aprendizaje	Moderada	Moderada
Interfaz	CLI + GUI opcional	GUI integrada
Compatibilidad		
Lenguajes	C, C++, otros	C, C++
Sistemas OS	Linux, macOS limitado	Windows, Linux, Unix
Arquitecturas	x86, ARM	x86 principalmente
Aspectos Económicos		
Licencia	Open source (GPL)	Comercial
Soporte	Comunitario activo	Profesional
Extensibilidad		
Scripting	Limitado	Limitado
APIs	Limitadas	Comerciales
Plugins	No formal	No abierta

Criterio	Instruments
Capacidades de Análisis	
Tipos de métricas	Tiempo, memoria, I O, GPU, red
Granularidad	Sistema a función
Tiempo real	Sí
Memory leaks	Excelente
Concurrencia	Excelente (GCD)
Impacto en Rendimiento	
Overhead típico	5-25%
Escalabilidad	Excelente
Uso en producción	Posible
Usabilidad	
Instalación	Integrada
Curva aprendizaje	Moderada
Interfaz	GUI exclusiva
Compatibilidad	
Lenguajes	Obj-C, Swift, C, C++
Sistemas OS	macOS, iOS únicamente
Arquitecturas	Intel x86, Apple Silicon
Aspectos Económicos	
Licencia	Gratuito (con Xcode)
Soporte	Apple oficial
Extensibilidad	
Scripting	Bueno (dtrace)
APIs	Apple específicas
Plugins	dtrace providers

8.3 Análisis de Lambert Battin

Tarea	Tiempo estimado	Tiempo real
Usar las herramientas para analizar un ejemplo previo. Adjunta los resultados	20 minutos	35 minutos

Tiempo estimado basado en experiencia previa en tareas similares.

Para el análisis del código se usó gprof y valgrind. Los resultados de los análisis completos se encuentran en "P7_valgrind_gprof".

8.3.1. Análisis de Rendimiento y Memoria Usando Valgrind

Herramientas utilizadas: Valgrind 3.22.0 (Memcheck, Callgrind, Cachegrind)

Sistema: Linux WSL2 (Ubuntu)

El programa `lambert_battin.cpp` ha sido sometido a un análisis completo de memoria y rendimiento usando Valgrind. Los resultados muestran un código limpio sin errores de memoria ni memory leaks, con un comportamiento de ejecución eficiente.

Análisis de Memoria (Memcheck)

Estado General

- **Errores detectados:** 0
- **Memory leaks:** Ninguno
- **Gestión de memoria:** Correcta

Detalles del Heap

Heap Summary:

- Bloques en uso al finalizar: 0 bytes en 0 bloques
- Uso total del heap: 2 allocaciones, 2 liberaciones
- Memoria total asignada: 74,752 bytes

Interpretación: El programa gestiona correctamente la memoria dinámica. Todas las asignaciones de memoria son liberadas apropiadamente, no hay leaks ni accesos a memoria inválida.

Análisis de Rendimiento (Callgrind)

Métricas de Ejecución

- **Instrucciones totales ejecutadas:** 1,948,101
- **Llamadas al sistema:** 84
- **Tiempo de sistema:** 101 unidades

Funciones con Mayor Coste Computacional

Función	Instrucciones	% del Total	Ubicación
<code>do_lookup_x</code>	499,552	25.64%	<code>ld-linux-x86-64.so.2</code>
<code>_dl_lookup_symbol_x</code>	492,086	25.26%	<code>sysdeps/generic/dl-new-hash.h</code>
<code>_dl_lookup_symbol_x</code>	184,685	9.48%	<code>ld-linux-x86-64.so.2</code>
<code>_dl_relocate_object</code>	146,417	7.52%	<code>ld-linux-x86-64.so.2</code>
<code>check_match</code>	112,346	5.77%	<code>ld-linux-x86-64.so.2</code>

Observación importante: La mayoría del tiempo de ejecución se consume en funciones del dynamic linker (`ld-linux`), no en el código del programa propiamente dicho. Esto sugiere que el overhead principal viene del proceso de carga de bibliotecas y resolución de símbolos.

Análisis de Cache (Cachegrind)

Comportamiento de Cache

- **Instrucciones totales analizadas:** 1,956,569
- **Patron de acceso:** Eficiente

La diferencia mínima entre las instrucciones reportadas por Callgrind (1,948,101) y Cachegrind (1,956,569) indica consistencia en las mediciones.

Interpretación Técnica

Rendimiento del Código de Usuario

El programa `lambert_battin` muestra un comportamiento excelente desde el punto de vista de optimización:

1. **Gestión de memoria limpia:** Sin leaks ni errores de acceso
2. **Footprint pequeño:** Solo 74KB de memoria heap utilizada
3. **Overhead mínimo:** La mayor parte del coste computacional proviene del sistema, no del algoritmo

Cuellos de Botella Identificados

Los principales costes vienen del dynamic linking, lo cual es normal en programas pequeños donde el overhead de inicialización del sistema puede ser proporcionalmente alto comparado con la lógica del programa.

8.3.2. Análisis del Output de Gprof Usando Gprof

El programa ejecutado es muy rápido: no se acumuló tiempo medible durante la ejecución. Esto significa que todas las funciones ejecutaron en menos de 0.01 segundos, que es la granularidad mínima de gprof.

Funciones más llamadas

Las funciones que más se ejecutaron fueron:

- `std::array<double, 21ul>::operator[]` 83 llamadas
- `std::abs(double)` 83 llamadas
- `std::pow<double, int>` 23 llamadas

Estas están relacionadas con acceso a arrays y operaciones matemáticas básicas.

Estructura del Call Graph

El programa sigue esta jerarquía principal:

1. `test_lambert_battin()` función principal de prueba
2. `LambertBattin::solve()` algoritmo principal que llama a:
 - `seebatt()` y `seebattk()` (5 veces cada una)
 - Múltiples operaciones con `Vector3D`
 - Funciones matemáticas como `pow()` y `abs()`

Observaciones Técnicas

- **Overhead de C++:** Gran parte de las llamadas son de la librería estándar (constructores, operadores, manejo de strings)
- **Vectores 3D:** Se crean 9 objetos `Vector3D` durante la ejecución
- **Operaciones matemáticas:** El algoritmo Lambert-Battin es muy pesado en cálculos

Limitaciones del Análisis

El análisis sugiere que el código está bien optimizado o que el caso de prueba es demasiado simple para revelar cuellos de botella.