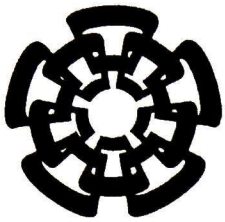
CT-977-SS1
DOV: 7016



Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional
Unidad Guadalajara

# Paralelización del Algoritmo del Método de Descomposición de Adomian y su Implementación en una GPU

Tesis que presenta:
**Gerardo Ramírez Arredondo**

para obtener el grado de:
**Maestro en Ciencias**

en la especialidad de:
**Ingeniería Eléctrica**

Director de Tesis
**Dr. Deni Librado Torres Román**

CINVESTAV del IPN Unidad Guadalajara, Guadalajara, Jalisco, Agosto de 2015.

# Paralelización del Algoritmo del Método de Descomposición de Adomian y su Implementación en una GPU

**Tesis de Maestría en Ciencias**
**Ingeniería Eléctrica**

Por:
**Gerardo Ramírez Arredondo**
Ingeniero en Electrónica
Universidad Michoacana de San Nicolás de Hidalgo 2008-2013

Becario de CONACYT, expediente no. 300971

Director de Tesis
**Dr. Deni Librado Torres Román**

CINVESTAV del IPN Unidad Guadalajara, Agosto de 2015.

# Agradecimientos

A mi hermano, Oscar, que me sigue acompañando en cada paso que doy y aún encuentra la manera de guiarme cuando me siento perdido.
A mis papás Gerardo y Guadalupe por creer en mi, incluso cuando yo no lo hice, por ayudarme a alcanzar mis metas y seguir ahí para las que faltan.
A mi tía Juany, mi tío Carlos y a Karlita por invitarme a su familia y hacerme sentir parte de ella.
A mi tío Carlos Arredondo por sus palabras de aliento y haberse convertido en un modelo a seguir.
A la familia Montenegro que siempre me apoya incondicionalmente y me enseñó que se puede tener todo pero no sirve de nada si no tienes con quien compartirlo.
A mi abuelita Rosa por sus consejos y enseñarme su forma particular de ver la vida.
A mi novia, Andrea, que cree en mis locuras, siempre me apoya y me ha ayudado a levantarme cuando nadie más vio que había caído.
A Jonathan, por su incomparable amistad y por sus buenos consejos de como afrontar la vida.
A Tele 13, la mejor generación que se pudo haber tenido. Leandro, Paquito, Shaides y Pablo.
Al gran maestro Charlie por su amistad y sus enseñanzas.
A mi asesor, el Dr. Deni Librado Torres Román por el conocimiento impartido, su apoyo y confianza.
A Daniel Robles por su ayuda y paciencia durante el desarrollo de la tesis.
A todos los Doctores que fueron parte de mi formación durante la maestría.
A mis amigos del área de telecomunicaciones.
A CONACyT por el apoyo económico brindado.

# Resumen

Aunque siempre, el modelado de la realidad se ha convertido en algo muy común en el campo de la ciencia, debido a la naturaleza del ser humano por querer entender lo que nos rodea, en las últimas décadas ha crecido vertiginosamente apoyado por la teoría de la computación. Como, no todo lo que nos rodea se comporta de manera lineal, por tanto es necesario analizar problemas no lineales, lo cual es complicado sin suponer linealidad. El método de descomposición de Adomian permite modelar problemas no lineales, sin la necesidad de suponer linealidad, dando así una aproximación más cercana al modelo real. Se tiene una gran cantidad de problemas no lineales, por tanto es conveniente buscar una manera de evaluarlos buscando que el costo computacional no sea alto comparado con otros métodos, así como aprovechar las nuevas herramientas disponibles, como por ejemplo los GPUs. Se propone tomar un problema no lineal, como es el calculo de los eigenvalores de una matriz o encontrar las raíces de un polinomio. Tomando esto como un primer paso, para posteriormente extender el método a diferentes tipos de ecuaciones no lineales, como logarítmicas, exponenciales, diferenciales, etc.

# Abstract

Modeling of reality has become something really common on the science field, due to the nature of the human being, wanting to understand their surroundings, in the last decades it has grown dizzily supported by theory of communication. Not everything that is around us behaves in a linear way, therefore is necessary to analyze nonlinear problems, which is complicated without assuming linearity. The Adomian decomposition method allows to model nonlinear problems, without the need of assuming linearity, giving a closer approximation to the real model. There is a great amount of nonlinear problems, therefore is convenient to seek a way to evaluate them taking in consideration the computational not being that high compared with other methods, also using new available tools, like GPUs. It is proposed to take a nonlinear problem, for example the calculation of eigenvalues of a matrix or finding the roots of a polynomial. Taking this as a first step, for extending the method to different types of nonlinear ecuations, like logarithmics, exponentials, differential, etc.

# Tabla de Contenido

# Índice de figuras

# Índice de tablas

# Capítulo 1

# Introducción

A lo largo de la historia, las personas se ha preocupado por el mundo que los rodea, entenderlo y poder modelarlo es una de las formas más eficaces para su estudio y compresión. El problema ha surgido cuando la realidad no es tan simple para ser modelada de manera lineal, hay muchas variables que tomar en cuenta y hay ocasiones que su compartamiento no obedece modelos lineales. Por tanto se han hecho modelos no-lineales de los problemas y buscado de la forma de resolverlos o conocer su respuesta hacía ciertos estímulos que ayudarán a entenderlo. Uno de los problemas que surgen, es que resolver modelos no-lineales o encontrar su comportamiento es complicado, entonces se busca la forma de transformalos en un problema que sea fácil de resolver, tal vez suponiendo en durante cierto rango se comporta de manera lineal, aunque al hacerlo la información que se obtenga no sea del todo precisa con respecto al modelo real, pero útil para ciertas aplicaciones. Siempre se quiere conocer el comportamiento más cercano al real, lo cual generalmente conlleva a utilizar recursos que son computacionalmente costosos.

Existen varias formas de resolver este tipo de problemas, una de ellas son las aplicaciones a nivel de Hardware, las cuales están hechas de manera rígida para el tipo de algoritmo que se implementa. Como por su naturaleza el Hardware es concurrente se pueden realizar algunas partes del algoritmo de manera paralela. Este tipo de diseño requiere de una mayor cantidad de tiempo para ser implementado que usar una herramienta ya hecha, por ejemplo las unidades de procesamiento gráfico (GPUs) diseñadas de tal forma que son capaces de realizar operaciones y procesos

en paralelo sin tener que dedicar mucho tiempo para programar su ejecución.

Una de ellas, son las tarjetas NVIDIA, las cuales han avanzando mucho en cuanto a desempeño, velocidad, cantidad de memoria y la información de uso es muy accesible, así como la cantidad de foros donde se pueden consultar dudas y problemas que surjan durante el proceso.

## 1.1. Estado del arte

El método de descomposición de Adomian permite encontrar soluciones a diferentes tipos de ecuaciones no lineales. Una de sus ventajas es que puede proveer una aproximación a una gran cantidad de ecuaciones no-lineales sin metodos de linearización, perturbaciones o discretización que son computacionalmente muy costosos. Adomian lo explica en [1], quien mediante polinomios propuestos, generador a partir de operadores no lineales, es capaz de calcular un valor relacionado con la ecuación no-lineal mediante un método de descomposición de este valor.

La investigación continuó por parte de Wazwaz [20], quien propuso una nueva forma de calcular los polinomios propuestos por Adomian en [1], tomando en cuenta características que generan las variables sometidas a un operador no lineal.

Fatoorehchi y Aboolghasemi proponen en [10], una forma de calcular los polinomios tomando el método propuesto por Wazwaz en [20] y utilizando MATLAB con variables simbólicas para ello. Proponiendo el cálculo de los polinomios para operadores no-lineales, como potencias, exponenciales, logaritmos.

Después Fatoorehchi y Aboolghasemi en [13] proponen una aplicación puntual para estos polinomios, en la cual se usan para encontrar eigenvalores reales de matrices mediante la descomposición de Adomian.

También, en la literatura se encuentran algunas aplicaciones en el mundo real para el método de descomposición de Adomian [3, 5, 9, 11, 12, 18, 21, 22].

## 1.2. Planteamiento del problema

Implementar el método de descomposición de Adomian, que es capaz de evaluar ecuaciones no-lineales, buscando que no sea costoso computacionalmente, que sea lo mas apegado a la realidad posible y que pueda ser competitivo con otros métodos que ya se encuentren implementados. Así como encontrar la forma de ejecutar la mayor parte del algoritmo de manera paralela para que tenga un mejor desempeño.

## 1.3. Objetivo general

- Implementar el método de descomposición de Adomian propuesto para MATLAB, en C++. Aplicar técnicas de paralelismo para obtener una versión paralela del método de descomposición de Adomian, implementable en CUDA C/C++ y realizar las comparaciones correspondientes.

### 1.3.1. Objetivos específicos

1. Analizar y comprender las bases matemáticas del método de descomposición.

2. Implementar una transformación de los polinomios propuestos para su calculo en MATLAB y poder ser utilizados en C++

3. Comparar el método de descomposición de Adomian con métodos ya optimizados, tales como los usados en MATALAB.

#### Objetivo alcanzado no especificado

- Implementar en MATLAB de manera general, un generador de plantillas para $n$ GPUs, que permita mejorar el desempeño de cualquier algoritmo programado de forma paralela en CUDA C/C++

## 1.4. Perspectiva de los capítulos

El trabajo de tesis se encuentra desarrollado en 5 capítulos, de manera general se describe a continuación cada uno de ellos:

Capítulo 1: Se hace el planteamiento del problema, los objetivos del trabajo y una breve introducción del tema. Capítulo 2: Se encuentran las bases teóricas necesarias para la compresión del trabajo. Capítulo 3: Se presenta el desarrollo del trabajo, modificaciones propuestas para la implementación en C++ del método de descomposición de Adomian, análisis de complejidad del algoritmo y el generador de plantillas para $n$ número de GPUs. Capítulo 4: Se presentan los resultados y algunas simulaciones realizadas para observar el desempeño del algoritmos implementado. Capítulo 5: Se tratan las conclusiones y los trabajos futuros.

# Capítulo 2

# Bases teóricas

## 2.1. Método de descomposición

Una ventaja del método de descomposición es que puede proveer una aproximación a una gran cantidad de ecuaciones no-lineales sin métodos de linearización, perturbaciones o discretización que pueden ser computacionalmente muy costosos. El problema de estas ecuaciones, es que se tienen que suponer condiciones que simplifiquen el problema para poder ser resuelto y estas soluciones no siempre representan de manera certera al modelo físico que se quiere describir. No-linealidades 'débiles' o 'pequeñas' perturbaciones son suposiciones comúnmente tomadas. [1]

### 2.1.1. Descripción del método de descomposición

Sea la siguiente ecuación $Fu(t) = g(t)$, donde F representa un operador diferencial ordinario no-lineal, que involucra los términos no-lineales y lineales. El término lineal se descompone en $L + R$, donde $L$ puede obtenerse de manera sencilla su inverso y $R$ es el resto del operador lineal. De acuerdo a [1], $L$ se toma como la derivada de mayor orden. Así, se puede reescribir la ecuación de la siguiente forma (Las ecuaciones 2.1, 2.2, 2.3, 2.4 y 2.5 son tomadas de [1].)

$$Lu + Ru + Nu = g \tag{2.1}$$

donde $Nu$ representa los términos no-lineales. Resolviendo para $Lu$ tenemos

$$Lu = g - Ru - Nu \tag{2.2}$$

Ya que $L$ es invertible, una expresión equivalente es

$$L^{-1}Lu = L^{-1}g - L^{-1}Ru - L^{-1}Nu \tag{2.3}$$

Si corresponde a un problema de valor inicial, el operador integral $L^{-1}$ puede ser considerado como integrales definidas desde $t_0$ hasta $t$. Si $L$ es un operador de segundo orden, $L^{-1}$ es un operador integral doble y $L^{-1}Lu = u - u(t_0) - (t - t_0)u'(t_0)$. Para problemas con valores acotados (y, si se desea, para problemas con valores iniciales también), integraciones indefinidas son usadas y las constantes son evaluadas de las condiciones dadas. Resolviendo (2.3) para $u$ tenemos

$$u = A + Bt + L^{-1}g - L^{-1}Ru - L^{-1}Nu \tag{2.4}$$

El término no-lineal $Nu$ será equiparada con $\sum_{n=0}^{\infty} A_n$, donde $A_n$ son polinomios especiales que se explicarán más adelante y $u$ será descompuesta como $\sum_{n=0}^{\infty} u_n$, donde $u_0$ está dado por $A + Bt + L^{-1}g$, así tenemos

$$\sum_{n=0}^{\infty} u_n = u_0 - L^{-1}R\sum_{n=0}^{\infty} u_n - L^{-1}\sum_{n=0}^{\infty} A_n$$

entonces, podemos escribir

$$\begin{aligned} u_1 &= -L^{-1}Ru_0 - L^{-1}A_0 \\ u_2 &= -L^{-1}Ru_1 - L^{-1}A_1 \\ &\vdots \\ u_{n+1} &= -L^{-1}Ru_n - L^{-1}A_n \end{aligned} \tag{2.5}$$

Los polinomios $A_n$ son generados para cada no-linealidad de tal forma que $A_0$ dependa sólo de $u_0$, $A_1$ dependa sólo de $u_0$ y $u_1$, $A_2$ dependa de $u_0$, $u_1$ y $u_2$, etc. Todos los componentes $u_n$ son calculables y $u = \sum_{n=0}^{\infty} u_n$. Si la serie converge, el término $n$-ésimo de la suma parcial $\phi_n = \sum_{i=0}^{n-1} u_i$ será una solución aproximada, ya que $\lim_{n\to\infty} \phi_n = \sum_{i=0}^{\infty} u_i = u$ por definición.

Es importante enfatizar que los polinomios $A_n$ pueden ser calculados para no linealidades complicadas de la forma $f(u, u', \cdots)$ o $f(g(u))$. También que como no se esta linealizando o suponiendo un 'no-linealidad debil', las soluciones tienden a ser, físicamente, más precisas que aquellas aproximaciones basadas en suposiciones que simplifican el problema. Además, el número de valores deseados puede ser calculado y es fácil comprobar si existe convergencia mientras se van calculando los términos [1].

Ahora bien, si se considera ecuación general de funciones de la siguiente forma, como se ve en [17]

$$u - N(u) = f(u) \tag{2.6}$$

Donde $N$ es un operador no lineal de un espacio de Hilbert $H$ que mapea al mismo espacio $H$, y se busca que $u \in H$ satisfaga (2.6). Se asume que (2.6) tiene una solución única para cada $f \in H$.

Donde, recordando de el método de descomposición de Adomian [1], podemos descomponer $u$ como una sumatoria infinita de términos

$$u = \sum_{n=0}^{\infty} u_n \tag{2.7}$$

Y la función no-lineal $N(u)$ se descompone como

$$N(u) = \sum_{n=0}^{\infty} A_n \tag{2.8}$$

Donde $A_n$ son los polinomios de Adomian por calcular. El cálculo de estos polinomios sugerido en [19] es

$$A_n(u_0, u_1, ..., u_n) = \frac{1}{n!}\left[\frac{d^n}{d\lambda^n} N\left(\sum_{i=0}^{\infty} \lambda^i u_i\right)\right]_{\lambda=0} \tag{2.9}$$

Si se sustituye (2.7) y (2.8) en (2.6), tenemos

$$\sum_{n=0}^{\infty} u_n - \sum_{n=0}^{\infty} A_n = f(u) \tag{2.10}$$

Cada término de la serie $\sum_{n=0}^{\infty} u_n$ está dado por la siguiente relación recurrente

$$\begin{cases} u_0 = f(u) \\ u_n = A_{n-1}, \quad n \geq 1 \end{cases} \tag{2.11}$$

Sin embargo, en la práctica, no todos los términos de la serie $\sum_{n=0}^{\infty} u_n$ se pueden calcular y la solución será aproximada por por la serie truncada $\sum_{n=0}^{N} u_n$.
Cherruault [7] dio una prueba de la convergencia del método de descomposición de Adomian, introdujo una nueva formulación del método estableciendo que

$$S_n = u_1 + u_2 + \cdots + u_n \tag{2.12}$$

Y probó que $S_n$ converge a la solución de la ecuación de punto fijo

$$N(u_0 + S) = S \tag{2.13}$$

El término, $N(u_0 + S) = N(\sum_{i=0}^{n} u_i)$, es aproximado por la expresión

$$N_n(u_0 + S_n) = \sum_{i=0}^{n} A_i \tag{2.14}$$

El método de descomposición de Adomian es equivalente a determinar la secuencia $S_n$ que está definida por

$$\begin{cases} S_0 = 0 \\ S_{n+1} = N_n(u_0 + S_n), \quad n \geq 0 \end{cases} \tag{2.15}$$

Para probar la convergencia del método de descomposición de Adomian, las siguientes dos condiciones, de acuerdo a [7], deben de cumplirse

- $N$ es un contracción ($\|N\| < 1$)
- $\|N_n - N\| = \epsilon_n \to 0$ $(n \to \infty)$

Estás dos condiciones son fuertes. La segunda condición asegura la convergencia $\sum_{n=0}^{\infty} A_n$, lo cual es difícil de satisfacer para funciones altamente no lineales.

## 2.2. Prueba de convergencia de el método de descomposición de Adomian

Una vez que se han definido los polinomios de Adomian en (2.9), se puede reescribir (2.5) de la siguiente manera

$$\begin{aligned} u_0 &= f \\ u_1 &= A_0(u_0) \\ u_2 &= A_1(u_0, u_1) \\ &\vdots \\ u_n &= A_{n-1}(u_0, u_1, \cdots, u_{n-1}) \end{aligned} \tag{2.16}$$

Se supone lo siguiente de [6]

1. La solución $u$ de (2.6) puede ser encontrada como una serie de funciones $u_i$, de la forma, $u = \sum_{n=0}^{\infty} u_n$. Además, esta serie se supone absolutamente convergente, tal que, $\sum_{i=0}^{\infty} u_i < +\infty$.

2. La función no lineal $N(u)$ se puede desarrollar en un serie completa con radio de convergencia infinito. En otras palabras, podemos escribir

$$N(u) = \sum_{n=0}^{\infty} N_0^n \frac{u^n}{n!}, \qquad |u| < \infty \tag{2.17}$$

Se tiene el siguiente teorema de [6]

**Teorema 1** *Con las hipótesis anteriores 1 y 2 la serie de Adomian* $u = \sum_{i=0}^{\infty} u_i$ *es la solución de la ecuación* (2.6)*, cuando las* $u_i$ *satisfacen las relaciones de* (2.16)*.*

La prueba del teorema 1 se puede encontrar en [6].

## 2.3. Cálculo de los Polinomios de Adomian

En [20], Wazwaz encontró una característica interesante de los polinomios de Adomian: La suma de los subíndices de los componentes de $u$ en cada término del polinomio $A_n$ es igual a $n$. Con lo cual, se encontró un esquema alternativo para obtener los polinomios de Adomian con un volumen reducido de cálculos.
Siguiendo lo propuesto por Wazwaz, se elaboró un código para computadora en MATLAB, que usa programación simbólica y funciones de cadenas eficientemente para calcular cualquier componente de Adomian deseado pertenecientes a diferentes no-linealidades. El código *AdomPoly* que se encuentra descrito en el capítulo de simulaciones, es el propuesto en [10]
Para el uso del código, se define una variable simbólica en MATLAB de la forma

$$NON = u_0 + u_1 + \cdots + u_n \tag{2.18}$$

Donde $u_i$ es una variable simbólica y $n$ es lo suficientemente grande. La función recibe como parámetro a NON de (2.18), modificada por algún operador no-lineal. El resultado del uso de la función para una no-linealidad $N(u) = u^2$ es:

| |
|---|
| $A_0 = u_0^2$ |
| $A_1 = 2u_0u_1$ |
| $A_2 = 2u_0u_2 + u_1^2$ |
| $A_3 = 2u_0u_3 + 2u_1u_2$ |
| $A_4 = u_2^2 + 2u_1u_3 + 2u_0u_4$ |
| $A_5 = 2u_2u_3 + 2u_0u_5 + 2u_1u_4$ |
| $\vdots$ |

**Tabla** 2.1: Polinomios calculados para una no-linealidad $N(u) = u^2$

El resultado del uso de la función para una no-linealidad $N(u) = u^3$ es:

| |
|---|
| $A_0 = u_0^3$ |
| $A_1 = 3u_0^2u_1$ |
| $A_2 = 3u_1^2u_0 + 3u_0^2u_2$ |
| $A_3 = u_1^3 + 3u_0^2u_3 + 6u_0u_1u_2$ |
| $A_4 = 3u_1^2u_2 + 3u_2^2u_0 + 3u_0^2u_4 + 6u_1u_3u_0$ |
| $A_5 = 6u_2u_3u_0 + 3u_2^2u_1 + 3u_1^2u_3 + 3u_0^2u_5 + 6u_1u_4u_0$ |
| $\vdots$ |

**Tabla** 2.2: Polinomios calculados para una no-linealidad $N(u) = u^3$

## 2.4. Eigenvalores reales de matrices mediante la descomposición de Adomian

Se dice que $\lambda$ es un eigenvalor de una matriz cuadrada de $\mathbf{A}$ de $n \times n$, si satisface la ecuación matricial siguiente

$$\mathbf{Ax} = \lambda\mathbf{x} \tag{2.19}$$

donde $\mathbf{x}$ es un vector columna no cero de dimensión $n$.
La ecuación (2.19) puede ser escrita de la forma

$$\mathbf{A} - \lambda\mathbf{I} = \mathbf{0} \tag{2.20}$$

donde $\mathbf{I}$ es una matriz de identidad de orden $n$.
Para que la ecuación (2.20) sea cierta, la matriz $\mathbf{A} - \lambda\mathbf{I}$ tiene que ser singular, esto es

$$\det(\mathbf{A} - \lambda\mathbf{I}) = 0 \tag{2.21}$$

El determinante del lado izquierdo de (2.21), se puede expandir, de [13] como

$$a_0\lambda^n + a_1\lambda^{n-1} + \cdots + a_{n-1}\lambda + a_n = 0 \tag{2.22}$$

Regularmente, en álgebra lineal, (2.22) es denominada a la ecuación característica o polinomio característico de la matriz $\mathbf{A}$.
Entre muchos métodos para la determinación de los valores y vectores propios,

el de Faddeev-Leverrier se destaca en la obtención del polinomio característico de matrices [15] y [16]. Suponiendo que $\mathbf{A}$ es $n \times n$, el método consiste en

$$\begin{cases} \mathbf{A}_1 = \mathbf{A} \\ \mathbf{A}_{i+1} = \mathbf{A}(\mathbf{A}_i + a_i\mathbf{I}) \end{cases} \tag{2.23}$$

donde

$$\begin{cases} a_0 = 1 \\ a_i = -\dfrac{trace(\mathbf{A}_i)}{i}; \qquad n \leq i \leq n \end{cases} \tag{2.24}$$

De acuerdo a ésto, todos los polinomios de la ecuación característica (2.22) se encuentran, recursivamente, mediante (2.23) y (2.24).

Se puede pensar que la ecuación (2.22) no tiene una complejidad alta, sin embargo, lo contrario lo muestra Abel y Galois en su trabajo [14] donde sugiere que no hay una formula general para encontrar las raíces de polinomios con grado mayor a cuatro. Un algoritmo numérico robusto y eficiente para determinar todas las raíces de un polinomio de orden alto es difícil de encontrar [2].

En este contexto encontrar los valores de $\lambda$ de (2.22) utilizando el método de descomposición de Adomian tiene un gran valor.

### 2.4.1. El método propuesto

Para utilizar el método de descomposición de Adomian para encontrar los eigenvalores, es indispensable escribir la ecuación (2.22) en forma de punto fijo, concretamente $\lambda = g(\lambda)$. Dando por hecho que $a_{n-1} \neq 0$, podemos dividir la ecuación (2.22) entre $a_{n-1}$, así tenemos la siguiente ecuación [13]

$$\frac{a_0}{a_{n-1}}\lambda^n + \frac{a_1}{a_{n-1}}\lambda^{-1} + \cdots + \frac{a_{n-1}}{a_{n-1}}\lambda + \frac{a_n}{a_{n-1}} = 0$$

Despejando $\lambda$ tenemos

$$\lambda = -\frac{a_0}{a_{n-1}}\lambda^n - \frac{a_1}{a_{n-1}}\lambda^{n-1} - \cdots - \frac{a_{n-2}}{a_{n-2}}\lambda^2 - \frac{a_n}{a_{n-1}} \tag{2.25}$$

Y de acuerdo con el método de descomposición de Adomian

$$\lambda = \sum_{i=0}^{\infty} \xi_i$$

Donde $\xi_0 = -\frac{a_n}{a_{n-1}}$ y las no-linealidades $\lambda^n, \lambda^{n-1}, \cdots, \lambda^2$, serán reemplazadas por sus polinomios de Adomian correspondientes, denotados por, $A_n, B_n, \cdots, Z_n$, entonces tenemos que la ecuación para encontrar el siguiente valor de la descomposición de Adomian es

$$\xi_{i+1} = -\frac{a_0}{a_{n-1}}A_i - \frac{a_1}{a_{n-1}}B_i - \cdots - \frac{a_{n-2}}{a_{n-2}}Z_i \; ; \; i \geq 0 \tag{2.26}$$

Para enfatizar $\lambda_1 = \sum_{i=0}^{\infty} \xi_i$ es un eigenvalor de la matriz $\mathbf{A}$. Es importante mencionar que, en algunas ocasiones, la secuencia producida por (2.26) diverge. Para corregir este defecto, se puede agregar una matriz diagonal auxiliar $\alpha\mathbf{I}$ a $\mathbf{A}$ para crear una nueva matriz $\Theta = \mathbf{A} - \alpha\mathbf{I}$. Una vez que $\alpha$ se escoge adecuadamente, el método de descomposición de Adomian encuentra un eigenvalor de la matriz $\Theta$. Con ayuda del Lemma 1, el valor correspondiente al eigenvalor de la matriz $\mathbf{A}$ se puede encontrar fácilmente.

**Lemma 1** *Sean* $\mathbf{A}$ *y* $\mathbf{B}$ *matrices de* $n \times n$, $\mathbf{I}$ *representa la matriz identidad de* $n$ *dimensiones,* $\alpha$ *es un número real, y* $eig()$ *representa un operador que regresa un eigenvalor de la matriz en su argumento. Si* $\mathbf{A} = \mathbf{B} + \alpha\mathbf{I}$ *y* $eig(\mathbf{A}) = \lambda$, *entonces se tiene que* $eig(\mathbf{B}) = \lambda - \alpha$.

La prueba se encuentra en [13].
Una vez que el eigenvalor, $\lambda_1$, es determinado, podemos encontrar los otros eigenvalores de la siguiente forma:

1. Se divide la ecuación (2.22) entre el eigenvalor encontrado, así tenemos

$$\frac{a_0\lambda^n + a_1\lambda^{n-1} + \cdots + a_{n-1}\lambda + a_n}{\lambda - \lambda_1} \tag{2.27}$$

2. En consecuencia, se crea una nueva ecuación de punto fijo (nuevas no-linealidades surgen, del tipo racional para ser específicos)

$$\frac{b_0 + b_1(\lambda - \lambda_1) + b_2(\lambda - \lambda_1)^2 + \cdots + b_{n-1}(\lambda - \lambda_1)^{n-1} + (\lambda - \lambda_1)^n}{\lambda - \lambda_1} = 0 \tag{2.28}$$

$$\frac{b_0}{\lambda - \lambda_1} + b_1 + b_2(\lambda - \lambda_1) + \cdots + b_{n-1}(\lambda - \lambda_1)^{n-2} + (\lambda - \lambda_1)^{n-1} = 0 \tag{2.29}$$

$$\frac{1}{b_2}\frac{b_0}{\lambda-\lambda_1}+\lambda+\frac{b_1}{b_2}-\lambda_1+\cdots+\frac{b_{n-1}}{b_2}(\lambda-\lambda_1)^{n-2}+\frac{1}{b_2}(\lambda-\lambda_1)^{n-1}=0 \quad (2.30)$$

$$\lambda=\lambda_1-\frac{b_1}{b_2}-\frac{1}{b_2}\frac{b_0}{\lambda-\lambda_1}+\cdots-\frac{b_{n-1}}{b_2}(\lambda-\lambda_1)^{n-2}-\frac{1}{b_2}(\lambda-\lambda_1)^{n-1} \quad (2.31)$$

3. Regresando al método de descomposición de Adomian para encontrar la solución, un nuevo eigenvalor, para esta nueva ecuación (2.31). Repitiendo el método mencionado, se puede encontrar todos los eigenvalores reales de la matriz.

Ejemplos numéricos de este método se pueden encontrar en [13].

## 2.5. Teoría de paralelismo

Hablar de teoría de paralelismo es un tema muy amplio que abarca muchos antecedentes matemáticos y una gran cantidad de conceptos, por esta razón, para esta sección se hablará solo de conceptos principales que sean útiles para el trabajo desarrollado y solo se plantearán de manera superficial las demostraciones matemáticas, dejando al lector el trabajo de profundizar un poco más si lo cree conveniente.
Las conceptos serán tomados de [4], a menos que se indique otra fuente en alguno de ellos.

### 2.5.1. Relaciones

El producto cartesiano $S \times T$, de dos conjuntos $S$ y $T$ es el conjunto de todos los pares ordenados de la forma $(a, b)$ donde $a \in S$ y $b \in T$ Una *relación* $R$ de un conjunto $S$ a un conjunto $T$ es cualquier subconjunto de $S \times T$. Una relación en un conjunto $S$ es un subconjunto de $S \times S$. Si $R$ es una relación en $S$ y $(a, b) \in R$, se escribe como $aRb$. La inversa de una relación $R$, se escribe como $R^{-1}$
Una relación $R$ en un conjunto $S$ es:

- reflexiva Si $aRa$ es cierto para toda $a$,
- irreflexiva Si $aRa$ es falso para toda $a$,
- simétrica Si $aRb$ implica $bRa$,
- antisimétrica Si $aRb$ y $bRa$ implica que $a = b$,
- transitiva Si $aRb$ y $bRc$ implica que $aRc$,

donde $a$, $b$ y $c$ denotan elementos arbitrarios del conjunto $S$.
Sea $U$ el conjunto universo y sean $A_1, A_2, A_3, \cdots, A_n$ $n$ conjuntos que son disjuntos, es decir $A_1 \cap A_2 \cap A_3 \cap \cdots \cap A_n \in \emptyset$.
Un *partición* de un conjunto $S$ no vacío ($S \subset U$), es una colección de pares disjuntos de subconjuntos no vacíos, cuya unión es $S$. Matemáticamente sería $A_1 \cup A_2 \cup A_3 \cup \cdots \cup A_n \in S$.

Una relación $R$ se denomina *relación de equivalencia* si es reflexiva, simétrica y transitiva.
Una relación es de *orden parcial* si es reflexiva, antisimétrica y transitiva, $\leq$.

Un ejemplo de un conjunto parcialmente ordenado es un conjunto con orden parcial en el. Supóngase $(S, \leq)$ es un conjunto parcialmente ordenado y $a, b$ son elementos de $S$. Un caso particular de $\leq$ es cuando, de dos elementos, uno es estrictamente menor al otro, esto se denomina por $<$. Si $a < b$, entonces se dice que $a$ es un *predecesor* de $b$ y que $b$ es un sucesor de $a$. Dos elementos de un conjunto se pueden comparar cuando es posible saber si uno es menor, mayor o igual al otro. Un subconjunto $T$ de $S$ es una *cadena* si dos elementos en $T$ se pueden comparar siempre, se dice que es una *anticadena* si dos elementos distintos de $T$ nunca son comparables.

Un *orden total* en un conjunto $S$ es un orden parcial, de tal forma que cualesquiera dos elementos de $S$ son siempre comparables, es decir, todo el conjunto $S$ es una cadena. Por tanto un conjunto *totalmente ordenado* es un conjunto con orden total en el.
***Ejemplo 2.1*** Sea $S$ un conjunto de todos los pares ordenados de la forma $(I_1, I_2)$ donde $I_1$ y $I_2$ son enteros en $\{0, 1, 2, 3\}$. Para $(i_1, i_2)$ y $(j_1, j_2)$ en $S$, se define

$$(i_1, i_2) \leq (j_1, j_2) \qquad iff \qquad i_1 \leq j_1 \quad and \quad i_2 \leq j_2$$

Para cualesquiera elementos $(i_1, i_2), (j_1, j_2)$ y $(k_1, k_2)$ de $S$ tenemos:

1. $(i_1, i_2) \leq (j_1, j_2)$;

2. $(i_1, i_2) \leq (j_1, j_2)$ y $(j_1, j_2) \leq (i_1, i_2)$ implican que $(i_1, i_2) = (j_1, j_2)$;

3. $(i_1, i_2) \leq (j_1, j_2)$ y $(j_1, j_2) \leq (k_1, k_2)$ implican que $(i_1, i_2) \leq (k_1, k_2)$

El ejemplo fue tomado de [4], donde se puede encontrar también la demostración del mismo.
El gráfico que representaría este conjunto sería

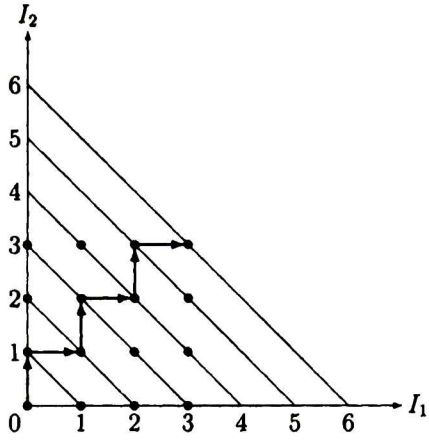



**Figura** 2.1: El conjunto $S$ del ejemplo **2.1**

Se puede notar que las relaciones entre puntos de un mismo conjunto para este ejemplo, siguen un patrón de tal forma que pareciera que el punto $(3,3)$ tiene relación con el punto $(0,0)$ en el plano de manera indirecta.

### 2.5.2. Orden lexicográfico

El producto cartesiano $S_1 \times S_2 \times \cdots \times S_m$ de $m$ subconjuntos $S_1, S_2, \cdots, S_m$ es el conjunto de todas las $m$-tuplas ordenadas de la forma $(a_1, a_2, \cdots, a_m)$ donde $a_1 \in S_1, a_2 \in S_2, \cdots, a_m \in S_m$. Se considera este productor cartesiano en $\mathbf{Z}^m$, donde $\mathbf{Z}$ es el conjunto de todos los enteros.
Sea $\mathbf{i} \in Z^m$ un vector entero de dimensión $m$, donde $\mathbf{i} = (i_1, i_2, \cdots, i_m)$, su elemento líder es el primer elemento diferente cero. Si su elemento líder es $i_l$, entonces el entero positivo $l$ entre 1 y $m$, se conoce como *nivel* de $\mathbf{i}$. Un vector es, lexicograficamente, positivo o negativo si su elemento líder es positivo o negativo, respectivamente.
El signo de un entero se denota como $sig(i)$, donde $sig(i) = 1$ si $i$ es positivo, $sig(i) = -1$ si $i$ es negativo y $sig(0) = 0$. El signo de un vector $\mathbf{i} = (i_1, i_2, \cdots, i_m)$ es el vector de signos de sus componentes

$$\mathbf{sig(i)} = (sig(i_1), sig(i_2), \cdots, sig(i_m))$$

Un vector $\mathbf{i}$, es un *vector de dirección* si cada uno de sus componentes es uno de los enteros: $1, 0, -1$.
Para $1 \leq l \leq m$, se define una relación $\prec_l$ en $\mathbf{Z}^m$, de tal forma que $\mathbf{i} \prec_l \mathbf{j}$ si

$$i_1 = j_1, i_2 = j_2, \cdots, i_{l-1} = j_{l-1}, \quad y \quad i_l < j_l$$

El orden lexicográfico $\prec$ en $\mathbf{Z}^m$ está definido como la unión de todas las relaciones $\prec_l$, tal que,

$$\mathbf{i} \prec \mathbf{j} \qquad iff \qquad \mathbf{i} \prec_l \mathbf{j} \quad para\ algun\ l\ en\ 1 \leq l \leq m$$

### 2.5.3. Matrices unimodulares

Una matriz es entera si todos sus elementos son enteros. Debido a que los indices en los bucles de nuestros programas son valores enteros, en el análisis de los mismos, se requieren soluciones enteras a ecuaciones, al igual que matrices enteras para transformar un conjunto de vectores enteros en otro conjunto de vectores enteros diferente. Las ecuaciones de variables enteras se llaman *ecuaciones diofantinas.*

Una matriz unimodular, está definida de tal forma que su inversa es necesariamente entera. Se dice que la matriz cuadrada entera $\mathbf{A}$ es unimodular si $\det(\mathbf{A}) = \pm 1$. Esta definición garantiza que la inversa $\mathbf{A}^{-1}$ exista, que sea una matriz entera y que es unimodular por si misma. Las matrices unimodulares se usan para la resolución de ecuación de ecuaciones diofantinas. De las cuales se hablará un poco mas a detalle en la sección 2.5.6.
Sea $\mathbf{R}^m$ un producto cartesiano de $m$ copias de la recta real, $m \geq 1$. Los elementos de $\mathbf{R}^m$ son los vectores reales de tamaño $m$. Un semi-espacio en $\mathbf{R}^m$ es un conjunto de la forma

$$\{(x_1, x_2, \cdots, x_m) \in \mathbf{R}^m : a_1 x_1, a_2 x_2, \cdots, a_m x_m \leq c\}$$

Para algún vector diferente de cero $(a_1, a_2, \cdots, a_m)$ y para algún número real $c$.
Un *politopo* en $\mathbf{R}^m$ es un subconjunto acotado, dado por la intersección de un número finito de semi-espacios.

### 2.5.4. Reducción escalonada

Reducir una matriz **A** de $m \times n$ a su forma escalonada, significa encontrar una matriz unimodular **U** y un matriz escalonada **S** de $m \times n$, tal que $\mathbf{UA} = \mathbf{S}$. Tal reducción siempre es posible. Las matrices **U** y **S** no son únicas.
En [4] se encuentran algoritmos para calcular estas matrices. Los cuales se utilizarán posteriormente en este trabajo

### 2.5.5. Dependencia de datos

Para analizar los conceptos de dependencia, primero se debe partir del hecho de que tenemos uno o varios bucles anidados, de la forma Una estructura de ciclos anidados perfecto, tiene la siguiente estructura:

$$
\begin{array}{ll}
L_1: & \textbf{do } I_1 = p_1, q_1 \\
L_2: & \quad \textbf{do } I_2 = p_2, q_2 \\
\vdots & \quad \vdots \\
L_m: & \quad\quad \textbf{do } I_m = p_m, q_m \\
 & \quad\quad\quad H(I_1, I_2, \cdots, I_m) \\
 & \quad\quad \textbf{enddo} \\
 & \quad\quad \vdots \\
 & \quad \textbf{enddo} \\
 & \textbf{enddo}
\end{array}
$$

Donde $p_1, q_1$ son constantes enteras, $p_r, q_r$ son funciones enteras evaluadas de $I_1, I_2, \cdots, I_{r-1}$ para $1 < r \leq m$. Una estructura de ciclos anidados perfecto cuando no hay ninguna instrucción entre lazos. Se representan los lazos como $\mathbf{L} = (L_1, L_2, \cdots, L_m)$. Un espacio de iteración se asocia a los lazos **L**, el cual contiene un punto en el espacio $m$, dicho *punto de iteración* se representa por $\mathbf{i} = (i_1, i_2, \cdots, i_m)$ donde los valores $i_1, i_2, \cdots, i_m$ representan los valores tomados por $x_1, x_2, \cdots, x_m$ respectivamente. Los vectores de iteración se puede escribir de manera general como $\mathbf{I} = (I_1, I_2, \cdots, I_m)$ que pueden tomar valores de los puntos de iteración.
El cuerpo del nido de lazos **L** es $H(I_1, I_2, \cdots, I_m)$ o $\mathbf{H(I)}$, los cuerpos de los bu-

cles contienen sentencias de asignación que se ejecutan de manera secuencial en el programa. Un vector de iteración dado $\mathbf{i} = (i_1, i_2, \cdots, i_m)$, define una instancia particular del cuerpo $\mathbf{H(i)}$. Una iteración $\mathbf{H(i)}$, se ejecuta antes de una iteración $\mathbf{H(j)}$, si y solo si $\mathbf{i} \prec \mathbf{j}$.

**Lemma:** Sean $S$ y $T$ dos instrucciones de asignación en el cuerpo de los lazos anidados $\mathbf{L}$. En la ejecución de $\mathbf{L}$, una instancia $S(\mathbf{i})$ de $S$ se ejecuta antes de una instancia $T(\mathbf{j})$ de $T$ si y solo si, alguna de las siguientes condiciones se cumplen:

$$\mathbf{i} \prec \mathbf{j}$$
$$\mathbf{i} = \mathbf{j} \;\&\; S < T$$

La relación de dependencia entre instrucciones de asignación es denotada por $\delta$, y se define como sigue: Para dos instrucciones $S$ y $T$, tenemos que $S\delta T$ si hay una instancia $S(\mathbf{i})$ de $S$, una instancia $T(\mathbf{j})$ de $T$, y una localidad de memoria $\mathbf{M}$, tal que:

Ambos $S(\mathbf{i})$ y $T(\mathbf{j})$ referencian (escriben o leen) $\mathbf{M}$.

$S(\mathbf{i})$ se ejecuta antes de $T(\mathbf{j})$ en $\mathbf{L}$.

- Durante la ejecución de $\mathbf{L}$, la localidad $\mathbf{M}$ no es escrita durante el periodo de tiempo que está entre la ejecución de $S(\mathbf{i})$ y el comienzo de la ejecución de $T(\mathbf{j})$.

Como de una memoria se 'lee' o se 'escribe', dos instrucciones puede referenciar a la memoria de 4 formas diferentes:

1. Si $S(\mathbf{i})$ escribe en la localidad de memoria $\mathbf{M}$ y $T(\mathbf{j})$ la lee, se tiene que $T$ tiene una *dependencia de flujo* con $S$

2. Si $S(\mathbf{i})$ lee en la localidad de memoria $\mathbf{M}$ y después $T(\mathbf{j})$ la escribe, entonces tenemos que $T$ es *antidependiente* con $S$

3. La instrucción $T$ tiene *dependencia de salida* con $S$, si $S(\mathbf{i})$ y $T(\mathbf{j})$ ambas escriben la localidad de memoria $\mathbf{M}$

4. La instrucción $T$ tiene *dependencia de entrada* con $S$, si $S(\mathbf{i})$ y $T(\mathbf{j})$ ambas leen la localidad de memoria $\mathbf{M}$

Las cuatro tipos de dependencias presentadas, de acuerdo a lo anterior, son: dependencia de flujo, antidependencia, dependencia de salida, dependencia de entrada y se representan por $\delta^f, \delta^a, \delta^o, \delta^i$ respectivamente.
Estas relaciones de dependencia se puede traslapar y constituyen la relación principal de dependencia:

$$\delta = \delta^f \cup \delta^a \cup \delta^o \cup \delta^i$$

Si una iteración $H(\mathbf{j})$ depende de una iteración $H(\mathbf{i})$, entonces el *vector diferencia* $\mathbf{d} = \mathbf{j} - \mathbf{i}$ entre dos puntos de iteración se le llama *vector distancia* para un nido de bucles $\mathbf{L}$. Ya que por definición tenemos que $\mathbf{i} \prec \mathbf{j}$, entonces $\mathbf{d} \succ \mathbf{0}$, esto es, un vector distancia siempre es lexicográficamente positivo.

### 2.5.6. Ecuaciones Diofantinas

Surgen ecuaciones durante el análisis de paralelismo debido a que los indices de nuestros bucles anidados cumplen con ser variables enteras, este tipo de ecuaciones polinomiales se conocen como *ecuaciones diofantinas.*
Un ejemplo en donde se tendrá este tipo de ecuaciones sería considerado el siguiente bucle

```
1: for I_1 = 1, 100 do
2:     for I_2 = 1, 200 do
3:         S: X(2I_1 + 2I_2 − 3) = Y(I_1, I_2 + 1) − 5
4:         T: Z(I_1 + 1, I_2) = X(6I_1 + 4I_2 − 2) − 2
5:     end for
6: end for
```

Cuando se quiere encontrar si alguna sentencia en el programa depende de otra,

se analizan las variables de salida y de entrada de cada una de ellas. Tomando en cuenta esto, si se quiere saber si la sentencia $T$ depende de la sentencia $S$ en el bucle anterior, tenemos que encontrar que la variable de salida de la sentencia $S$, $X(2I_1+2I_2-3)$, tiene alguna dependencia con la variable de entrada de la sentencia $T$, $X(6I_1+4I_2-2)$. Ambas representan una localidad de memoria de donde se va a almacenar o leer algún valor, representarán la misma localidad de memoria, si y sólo si

$$2i_1 + 2i_2 - 3 = 6i_1 + 4i_2 - 2$$

Despejando

$$2i_1 - 6i_1 + 2i_2 - 4i_2 = 1 \tag{2.32}$$

Si se considera esta ecuación sin restricciones extras, sólo las dadas en el bucle que contiene las dos sentencias, es fácil ver que se tiene un número infinito de soluciones reales, por ejemplo, si se le da valores enteros arbitrarios a $j_1, j_2, i_2$ y tratamos de resolver para $i_1$, tenemos

$$i_1 = (6i_1 - 2i_2 + 4i_2 + 1)/2$$

Para decidir si se tienen soluciones enteras, se puede observar que el máximo común divisor de los coeficientes en el lado izquierdo de (2.32) es 2. Por tanto, si existiera una solución entera, 2 tendría que dividir de manera exacta también al lado derecho de la ecuación, lo cual no se cumple. Así, no existen soluciones enteras para esta ecuación diofantina y la sentencia $T$ no depende de $S$ en el programa dado.
Esto se puede generalizar, considerando la siguiente ecuación diofantina

$$a_1x_1 + a_2x_2 + \cdots + a_mx_m = c$$

que tiene $m$ variables, los coeficientes $a_1, a_2, \cdots, a_m$ no todos son cero y del lado derecho de la ecuación se tiene un entero $c$. Sea $g$ el máximo común divisor de $a_1, a_2, \cdots, a_m$.
Se puede escribir la ecuación anterior de forma matricial, como

$$\mathbf{xA} = c \tag{2.33}$$

donde $\mathbf{x} = (x_1, x_2, \cdots, x_m)$ y $\mathbf{A}$ denota la matriz de coeficientes $(a_1, a_2, \cdots, a_m)'$. Mediante el algoritmo de reducción escalonada mencionada anteriormente, podemos

encontrar una matriz escalonada **S** y una matriz unimodular **U** de forma que $\mathbf{UA} = \mathbf{S}$.
Así tenemos el siguiente teorema

**Teorema 2** *La ecuación diofantina lineal* (2.33) *tiene una solución si y sólo si, el máximo común divisor de sus coeficientes divide a c. Cuando la solución existe, el conjunto de todas las soluciones está dado por la formula (que representa la solución general)*

$$\mathbf{x} = (c/g, t_2, t_3, \cdots, t_m) \cdot \mathbf{U} \tag{2.34}$$

*Donde* $t_2, t_3, \cdots, t_m$ *son valores enteros arbitrarios, y* **U** *es cualquier matriz unimodular de* $m \times m$ *tal que* $\mathbf{UA} = (g, 0, \cdots, 0)'$

La prueba al Teorema, se encuentra en [4].

### 2.5.7. Ecuación de dependencia

Tomando en cuenta el modelo de bucles anidados de la subsección de dependencia de este capítulo, se supone que todas las variables en el programa son arreglos de elementos con subíndices de manera lineal en $I_1, I_2, \cdots, I_m$. Una variable que es un elemento de un arreglo $n$-dimensional $X$, tiene la forma

$$X(a_{11}I_1 + \cdots + a_{m1}I_m + a_{10}, \cdots, a_{1n}I_1 + \cdots + a_{mn}I_m + a_{n0})$$

Donde los coeficientes son todos variables enteras. En forma matricial, esta variable puede ser escrita como $X(\mathbf{IA} + \mathbf{a}_0)$ donde

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

$\mathbf{I} = (I_1, I_2, \cdots, I_m)$ y $\mathbf{a}_0 = (a_{10}, a_{20}, \cdots, a_{n0})$. La matriz de coeficientes de este elemento de $X$ es **A** y su termino constante es el vector $\mathbf{a}_0$.
Sean $S$ y $T$ dos sentencias, no necesariamente distintas, en $H$. Sea $X(\mathbf{IA} + \mathbf{a}_0)$

denoten una variable en $S$ y $X(\mathbf{IB} + \mathbf{b_0})$ una variable en $T$, donde $X$ es un arreglo $n$-dimensional, $\mathbf{A}$ y $\mathbf{B}$ son matrices enteras de $m \times n$, y $\mathbf{a_0}$ y $\mathbf{b_0}$ son $n$-vectores enteros. La instancia de $X(\mathbf{IA} + \mathbf{a_0})$ para un punto de iteración $\mathbf{i}$ es $X(\mathbf{iA} + \mathbf{a_0})$ y la instancia de $X(\mathbf{IB} + \mathbf{b_0})$ para un punto de iteración $\mathbf{j}$ es $X(\mathbf{jB} + \mathbf{b_0})$.

### 2.5.8. Transformación de un nido de bucles anidados mediante matrices de transformaciones unimodulares

Esta sección tiene como objetivo plantear el procedimiento para transformar un nido de bucles anidados (perfecto) mediante matrices de transformación unimodulares. El procedimiento consta de 8 pasos, los cuales son:

1. Encontrar todos los vectores distancia $\mathbf{d}$.
2. Seleccionar la matriz de transformación $\mathbf{U}$.
3. Verificar la legalidad de la transformación, $\mathbf{dU} \succ \mathbf{0}$ para cada vector distancia $\mathbf{d}$.
4. Calcular el nuevo vector de iteración $\mathbf{y} = (y_1, y_2, \cdots, y_m)$, $\mathbf{x} = \mathbf{y}(\mathbf{U}^{-1})^t$
5. Expresar los límites del espacio de iteración original en forma de politopo, $\mathbf{xA} \leq \mathbf{b}$.
6. Calcular el nuevo espacio de iteración, $\mathbf{y}(\mathbf{U}^{-1})^t\mathbf{A} \leq \mathbf{b}$.
7. Resolver el sistema de desigualdades aplicando el método de eliminación de Fourier-Motzkin, para encontrar los nuevos límites inferiores y superiores de $\mathbf{y}$.
8. Reemplazar $\mathbf{x}$ por $\mathbf{y}$ en el cuerpo del nido de bucles (ahora $H(\mathbf{y})$) y en los nuevos límites de los bucles *for*.

Se llamará *politopo destino* al espacio de iteración del nido de bucles resultante después de realizar una transformación.

### 2.5.9. Tipos de paralelismo

Existen dos tipos de paralelismo, el *paralelismo interno* que se refiere a la posibilidad de ejecutar de manera paralela los bucles internos y el *paralelismo externo* que es para los bucles externos.

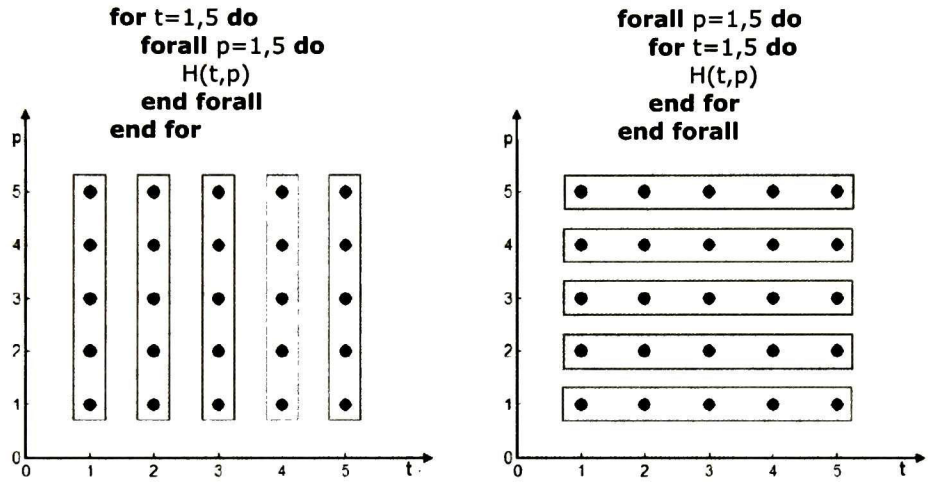



**Figura** 2.2: Paralelismo interno y externo

En la figura 2.2 se muestra un ejemplo en donde se tienen dos bucles y uno de ellos se puede ejecutar en paralelo, indicado por un *forall*, el índice $t$ representa un determinado tiempo y el índice $p$ representa un determinado número de elementos de procesamiento.
Como podemos observar en la parte izquierda, tenemos que el bucle externo es un *for* por lo que las iteraciones $t$ son secuenciales, mientras que, el bucle interno es un *forall* por lo que las iteraciones en $p$ se pueden ejecutar de manera paralela. En cada $t$ se pueden ejecutar los 5 puntos de iteración que se encuentran encerrados en el recuadro azul (bajo el supuesto que no existen dependencias en $p$). Por tanto, si se tuvieran 5 elementos de procesamiento, se podrían ejecutar los 5 puntos de iteración en la dirección de $p$ en cada tiempo. Los 5 elementos de procesamiento deben esperar

a que todos terminen para que la siguiente iteraciónn de $t$ pueda comenzar, de allí la razón por la que a este tipo de paralelismo también se le conoce como paralelismo síncrono, o también, paralelismo vertical.
Ahora describiremos el paralelismo externo, de igual manera si se tienen 5 elementos de procesamiento, cada uno de ellos puede ejecutar el bucle interno (de manera secuencial) para las 5 iteraciones en $t$, ya que no se necesita sincronía entre los elementos de procesamiento, este tipo de paralelismo también se le conoce como paralelismo asíncrono, o también, paralelismo horizontal.

## 2.6. Resumen del capítulo

Es importante lo mencionado en este capítulo, ya que se abordaron algunos temas teóricos necesarios para la compresión del trabajo realizado. La comprensión de las ecuaciones del método propuesto de Adomian ayudan a comprender en el siguiente capítulo algunas de las decisiones tomadas durante la implementación, así como el análisis de complejidad que se presenta más adelante. La teoría de paralelismo ayuda al lector a familiarizarse con las decisiones tomadas al momento de implementar el algoritmo en un versión paralela.

# Capítulo 3

# Desarrollo

## 3.1. Retomando el problema

Como se menciona anteriormente, el modelado de la realidad es algo complejo que casi siempre termina en resolver ecuaciones no lineales para encontrar el comportamiento de lo se está modelando. Uno de los problemas es la forma de resolverlo, ya que hay muchos métodos donde se tiene que suponer linealidad en el problema para encontrar una aproximación, el método de descomposición de Adomian, permite encontrar la solución a problemas no lineales sin suponer linealidad, lo cual da una aproximación más certera con el modelo físico. Se trata en este capítulo, un problema no lineal, tal es encontrar los eigenvalores de una matriz o encontrar las raíces de algún polinomio, pensando en ampliar posteriormente el tipo de ecuaciones no lineales capaces de ser resueltas por el método.

## 3.2. Manejo de los polinomios

### 3.2.1. MATLAB

En el manejo de los polinomios en MATLAB, se utiliza el tipo de variable 'celda', para el almacenamiento se toma una celda de $n \times m$, donde $n$ es el tamaño de la matriz (recordando que es cuadrada), y $m$ es el número de polinomios que se quieran evaluar. Cabe hacer mención que entre más polinomios se evalúen, más preciso serán

los eigenvalores calculados y por ende más preciso el algoritmo, pero hay que tomar en cuenta que esto también aumenta el tiempo de ejecución.
De acuerdo al algoritmo propuesto en [10], las variables que obtenemos son de tipo simbólico. Las celdas, el tipo de variable que se escogió para almacenar los polinomios, tiene la propiedad de poder almacenar una variable simbólica en cada una de sus posiciones.

Debido al tipo de variables utilizadas (simbólicas), el uso del programa se ve limitado a poder ser usado sólo con MATLAB, entonces se busca una solución para poder ejecutar en el lenguaje C++ el algoritmo.
Tomando en cuenta que los polinomios generados para no-linealidades, donde se utilizan potencias, sólo tienen $u_i$'s y factores de ponderación para cada término del polinomio, se toma un ejemplo concreto del polinomio $A_3$ cuando es calculado para una no linealidad $Nu = u^3$, se puede reescribir de la siguiente manera el polinomio

| | | | | |
|---|---|---|---|---|
| | | $3u_0^2u_3^1$ | | $[3,0,2,3,1]$ |
| $A_3 = 3u_0^2u_3^1 + 1u_1^3 + 6u_0^1u_1^1u_2^1$ | $\rightarrow$ | $1u_1^3$ | $\rightarrow$ | $[1,1,3]$ |
| | | $6u_0^1u_1^1u_2^1$ | | $[6,0,1,1,1,2,1]$ |

**Tabla** 3.1: Ejemplo de conversión de polinomios simbólicos a vectores numéricos

De manera general un polinomio de Adomian, pude tener la siguiente forma

$$A_n = a_1u_0^{p_0}u_1^{p_1}\cdots u_k^{p_k} + a_2u_0^{p_0}u_1^{p_1}\cdots u_k^{p_k} + \cdots + a_tu_0^{p_0}u_1^{p_1}\cdots u_k^{p_k} \tag{3.1}$$

Donde $a_t$ son constantes que ponderan a cada uno de los términos $u_k$ son valores calculados que corresponden a los de la ecuación (2.7) y $p_k$ son las potencias a las cuales cada uno de estos valores están elevados. Nótese que puede faltar alguno de los $u_k$ en los términos del polinomio, no siempre los tiene todos.
Se puede notar, que la conversión de la tabla 3.1 se puede generalizar siguiendo los pasos:

1. Separar, lo términos del polinomio de Adomian que se quiera almacenar.
2. Colocar el ponderador $a_t$ de cada uno de los términos, como primer elemento del vector que lo representará.

3. Los subíndices de $u_k$ y las potencias $p_k$ se colocaran enseguida, respectivamente para todas las $u_k$ que contenga el término del polinomio.

De manera general, el primer término de (3.1) quedaría almacenado en un vector que tendrá la forma

$$[a_1, 0, p_0, 1, p_1, \cdots, k, p_k]$$

Con esta forma de conversión lo que se gana es que ahora en lugar de evaluar variables simbólicas en MATLAB, lo cual es computacionalmente costos, podemos simplemente acceder a la posición del vector de $u$'s con el subíndice almacenado y elevarlo a la potencia indicada con el superíndice también almacenado. Una vez hecho ésto, solo se debe multiplicar por el ponderador que sabemos que siempre esta al principio del vector de cada uno de los términos. Lo cual convierte la evaluación de simbólicos en un acceso a vectores, multiplicaciones y elevar a alguna potencia. Se hace lo indicado en la tabla 3.1, para cada uno de los polinomios que se hayan calculado y almacenado en la celda. Entonces en MATLAB

- A cada término de un polinomio $A_n$ se le asocia un vector.

  A cada polinomio $A_n$ una celda conteniendo a los vectores asociados a sus términos, o un vector de vectores.

  Un conjunto de polinomios se representa por una celda de celdas.

Una vez que todos los polinomios hayan sido afectados por esta conversión, se almacenan en un archivo .txt, el cual será utilizado en C++ para poder precargar lo valores de los polinomios y no tener que estar haciendo el cálculo de los mismos cada vez que se quiera utilizar el algoritmo.

### 3.2.2. C++

Debido al acomodo que se le dio a los vectores que representan los términos de cada uno de los polinomios, es fácil ver que siempre se van a acceder a las posiciones de los vectores en orden secuencial.
Se puede pensar en hacer un 'prisma' de vectores, el cual será llenado a partir del archivo .txt generado por MALTLAB que contiene la información de los polinomios de Adomian. Donde sus ejes, estarán representando lo siguiente

- Uno de sus ejes es el orden de la no-linealidad con la que se este tratando (por ejemplo $Nu = u^2, Nu = u^3$,etc).

  Otro eje es el número del polinomio de Adomian, calculado para cada una de las no-linealidades (por ejemplo $A_0, A_1$,etc).

- El último eje representa cada uno de los términos que contengan los polinomios.

El prisma de vectores, tiene la forma de la figura 3.1

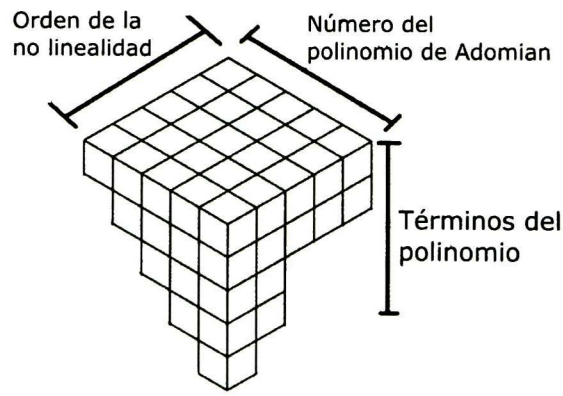



**Figura** 3.1: Ejes del prisma de vectores

Supongamos que se tiene una no-linealidad de orden $m$, es decir $Nu = u^m$, y se quiere evaluar el polinomio número $n$, es decir $A_n$, donde este polinomio tiene $K$ términos, entonces la formar de calcular el valor del polinomio es evaluar cada uno de sus términos que están contenidos, cada uno, en una lista en la posición $k$, para

el $k$-ésimo término.
Podemos observar esto de forma gráfica en el prisma de vectores en la figura 3.2

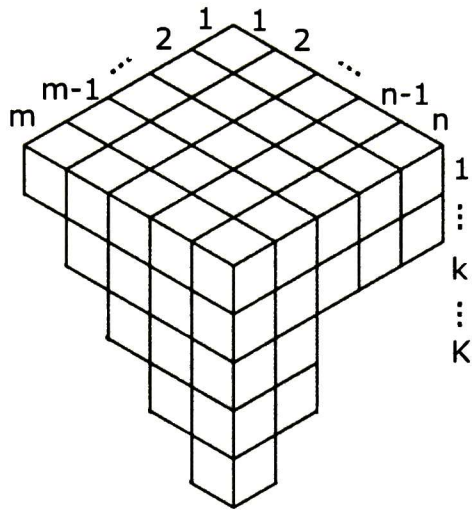



**Figura** 3.2: Ubicación de los términos de un polinomio en el prisma de vectores

Se observa en las tablas 2.1 y 2.2, que si el número de polinomios que se desea calcular es mayor, también el número de términos que contengan cada uno de ellos será mayor. Por tanto, la motivación para realizar el acomodo de los vectores de esta manera, es que en cualquier polinomio $A_n$ sus términos no son dependientes entre ellos, esto permite pensar en que de manera paralela se calcule el valor numérico de cada uno de ellos, posteriormente sumarlos y encontrar el valor del polinomio ya evaluado. Este acomodo nos permite eso, ya que se podría acceder a cada uno de los vectores por separado, hacer el cálculo y después sumar los resultados.

### 3.2.3. CUDA C

Como se puede observar, la representación en forma de un prisma de vectores que se usó para poder ejecutar el algoritmo en C++ y MATLAB es irregular, ya que el número de términos en cada polinomio varia con forme el número, de estos polinomios calculados, va aumentando. De la misma forma, cada uno de estos términos es irregular, ya que no todos tienen el mismo número de variables a evaluar.

El lenguaje de programación CUDA C recibe en sus funciones sólo punteros, por lo tanto se debe encontrar la forma de representar este arreglo de cuatro-dimensiones en un arreglo de una sola dimensión para posteriormente poder acceder de manera lineal al mismo. Se nota que si los vectores que se están manejando fueran regulares, es decir todos tuvieran la misma longitud y fueran el mismo número de términos en cada polinomio, sería una tarea sencilla empezar a "desdoblar" nuestro arreglo y posteriormente acceder a él con lo índices originales solo ponderados por la posición deseada. Pero, la irregularidad en la longitud de los vectores, representa un problema al momento de querer linealizarlo.

Por tanto, se propuso una forma para poder linealizar un arreglo cuatri-dimensional, que facilmente puede ser la idea extendida a una arreglo $n$-dimensional que también sea irregular en la longitud de sus vectores.

#### Forma lineal propuesta para un arreglo cuatri-dimensional irregular

Si los vectores generados por los polinomios tuvieran la misma longitud, el mismo número de términos, sería una tarea sencilla convertir el arreglo de cuatro dimensiones en un arreglo lineal, ya que sólo se tendría que calcular el número de posiciones que hay que recorrer entre un vector y otro. Se tendría

$$A_n = A_n(K, u_0, u_1, \cdots, u_n) \tag{3.2}$$

donde K es el número de términos que tiene el polinomio de Adomian, de acuerdo a la ecuación (3.1) podemos descomponer un polinomio como

$$A_n = \sum_{i=1}^{K} (a_i u_0^{n_0^i} u_1^{n_1^i} \cdots u_n^{n_n^i}) \tag{3.3}$$

sabiendo el número términos de los polinomios, se suma su longitud y se reserva el espacio requerido, así

$$for\ each\ i, \qquad \sum_{j=1}^{n} 1 = n \tag{3.4}$$

de acuerdo al acomodo propuesto para los términos, la longitud de cada uno de los vectores sería entonces $2n + 1$, $2n$ por el almacenamiento del número de $u_i$ y de su potencia correspondientes y 1 por el valor de ponderación del término $a_i$. De aquí es realizar operaciones para reservar el espacio en la memoria y poder acomodarlos y acceder a ellos dando saltos equidistantes a través del vector completo.
Así se podría realizar para un arreglo de cuatro dimensiones regular, pero se tomo otro camino, debido a la irregularidad de los términos en los polinomios de Adomian y a que no siempre tienen todos los valores de $u_i$. Se propusó una forma de ahorrar memoria y sólo almacenar los datos necesarios para acceder a los vectores generados a través de punteros y del número de elementos que contenga el vector en cuestión. Para representar nuestro arreglo irregular de manera lineal, serán necesarios dos vectores de datos:

1. Uno que contenga la información general del arreglo de cuatro dimensiones, es decir cuantos vectores hay hacía cada dirección y cuantos elementos contienen los vectores de manera particular, así con un apuntador de donde se encuentra cada uno ellos en la memoria para poder acceder directamente a uno en particular, sin necesidad de tener que recorrerlos todos hasta encontrar el deseado.

2. El segundo vector será de saltos, el cual contendrá en cada una de sus posiciones el número de elementos que hay que brincar del primer vector para poder llegar directamente al vector buscado.

El primer vector lo podemos ver de la siguiente manera:

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $TP^{(1,1)}$ | $*ptr^{(1,1,1)}$ | NE | $*ptr^{(1,1,2)}$ | $\cdots$ | $TP^{(1,2)}$ | $\cdots$ | $TP^{(m,n)}$ | $*ptr^{(m,n,1)}$ | NE | $\cdots$ |

Este vector será conocido como vector de posiciones de nuestro arreglo.
Donde $TP^{(m,n)}$ representa el número de términos en el polinomio al que se quiere acceder, es decir, de acuerdo a la figura 3.1 la $m$ representa el orden de la no-linealidad que va a representar el polinomio y la $n$ representa el orden del polinomio que se quiere evaluar correspondiente a esa no-linealidad. Por tanto $TP^{(m,n)}$ nos daría la información de la profundidad del prisma para el polinomio.
$*ptr^{(m,n,k)}$ representa un puntero, el cual nos dice la localidad de memoria en donde empieza el vector correspondiente a el termino $k$-ésimo del polinomio.
$NE$ es el número de elementos que hay en el vector al que hace referencia el puntero anterior.
Notamos que el número de punteros que representan cada uno de los términos del polinomio, dependerá del número almacenado en $TP^{(m,n)}$.

La motivación para realizar este acomodo en un vector, es que al momento de querer ejecutar el algoritmo en el lenguaje de programación CUDA C que permite realizar operaciones sobre un GPU, las funciones sólo reciben punteros. Esto hace que sea imposible saber la longitud de los vectores y la profundidad de los arreglos sin información sobres los mismos.
El vector de posiciones propuesto anteriormente es capaz de proporcionar esta información. Su desventaja, es que al ser irregular la profundidad y dimensión de nuestros vectores en el arreglo cuatri-dimensional, no es posible saber como llegar de un $TP^{(m,l)}$ a un $TP^{(m,r)}$ donde $l > r$ o $l < r$ sin recorrer y evaluar uno por uno los $TP$ y avanzando las localidades de memoria necesarias.
Lo cual nos lleva a proponer un vector de 'saltos', el cual tendrá como propósito brindar la información necesaria para desplazarnos en el vector de posiciones anterior entre $TPs$. El cual tendrá la siguiente forma

| | | | | |
|---|---|---|---|---|
| $Pos_{TP^{(1,1)}}$ | $Pos_{TP^{(1,2)}}$ | $\cdots$ | $Pos_{TP^{(m,n-1)}}$ | $Pos_{TP^{(m,n)}}$ |

Donde cada uno de los elementos de este vector, representa el número de posiciones que hay que recorrer del primer vector, el vector de posiciones, para acceder directamente al $TP^{(m,n)}$ deseado sin necesidad de recorrer todo el vector original.

Recordando el ejemplo que se dio para la conversión de los polinomios en vectores de la tabla 3.1. Se escribirá de nuevo para que la lectura sea mas sencilla

| | | | | |
|---|---|---|---|---|
| | | $3u_0^2u_3^1$ | | $[3,0,2,3,1]$ |
| $A_3 = 3u_0^2u_3^1 + 1u_1^3 + 6u_0^1u_1^1u_2^1$ | $\rightarrow$ | $1u_1^3$ | $\rightarrow$ | $[1,1,3]$ |
| | | $6u_0^1u_1^1u_2^1$ | | $[6,0,1,1,1,2,1]$ |

**Tabla** 3.2: Ejemplo de conversión de polinomios simbólicos a vectores numéricos

El vector generado en una dimensión quedaria de la siguiente forma

$$A_3 = [3,0,2,3,1,1,1,3,6,0,1,1,1,2,1]$$

Reescribiendo la ecuación 3.2 para el caso irregular, tenemos

$$A_n = A_n(K, NE_0, NE_1, \cdots, NE_K, u_0, u_1, \cdots, u_n) \tag{3.5}$$

donde la variable $NE_K$ agregada, representa el número de elementos que contiene cada uno de los $K$ términos en el polinomio. Si se tiene el término general de un polinomio

$$Termino\ K - ésimo\ del\ polinomio\ = a_K u_0^{n_0^K} u_1^{n_1^K} \cdots u_n^{n_n^K}$$

entonces $NE_K$ se calcula por

$$NE_K = 2\sum_{j=1}^{n} 1 + 1 = 2n + 1 \tag{3.6}$$

Como en la ecuación 3.4.
De la ecuación (3.5) tenemos que $K = 3$ para éste caso, y que

$NE_0 = 2 \times 2 + 1 = 5$

$NE_1 = 2 \times 1 + 1 = 3$

- $NE_2 = 2 \times 3 + 1 = 7$

Del vector de posiciones propuesto tenemos que $TP^{(1,1)} = K = 3$ para el ejemplo. Es necesario tomar las localidades de memoria donde empieza cada uno de los términos del polinomio y almacenarlos en punteros, entonces

$$A_3 = [\mathbf{3}, 0, 2, 3, 1, \mathbf{1}, 1, 3, \mathbf{6}, 0, 1, 1, 1, 2, 1]$$

de os valores que están en negritas se tomará su localidad de memoria y se almacenara en punteros, se tiene

- $*ptr^{(1,1,1)} = \&A_3[0]$
- $*ptr^{(1,1,2)} = \&A_3[5]$
- $*ptr^{(1,1,3)} = \&A_3[8]$

Así el vector de posiciones para el ejemplo quedaría como

| | | | | | | |
|---|---|---|---|---|---|---|
| 3 | $*ptr^{(1,1,1)}$ | 5 | $*ptr^{(1,1,2)}$ | 3 | $*ptr^{(1,1,3)}$ | 7 |

## 3.3. Pseudocódigo para análisis de dependencia

Para realizar el cálculo de los $L$ eigenvalores en una matriz, se tiene que llevar a cabo el siguiente pseudocódigo:

```
1: L = Número de eigenvalores en la matriz
2: N = Número de valores de la descomposición de la sumatoria para encontrar
   un eigenvalor
3: for l = 0 hasta l = (L − 1) do
4:     x(l) = 0
5:     for i = 0 hasta i = (N − 1) do
6:         Calcular u_i
7:         x(l) = x(l) + u(i)
8:     end for
9:     Aplicar el método de horner con el eigenvalor calculado x sobre el polinomio
   característico y reducirlo de orden
10: end for
```

A continuación se explicará parte por parte el pseudocódigo aplicando un análisis de dependencia a cada una de ellas.
Reescribiendo la ecuación (2.7)

$$x = \sum_{i=0}^{N-1} u_i \tag{3.7}$$

Se observa que la forma de representarla de manera computacional es mediante el siguiente fragmento de pseudocódigo, el cual representa la ecuación (3.7)

```
1: x(l) = 0
2: for i=0:(N − 1) do
3:     x(l) = x(l) + u(i)
4: end for
```

Donde $N$ es el número de términos de la sumatoria de la ecuación 3.7.

La forma que tiene esta parte del algoritmo, no permite realizar ninguna iteración en paralelo, ya que dependiendo de la precisión que se quiera dar al algoritmo sera necesario sumar más valores de $u_i$, es decir $N$ será más grande y se tendrán que calcular todos. O truncar la serie de la ecuación (3.7) cuando se alcancé la precisión deseada.
Cada uno de los valores representados como $u(i)$ en el pseudocódigo anterior se puede calcular mediante la ecuación (2.26), reescribiendola tenemos

$$u_{i+1} = -\frac{a_0}{a_{n-1}}A_i - \frac{a_1}{a_{n-1}}B_i - \cdots - \frac{a_{n-2}}{a_{n-2}}Z_i \; ; \; i \geq 0 \tag{3.8}$$

Si los valores correspondientes a $A_i, B_i, \cdots, Z_i$ (Polinomios de Adomian) con sus respectivos coeficientes, ya fueron calculados y se almacenan en un vector $\mathbf{P}$, podemos reescribir la ecuación de la siguiente forma

$$u_{i+1} = P(0) + P(1) + \cdots + P(n-2) + P(n-1) \; ; \; i \geq 0 \tag{3.9}$$

Donde, $P(0) = -\frac{a_0}{a_{n-1}}A_i, P(1) = -\frac{a_1}{a_{n-1}}B_i$ y así sucesivamente.
Y su pseudocódigo, viene dado por

```
1: u(i + 1) = 0 para i > 0
2: for j = 0 : (n − 1) do
3:     u(i + 1) = u(i + 1) + P(j)
4: end for
```

Donde $n$ es el número de polinomios, ya ponderados, que se deben evaluar debido al orden de las no-linealidades que se presente en el problema.
Al igual que cuando se analizó el pseudocódigo anterior, se observa que no se pueden realizar operaciones en paralelo, debido a que la variable de entrada de la iteración actual, también depende de la variable de salida de la iteración anterior.
Juntando los pseudocódigos, tendríamos

```
1: x(l) = 0
2: u(0) = a
3: for i = 0 : (N − 1) do
4:     x(l) = x(l) + u(i)
5:     u(i + 1) = 0
6:     for j = 0 : (n − 1) do
7:         u(i + 1) = u(i + 1) + P(j)
8:     end for
9: end for
```

Si se analiza la estructura de los polinomios de Adomian, tomando como ejemplo el polinomio $A_5$ de la tabla 2.2

$$A_5 = 6u_2u_3u_0 + 3u_2^2u_1 + 3u_1^2u_3 + 3u_0^2u_5 + 6u_1u_4u_0$$

Si reescribimos los terminos en el polinomio anteior, tenemos

$$A_5 = a_5(0) + a_5(1) + a_5(2) + a_5(3) + a_5(4)$$

donde $a_5(0) = 6u_2u_3u_0$ y de manera similar para los demás. Se puede generalizar para cualquier polinomio, de modo que un polinomio $A_i$ estará compuesto de $F$ términos, que se pueden escribir de la forma:

$$A_i = a_i(0) + a_i(1) + a_i(2) + a_i(3) + \cdots + a_i(F-1) \tag{3.10}$$

donde cada uno de los $a_i$ representará un término del polinomio original.
Tomando como punto de partida para el análisis, la ecuación (2.9), podemos escribir que:

$$\begin{aligned}
&u(0), u(1), u(2), \cdots, u(i) \quad \textit{son estimaciones o valores calculados.} \quad y \\
&a_i(0) = f(u(0), u(1), u(2), \cdots, u(i)) \\
&a_i(1) = g(u(0), u(1), u(2), \cdots, u(i)) \\
&\vdots \\
&a_i(F-1) = h(u(0), u(1), u(2), \cdots, u(i))
\end{aligned} \tag{3.11}$$

Se observa que cada uno de los términos del polinomio se calcula con las estimaciones obtenidas en las iteraciones anteriores.
También, las variables de salida $a_i(t)$, no dependen de las variables de entrada $u(0), u(1), u(2), \cdots, u(i)$ o viceversa, ya que nunca tratan de leer o escribir la misma localidad de memoria, debido a ésto, todos ellos se pueden evaluar al mismo tiempo y posteriormente sumarlos.
De modo que el pseudocódigo sería:

```
1: for all f = 0 : F − 1 do
2:     Calcular  a_i(f)
3: end for
4: A_i = 0
5: for t = 0 : F − 1 do
6:     A_i = A_i + a_i(f)
7: end for
```

De manera similar, en la ecuación (3.9) todos los términos $P(0), P(1), \cdots, P(n)$ dependen de sus respectivos polinomios $A_i, B_i, \cdots, Z_i$. A cada uno de éstos polinomios le corresponde el análisis de la ecuación (3.11), por lo que se tiene

$$\begin{aligned} &u(0), u(1), u(2), \cdots, u(i) \quad \textit{son estimaciones o valores calculados.} \quad y \\ &A_i = f(u(0), u(1), u(2), \cdots, u(i)) \\ &B_i = g(u(0), u(1), u(2), \cdots, u(i)) \\ &\vdots \\ &Z_i = h(u(0), u(1), u(2), \cdots, u(i)) \end{aligned} \tag{3.12}$$

Se observa que cada uno de los polinomios, se calcula con las estimaciones obtenidas en las iteraciones anteriores.
También, las variables de salida $A_i, B_i, \cdots, Z_i$, no modificarán las variables de entrada $u(0), u(1), u(2), \cdots, u(i)$, entonces todos los polinomios $A_i, B_i, \cdots, Z_i$ se pueden evaluar al mismo tiempo y posteriormente sumarlos.
De modo que el pseudocódigo sería:

1: **for all** $k = 0 : n - 1$ **do**
2: $\quad$ *Calcular* $P(k)$
3: **end for**
4: $u_{i+1} = 0$
5: **for** $k = 0 : n - 1$ **do**
6: $\quad u_{i+1} = u_{i+1} + P(k)$
7: **end for**

## 3.4. Análisis de complejidad

Para el cálculo de la complejidad del algoritmo, se irán tomando fragmentos del pseudocódigo y analizandose por separado para que sea más sencillo y comprensible el análisis de acuerdo a [8].
Se empieza por analizar el siguiente fragmento de pseudocódigo

```
1: A_i = 0
2: for t = 0 : F − 1 do
3:     Calcular  a_i(f)
4:     A_i = A_i + a_i(f)
5: end for
```

A continuación, en el número de la sentencia del pseudocódigo, se pone el costo o trabajo de la misma

```
1: c_1 1
2: c_2(F + 1)
3: c_3 LF
4: c_4 F
```

De acuerdo a [8], $c_1, c_2, c_3, c_4$ son constantes de tiempo que tarda el procesador que se vaya a usar en ejecutar las sentencias. Las cuales se pueden juntar en una sola constante de tiempo general para el pseudocódigo.
Para calcular la complejidad completa del pseudocódigo $T_1(LF)$, se tiene que sumar cada uno de los costos de las sentencias. Así, tenemos entonces

$$T_1(LF) = c_1 + c_2(F + 1) + c_3LF + c_4F$$

se juntan las constantes en una, llámese $C'$ Como $LF > F + 1 > F$ para todos los casos, entonces sólo se toma a $LF$ como complejidad, ya que es el peor caso que se puede tener, es el que se toma para la complejidad. Aclarar que $L$ es un parámetro que viene de otra parte del algoritmo, más adelante se mostrará. Entonces, como

complejidad de este pseudocódigo es

$$T_1(n) \in \mathcal{O}(C'LF)$$

La complejidad $T_1(LF)$, se usa en la sentencia 3, del siguiente fragmento de pseudocódigo

```
1: u(i + 1) = 0
2: for j = 0 : (n − 1) do
3:     u(i + 1) = u(i + 1) + P(j)
4: end for
```

De manera similar, se escribe la complejidad de cada una de las sentencias

```
1: c1
2: c2(n + 1)
3: c3n(C'LF)
```

Sumando la complejidad de cada una de las sentencias tenemos

$$T_2(nLF) = c_1 + c_2(n+1) + c_3 nC'(LF)$$

Juntando las constantes, incluyendo $C'$, en una sola $C''$ se tiene

$$T_2(nLF) = C''((n+1) + nLF)$$

pero como $nLF > (n+1)$ para todos los casos, se puede escribir como

$$T_2(nLF) \in \mathcal{O}(C''nLF)$$

La complejidad $T_2(nLF)$, será usada en el siguiente pseudocódigo

1: $L$ = Número de eigenvalores en la matriz
2: $N$ = Número de valores de la descomposición de la sumatoria para encontrar un eigenvalor
3: **for** $l = 0$ hasta $l = (L - 1)$ **do**
4: $x(l) = 0$
5: **for** $i = 0$ hasta $i = (N - 1)$ **do**
6: Calcular $u_i$
7: $x(l) = x(l) + u(i)$
8: **end for**
9: Aplicar el método de horner con el eigenvalor calculado $x$ sobre el polinomio característico y reducirlo de orden
10: **end for**

Escribiendo las complejidades de cada una de las sentencias, tenemos

1: $c_1 1$
2: $c_2 1$
3: $c_3(L+1)$
4: $c_4(L)$
5: $c_5(n+1)(L)$
6: $c_6(NL)(C''nLF)$
7: $c_7 NL$
8:
9: $c_8(L+1)(L)$

El método de horner, de acuerdo a [8] su complejidad es $L$.

$$T_3(NnLF) = c_1+c_2+c_3(L+1)+c_4L+c_5(n+1)(L)+c_6(NL)(C''nLF)+c_7(NL)+c_8(L+1)(L)$$

Juntando las constantes, incluyendo $C''$, en una sola $C'''$. Se tiene que $L < (L+1) < (L+1)(L)$ y $(NL) < (NnL^2F)$ para todos los casos, tenemos entonces

$$T_3(NnLF) = C'''((L+1)(L) + (NnL^2F) + (n+1)(L))$$

y como $(n+1)(L) < (NnL^2F)$ y $(L+1)(L) < (NnL^2F)$ se cumple siempre, se

puede escribir como

$$T_3(NnLF) \in \mathcal{O}(C'''NnL^2F)$$

$T_3(NnLF)$ es la complejidad del algoritmo, entonces $T(NnLF) = T_3(NnLF)$, se tiene

$$T(NnLF) \in \mathcal{O}(NnL^2F)$$

Donde $N$ es la cantidad de valores en los cuales se descompuso la sumatoria de la ecuación (2.7), $n$ es el número de polinomios de Adomian que se evaluó para cada una de las $u_i$, $L$ es el orden de no-linealidades que se tienen y $F$ es la cantidad máxima de términos que pueden tener los polinomios de Adomian, se toma el máximo porque es el peor de los casos.

## 3.5. Reescritura de los polinomios

De acuerdo a lo visto en la sección anterior, hay forma de calcular los términos de cada uno de los polinomios de Adomian generados por las no-linealidades de manera paralela.

Cuando se va a calcular el siguiente valor de la suma infinita de valores en los que se descompuso uno de los valores a encontrar, representada por la ecuación (2.7), o en el caso computacional representado por la serie truncada de la ecuación (3.7), se tiene que ponderar cada uno de los polinomios ya evaluados correspondientes a las diferentes no-linealidades. Lo cual implica que cada uno de los términos se debe calcular siempre partiendo desde cero, es decir, sólo de los valores previamente almacenados en las iteraciones anteriores.

Aunque, analizando y reorganizando los polinomios de Adomian, se puede notar las similitudes que existen entre algunos de sus términos, aun cuando no pertenezcan a la misma no-linealidad. Para ejemplificarlo, en la siguiente tabla se puede ver la comparativa entre diferentes polinomios, calculados a partir de distintas no-linealidades.
En la tabla 3.3, se observan diferentes polinomios de Adomian para diferentes no-linealidades, se puede observar que para un $A_i$ no difiere mucho la forma en que está escrito comparado con los de las otras no-linealidades.
Entonces, es posible que algunos valores solo se tengan que calcular una vez, ser almacenados y después ser reutilizados en el calculo de algún otro polinomio, lo cual resultaría en un disminución de multiplicaciones, y por ende disminuiría el tiempo

| $\mathbf{A_i}$ | $\mathbf{A_0}$ | $\mathbf{A_1}$ | $\mathbf{A_2}$ |
|---|---|---|---|
| $\mathbf{u^2}$ | $u_0^2$ | $2u_0u_1$ | $2u_0u_2 + u_1^2$ |
| $\mathbf{u^3}$ | $u_0^3$ | $3u_0^2u_1$ | $3u_0^2u_2 + 3u_1^2u_0$ |
| $\mathbf{u^4}$ | $u_0^4$ | $3u_0^3u_1$ | $4u_0^3u_2 + 6u_1^2u_0^2$ |

**Tabla** 3.3: Comparativa de diferentes polinomios, para diferentes no-linealidades

| $A_i$ | $A_0$ | $A_1$ | $A_2$ |
|---|---|---|---|
| $u^2$ | $a^2_{0_1}$ | $2a^2_{1_1}$ | $2a^2_{2_1} + a^2_{2_2}$ |
| $u^3$ | $a^3_{0_1}$ | $3a^3_{1_1}$ | $3a^3_{2_1} + 3a^3_{2_2}$ |
| $u^4$ | $a^4_{0_1}$ | $3a^4_{1_1}$ | $4a^4_{2_1} + 6a^4_{2_2}$ |

**Tabla** 3.4: Comparativa de diferentes polinomios, afectados por diferentes no-linealidades (Modificada)

| $A_i$ | $A_0$ | $A_1$ | $A_2$ |
|---|---|---|---|
| $u^2$ | $a^2_{0_1} = u^2_0$ | $a^2_{1_1} = u_0 u_1$ | $a^2_{2_1} = u_0 u_2, \quad a^2_{2_2} = u^2_1$ |
| $u^3$ | $a^3_{0_1} = a^2_{0_1} u_0$ | $a^3_{1_1} = a^2_{1_1} u_0$ | $a^3_{2_1} = a^2_{2_1} u_0, \quad a^3_{2_2} = a^2_{2_2} u_0$ |
| $u^4$ | $a^4_{0_1} = a^3_{0_1} u_0$ | $a^4_{1_1} = a^3_{1_1} u_0$ | $a^4_{2_1} = a^3_{2_1} u_0, \quad a^4_{2_2} = a^3_{2_2} u_0$ |

**Tabla** 3.5: Cambio de variables en los polinomios

de procesamiento del algoritmo.
De acuerdo a lo anterior, podemos reescribir la tabla 3.3 de manera que se haga énfasis en el reuso de términos previamente calculados, para el cálculo de los términos, pertenecientes a diferentes polinomios. Se utilizará la notación $a^k_{i_j}$, donde $i$ hará referencia al número del polinomio, $j$ es el número del término del polinomio y $k$ es la no-linealidad a la que pertenece, cabe hacer mención que no necesariamente $a^k_{i_j}$ representará un termino completo del polinomio, puede ser que soló sean algunos de los elementos del término, una vez dicho ésto tenemos la tabla 3.4.

Donde los cambios realizados en cada uno de los polinomios con respecto al original son los mostrados en la tabla 3.5.

De la tabla 3.5, se puede ver que los cambios hechos entre las nuevas variables propuestas son muy pequeños, es sólo multiplicarse por una de las variables originales previamente almacenada. Lo cual podría ser de gran ayuda si se decide ir guardando los valores nuevos, ya que se irían disminuyendo el número de operaciones para el cálculo al no tener que realizar todo desde cero. Si se revisan los otros polinomios se puede observar que la relación es similar a la descrita en la tabla 3.5.

| $\mathbf{u}$ | $\mathbf{u^2}$ | $\mathbf{u^3}$ | $\mathbf{u^4}$ | $\cdots$ | $\mathbf{u^n}$ |
|---|---|---|---|---|---|
| $u_0$ | $u_0u_0$ | $u_0u_0u_0$ | $u_0u_0u_0u_0$ | $\cdots$ | $u_0u_0\cdots u_0u_0$ |
| $u_1$ | $u_0u_1$ | $u_0u_0u_1$ | $u_0u_0u_0u_1$ | $\cdots$ | $u_0u_0\cdots u_0u_1$ |
| $u_2$ | $u_0u_2$ | $u_0u_0u_2$ | $u_0u_0u_0u_2$ | $\cdots$ | $u_0u_0\cdots u_0u_2$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\cdots$ | $\vdots$ |
| $u_7$ | $u_0u_m$ | $u_0u_0u_m$ | $u_0u_0u_0u_m$ | $\cdots$ | $u_0u_0\cdots u_0u_m$ |
| $u_8$ | $u_1u_0$ | $u_0u_1u_0$ | $u_0u_0u_1u_0$ | $\cdots$ | $u_0u_0\cdots u_1u_0$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\cdots$ | $\vdots$ |
| $u_{m-1}$ | $\vdots$ | $\vdots$ | $\vdots$ | $\cdots$ | $\vdots$ |
| $u_m$ | $u_mu_{m-1}$ | $\vdots$ | $\vdots$ | $\cdots$ | $\vdots$ |
| | $u_mu_m$ | $u_mu_mu_{m-1}$ | $\vdots$ | $\cdots$ | $\vdots$ |
| | | $u_mu_mu_m$ | $u_mu_mu_mu_{m-1}$ | $\cdots$ | $\vdots$ |
| | | | $u_mu_mu_mu_m$ | $\cdots$ | $\vdots$ |
| | | | | $\cdots$ | $u_mu_m\cdots u_mu_{m-1}$ |
| | | | | $\cdots$ | $u_mu_m\cdots u_mu_m$ |

**Tabla** 3.6: Propuesta de variables almacenadas para los polinomios de Adomian (Matriz **G**)

Se puede pensar en dos pasos principales para hacer esto:

El primer paso, sería encontrar una manera de guardar estos valores para que sea fácil acceder a ellos y almacenarlos cada vez que se calcule un valor nuevo.

- El segundo paso, sería reescribir los polinomios de acuerdo a esta nueva forma de almacenamiento de valores para que cada uno de los términos tenga una representación única con la nueva forma propuesta.

De acuerdo al primer paso, se propone almacenar estos valores en una matriz de la forma expuesta en la tabla 3.6. Llamada matriz **G**.

Esta matriz **G**, de valores guardados, tendrá una longitud de $n \times m^2$ y como se puede observar, será irregular ya que las últimas columnas tendrán más valores al-

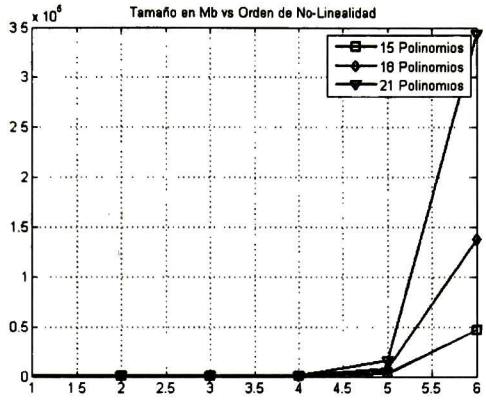



**Figura** 3.3: MB necesarios para almacenar los valores precalculados para 15, 18 y 21 polinomios mediante permutaciones

macenados que las primeras debido ya que el número de elementos para su cálculo va creciendo.
Se puede ver que la cantidad de valores para almacenar va creciendo exponencialmente con respecto al número de no-linealidades que se tengan. Por tanto está forma de almacenamiento sirve solo para cuando la cantidad de no-linealidades es pequeña, ésto debido a la gran cantidad de localidades de memoria necesarias para almacenar todos los valores.

La organización de esta matriz **G** está dada como permutaciones del número total de valores en los que se descompuso la sumatoria (2.7). Al usar permutaciones el número de valores para almacenar crece bastante ya que se están repitiendo los calculos de los mismos valores, es por eso que se propone un cambio para que el número de valores a almacenar disminuya, evitando la repetición de algunos términos. En la figura 3.3 se observa que se requiere de mucha memoria para almacenar los valores para cuando la no-linealidad es de orden 6.

Como ejemplo se puede ver en la tabla 3.6 en la columna correspondiente a $u^2$, el valor del término $u_0u_1$ es el mismo que el de $u_1u_0$, por tanto se podría almace-

nar sólo una vez. Otro ejemplo es en la columna correspondiente a $u^3$, el valor del término $u_0u_0u_1$ es el mismo que el de $u_0u_1u_0$, de igual manera se puede almacenar sólo una vez. De la misma manera, se observa que en todas las columnas se tiene un comportamiento similar y que conforme va creciendo el orden de la no-linealidad, el número de términos también van a ir aumentando.
Como una forma de optimizar este problema, se propone que en lugar de permutaciones, se realicen combinaciones con repetición. Si se realiza esta cambio, ahora la cantidad de elementos de las columnas estará dado de la siguiente manera

$$Numero \quad de \quad elementos = \frac{(m+r-1)!}{r!(m-1)!} \tag{3.13}$$

Donde $m$ es la cantidad de términos $u_i$ en los que se descompuso la ecuación (2.7) y $r$ es el orden de la no-linealidad que representa la columna de la matriz $\mathbf{G}$. La nueva forma de la matriz es muy similar pero sin los elementos repetidos y el número de los mismos variará de acuerdo a (3.13). La nueva forma de la matriz sería la dada en la tabla 3.7.

Ahora que se tienen los valores almacenados de esta forma, se puede pensar en una nueva forma de escribir los polinomios donde cada uno de los términos se sustituya de ser posible con algún elemento $(g_{11}, g_{12}, \cdots) \in \mathbf{G}$.
Para ejemplificarlo, tomemos como ejemplo los primeros tres polinomios de las no-linealidades $u^2, u^3, u^4$

| $\mathbf{u}$ | $\mathbf{u^2}$ | $\mathbf{u^3}$ | $\mathbf{u^4}$ | $\cdots$ | $\mathbf{u^n}$ |
|---|---|---|---|---|---|
| $u_0$ | $u_0u_0$ | $u_0u_0u_0$ | $u_0u_0u_0u_0$ | $\cdots$ | $u_0u_0\cdots u_0u_0$ |
| $u_1$ | $u_0u_1$ | $u_0u_0u_1$ | $u_0u_0u_0u_1$ | $\cdots$ | $u_0u_0\cdots u_0u_1$ |
| $u_2$ | $u_0u_2$ | $u_0u_0u_2$ | $u_0u_0u_0u_2$ | $\cdots$ | $u_0u_0\cdots u_0u_2$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\cdots$ | $\vdots$ |
| $u_7$ | $u_0u_m$ | $u_0u_0u_m$ | $u_0u_0u_0u_m$ | $\cdots$ | $u_0u_0\cdots u_0u_m$ |
| $u_8$ | $u_1u_1$ | $u_0u_1u_1$ | $u_0u_0u_1u_1$ | $\cdots$ | $u_0u_0\cdots u_1u_1$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\cdots$ | $\vdots$ |
| $u_{m-1}$ | $\vdots$ | $\vdots$ | $\vdots$ | $\cdots$ | $\vdots$ |
| $u_m$ | $u_{m-1}u_m$ | $\vdots$ | $\vdots$ | $\cdots$ | $\vdots$ |
| | $u_mu_m$ | $u_{m-1}u_mu_m$ | $\vdots$ | $\cdots$ | $\vdots$ |
| | | $u_mu_mu_m$ | $u_{m-1}u_mu_mu_m$ | $\cdots$ | $\vdots$ |
| | | | $u_mu_mu_mu_m$ | $\cdots$ | $\vdots$ |
| | | | | $\cdots$ | $u_{m-1}u_m\cdots u_mu_m$ |
| | | | | $\cdots$ | $u_mu_m\cdots u_mu_m$ |

**Tabla** 3.7: Propuesta de variables almacenadas para los polinomios de Adomian sin repetición (Nueva matriz **G**)

$$
\begin{aligned}
&Para \quad u^2 \\
A_0 =& u_0^2 \\
A_1 =& 2u_1u_0 \\
A_2 =& 2u_2u_0 + u_1^2 \\
&Para \quad u^3 \\
A_0 =& u_0^3 \\
A_1 =& 3u_1u_0^2 \\
A_2 =& 3u_2u_0^2 + 3u_1^2u_0 \\
&Para \quad u^3 \\
A_0 =& u_0^4 \\
A_1 =& 4u_1u_0^3 \\
A_2 =& 4u_2u_0^3 + 6u_0^2u_1^2
\end{aligned}
$$

La matriz correspondiente para estas cuatro no-linealidades, a partir de la tabla 3.7, tomando en cuenta que la sumatoria de la ecuación (2.7) se descompuso solo en tres valores, sería lo mostrado en la tabla 3.8

La tabla 3.8 representa la forma que tendrá la matriz **G**, reescribiendo los polinomios

| | | | |
|---|---|---|---|
| $u_0$ | $u_0u_0$ | $u_0u_0u_0$ | $u_0u_0u_0u_0$ |
| $u_1$ | $u_0u_1$ | $u_0u_0u_1$ | $u_0u_0u_0u_1$ |
| $u_2$ | $u_0u_2$ | $u_0u_0u_2$ | $u_0u_0u_0u_2$ |
| | $u_1u_1$ | $u_0u_1u_1$ | $u_0u_0u_1u_1$ |
| | $u_1u_2$ | $u_0u_1u_2$ | $u_0u_0u_1u_2$ |
| | $u_2u_2$ | $u_1u_1u_1$ | $u_0u_0u_2u_2$ |
| | | $u_1u_1u_2$ | $u_0u_2u_1u_1$ |
| | | $u_2u_0u_2$ | $u_0u_2u_1u_2$ |
| | | $u_2u_1u_2$ | $u_0u_2u_2u_2$ |
| | | $u_2u_2u_2$ | $u_2u_1u_1u_1$ |
| | | | $u_2u_1u_1u_2$ |
| | | | $u_2u_1u_2u_2$ |
| | | | $u_2u_2u_2u_2$ |

**Tabla** 3.8: Forma de la matriz resultante para cuatro no-linealidades

a partir de las posiciones de la matriz, tenemos:

$$
\begin{aligned}
&Para \quad u^2 \\
A_0 =& g_{12} \\
A_1 =& 2g_{22} \\
A_2 =& 2g_{32} + g_{42} \\
&Para \quad u^3 \\
A_0 =& g_{13} \\
A_1 =& 3g_{23} \\
A_2 =& 3g_{33} + 3g_{43} \\
&Para \quad u^3 \\
A_0 =& g_{14} \\
A_1 =& 4g_{24} \\
A_2 =& 4g_{34} + 6g_{44}
\end{aligned}
$$

Donde $g_{12} = u_0^2, g_{22} = u_1u_0, \cdots, g_{44} = u_0^2u_1^2$.

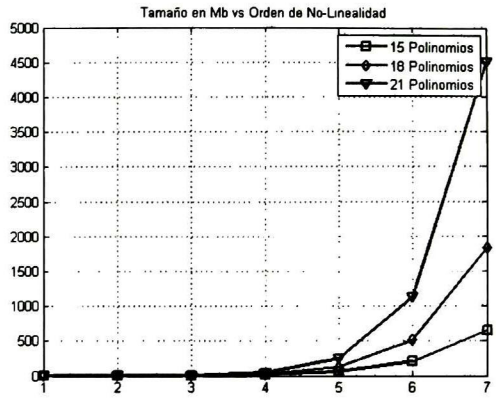



**Figura** 3.4: MB necesarios para almacenar los valores precalculados para 15, 18 y 21 polinomios

De la misma manera se podrían reescribir cada uno de los polinomios para eliminar operaciones multiplicativas cada vez que se calcula un polinomio y convertirlo sólo en un acceso de memoria. Pensando en que se pueda calcular está matriz de manera paralela cuando ya se tengan los valores principales $u_0, u_1, u_2, \cdots, u_n$.

El uso de está nueva propuesta para la evaluación de los polinomios, supone una reducción del número de multiplicaciones necesarias para la evaluación pero también implica que la cantidad de memoria que se debe tener para almacenar estos valores precalculados irá creciendo conforme aumenta el grado u orden de la no-linealidad y el número de términos en los que se descomponga la sumatoria de la ecuación (2.7) aumente. Tomando en cuenta ésto se hizo un análisis de memoria teniendo en consideración que se están almacenando valores flotantes para cada una de las celdas de nuestra matriz **G** propuesta en la tabla 3.7 y que los bytes necesarios para almacenar un flotante es 4. El análisis pretende mostrar como va aumentando la memoria (en MB) conforme crece el número de polinomios y el orden de la no-linealidad. En la figura 3.4 se observa como a partir de la no-linealidad de orden 6 el espacio de memoria necesario para almacenar nuestra matriz **G**, va creciendo de manera muy rápida para 18 y 21 polinomios. Lo cual dificulta el almacenamiento. Actualmente

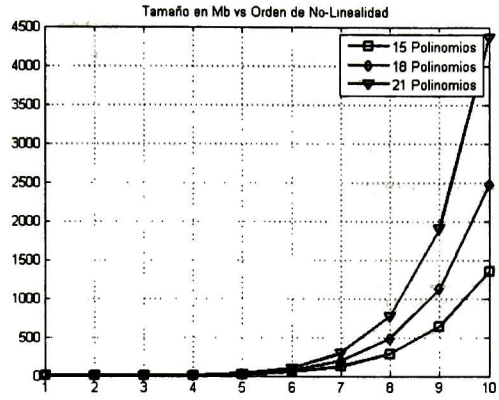



**Figura** 3.5: MB necesarios para almacenar los valores precalculados para 11, 12 y 13 polinomios

la tarjeta *Tesla K20* de Nvidia, tiene una capacidad de memoria de 5Gb.

Como una segunda opción, se puede proponer que el número de polinomios que se evalúen sea menor para poder almacenar más valores precalculados en la memoria, pero tendría que tomarse en cuenta que la precisión del algoritmo bajaría. Si éste es el caso, se tendría que analizar cual de las dos es la mas adecuada para la aplicación para la que se va utilizar el algoritmo.

En la figura 3.5 se observa como al disminuir la cantidad de polinomios que se quieren evaluar, la cantidad de memoria que necesitamos disminuye, lo cual permite que el orden de la no-linealidad aumente.

## 3.6. Suma de los elementos de un vector en forma paralela

Durante la elaboración de este trabajo se observó que hay una parte del algoritmo que por su naturaleza es secuencial.Por ejemplo, la sumatoria de los elementos de un vector o de los valores que se van calculando en cada iteración. Sea un vector de $n$ elementos, dado por

$$\mathbf{v} = [v_0, v_1, v_2, \cdots, v_{n-2}, v_{n-1}]$$

Una forma de realizar la sumatoria de todos los elementos es la siguiente

```
1: acum = 0
2: for i = 0 : n − 1 do
3:     acum = acum + v_i
4: end for
```

Esto no representa un problema cuando la longitud de los vectores no es muy grande, aproximadamente cuando $n < 2^9$ Cuando esta $n$ aumenta, el tiempo aumenta también ya que el número de operaciones que tiene que realizar es mayor. Pensando en que el número de términos en los polinomios de Adomian va creciendo conforme aumenta el orden de las no-linealidades y el número de polinomios deseados, va a llegar un momento en que la longitud de los elementos almacenados sea tan grande que empiece a consumir mucho tiempo en el cálculo de las sumatorias. Por tal motivo, se busca la forma de poder reducir el tiempo en esta parte del algortimos, bajando la complejidad de la misma. Representando el vector anterior $\mathbf{v}$ de valores como un arreglo de datos, llamese $\mathbf{v}'$. tenemos

$$\mathbf{v}' = \boxed{v'_0 \mid v'_1 \mid v'_2 \mid v'_3 \mid \cdots \mid v'_{n-4} \mid v'_{n-3} \mid v'_{n-2} \mid v'_{n-1}}$$

Donde este vector tiene $n$ valores y $n$ es un potencia de 2. Como el vector ya está almacenado y los valores no cambiarán, al menos antes de que se realice la suma de todos ellos, se propone que para que este algoritmo de complejidad $n$ se haga de $\log_2(n)$, se debe realizar $n/2$ sumas del vector al mismo tiempo. Tomando esto en cuenta y aprovechando que en los GPUs utilizados es factible tener muchos hilos

ejecutandose al mismo tiempo, se puede pensar en que cada hilo realice una suma de dos elementos del vector para reducir el tiempo de ejecución. De igual manera, cada una de las sumas resultantes de cada hilo, se almacenará en un nuevo vector $\mathbf{v}$, representado como $\mathbf{v}''$ Reescribiendo los elementos del vector anterior se tiene entonces en la parte inferior el hilo que estará realizando la suma y por la parte de arriba el lugar donde se estaría almacenando el resultado en el nuevo vector, así

$$\overbrace{\underbrace{v'_0 + v'_1}_{Hilo\ 0}}^{v''_0} \quad \overbrace{\underbrace{v'_2 + v'_3}_{Hilo\ 1}}^{v''_1} \quad \cdots \quad \overbrace{\underbrace{v'_{n-4} + v'_{n-3}}_{Hilo\ n/2-2}}^{v''_{n/2-1}} \quad \overbrace{\underbrace{v'_{n-2} + v'_{n-1}}_{Hilo\ n/2-1}}^{v''_{n/2}}$$

Después de esta operación, tendríamos un nuevo vector con $n/2$ elementos, de la siguiente forma

$$\mathbf{v}'' = \boxed{v''_0} \boxed{v''_1} \boxed{v''_2} \boxed{v''_3} \boxed{\cdots} \boxed{v''_{n/2-4}} \boxed{v''_{n/2-3}} \boxed{v''_{n/2-2}} \boxed{v''_{n/2-1}}$$

Si se repiten los pasos sobre el nuevo vector $\mathbf{v}''$ se tendrá como resultado un vector de $n/4$ elementos, después aplicando sobre el nuevo vector se tendrá un vector resultante de $n/8$ y así hasta que tengamos como resultado un solo elemento que será la suma de todos los elementos del vector original.

## 3.7 Generador de plantillas de código en CUDA C para un número variable de GPUs

Al obtener resultados positivos de la evaluación de algoritmos en un GPU, se pensó en dividir el trabajo en dos GPUs y observar los resultados. El algoritmo implementado fue el de la sección anterior, la suma de los elementos de un vector, donde las simulaciones y resultados se muestran en el capítulo 4 de esta tesis.
Una vez realizado esto, se contempló la posibilidad de que los algoritmos previamente usados en nuestro grupo de trabajo, pudieran dividir el trabajo de esta misma forma, en dos GPUs. Algo tedioso que quita tiempo al programador es la preparación de la plantilla del código para poder ser ejecutada en dos GPUs, o porque detenerse ahí y extender la idea a tres, cuatro o $n$ número de GPUs, ya que hay funciones para los diferentes algoritmos que requieren muchas variables en el GPU, así como realizar la copia de variables previamente almacenadas en el CPU, el regreso de las mismas, la preparación del número de hilos, número de bloques que se utilizarán en cada GPU, mediciones de tiempo, creación de eventos para sincronizar los GPUs, entre otros. Debido al tiempo consumido por el programador para hacer esto, se propone una función en Matlab parametrizable con el número de GPUs que se vayan a usar y algunos otros argumentos para la misma, que se mencionarán a continuación:

- numOfGPU: Indica el número de GPUs que se vayan a utilizar
- numOfThreadsPerGPU:Es un vector que nos dirá si se quiere que los hilos que se manejen dentro del GPU deben tener dimensiones fijas o si su valor es 1.$^{en}$tonces se dividirán los hilos equitativamente entre los GPUs de acuerdo al siguiente valor del vector [-1,totalNumberOfThreads].
- numOfBlocksPerGPU: Es un vector que nos dirá si se quiere que los bloques que se manejen dentro del GPU deben tener dimensiones fijas o si su valor es 1.$^{en}$tonces se dividirán los bloques de acuerdo a cuantos sean requeridos en cada GPU si se dividió previamente de manera equitativa.

  numOfInputVariablesPerGPU: Es un vector del longitud = numOfGPU donde cada uno de los elementos indica el número de variables de entrada que se tendrá para cada uno de los GPUs.

nameOfInputVariablesPerGPU: Es un vector de longitud = numOfInputVariablesPerGPU(1) + ... + numOfInputVariablesPerGPU(numOfGPU) donde las primeras posiciones representarán los nombres de las variables de entrada que se quieran en el primer GPU y asi sucesivamente.

- typeOfInputVariablePerGPU: Es un vector de longitud = numOfInputVariablesPerGPU(1) + ... + numOfInputVariablesPerGPU(numOfGPU) donde las primeras posiciones representarán el tipo de las variables de entrada que se quieran en el primer GPU y asi sucesivamente.

- lengthOfInputVariablePerGPU: Es un vector que contiene las longitudes de las variables de entrada que se van a reservar para cada uno de las variables en los diferentes GPUs.

- nameOfInputVariablePerGPUFromHost: Es un vector que contiene el nombre de las variables del Host de las cuales se van a copiar valores hacía cada una de las variables en los diferentes GPUs.

- lengthOfInputVariablePerGPUFromHost: Es un vector que contiene las longitudes de las variables del Host que se van a copiar hacía cada uno de las variables en los diferentes GPUs, si el valor es 1"Se tomarán la longitud de la variable lengthOfInputVariablePerGPU, para cada una.

- numOfOutputVariablesPerGPUToHost: Es un vector del longitud = numOfGPU donde cada uno de los elementos indica el número de variables de salida que se tendrá para cada uno de los GPUs.

- nameOfOutputVariablePerGPUToHostInGPU: Es un vector de longitud = numOfOutputVariablesPerGPUToHost(1) + ... + numOfOutputVariablesPerGPUToHost(numOfGPU) donde las primeras posiciones representarán los nombres de las variables de salida que se quieran del primer GPU y asi sucesivamente.

- nameOfOutputVariablePerGPUToHostInHost: Es un vector de longitud = numOfOutputVariablesPerGPUToHost(1) + ... + numOfOutputVariablesPerGPUToHost(numOfGPU) donde las primeras posiciones representarán los nombres de las variables de salida que se quieran del primer GPU y asi sucesivamente.

lengthOfOutputVariablePerGPUToHost: Es un vector que contiene las longitudes de las variables del device que se van a copiar hacía cada uno de las variables en el host.

typeOfOutputVariablePerGPU: Es un vector de longitud = numOfOutputVariablesPerGPU(1) + ... + numOfOutputVariablesPerGPU(numOfGPU) donde las primeras posiciones representarán el tipo de las variables de salida que se quieran en el primer GPU y asi sucesivamente.

- nameOfTheFunction: Es el nombre de la función que se va a ejecutar en el GPU como kernel, también es el nombre que se le va a dar al archivo .cu de salida generado por esta función.

La forma de generar el esqueleto a partir de estos argumentos de la función es la siguiente:

1: Calculo o asignación de hilos y bloques para los Kernel
2: Espacio para que el usuario modifique las variables que se van a copiar o haga algún proceso en el CPU
3: Declaración y copia de variables del Host a los diferentes devices
4: Iniciar los streams y las variables para medir tiempos en cada uno de ellos
5: Lanzamiento de los Kerneles y captura de los eventos
6: Sincronización de los eventos y almacenamiento de tiempos de ejecución en variables y en archivo .txt
7: Regreso de las variables de cada uno de los GPUs al Host

Los códigos usadas para el generador de plantillas en MATLAB, vienen brevemente explicados así como con un link hacía el archivo que se agregó en el CD en la tabla 4.1

## 3.8. Resumen del capítulo

Durante este capítulo se explicó el desarrollo del trabajo realizado durante la tesis. Se trata el problema del manejo de los polinomios de Adomian, proponiendo

una estructura computacional para su almacenamiento que permite manejarlos como arreglos vectoriales para su uso en C++ y posteriormente en CUDA C/C++. Esta estructura permite almacenar vectores con diferente número de componentes o dimensiones para un arreglo de cuatro dimensiones. Se propone una nueva forma de sumar los elementos de un vector de forma paralela para reducir la complejidad del algoritmo. Como un objetivo alcanzado no especificado al inicio del trabajo se propone un generador de plantillas para el uso de $n$ GPUs con una función específica.

# Capítulo 4

# Simulaciones

## 4.1. MATLAB

Todos los códigos utilizados para ejecutar el algoritmo en MATLAB se encuentran incluidos en un disco aparte. Más adelante se explica brevemente cada una de los códigos, así como un link hacía el archivo en la tabla 4.1.
Se realiza el cálculo de los polinomios en MATLAB, usando el código *AdomPoly*, y se pasan a vectores utilizando el código *vecAdom2Vec*. Se procede a evaluar las polinomios a partir de la ecuación característica obtenida de la matriz, el código *eigenValuesAdom* es un ejemplo de como se hace. Este último ejemplo, se realiza utilizando los vectores de enteros obtenidos de convertir los polinomios simbólicos, si se quisiera evaluar de manera simbólica, se puede agregar al final de este ejemplo el código *EvaluacionSimbolica*
De acuerdo a lo discutido anteriormente, una de las formas de poder ejecutar este algoritmo en C++ es pasar los vectores de enteros generados a partir de los polinomios simbólicos, a un archivo en formato .txt para después ser leídos en otro lenguaje de programación, en este caso C++. El código *Matlab2txt* es utilizado para pasar el contenido de los vectores enteros a un archivo en formato .txt.

Se hizo una simulación usando el generador de plantillas *multiGPUFrame* para multiples GPUs, usando parametros que vienen comentados ahí mismo con la finalidad de que sea posiblre replicar la prueba y tener un ejemplo de como usarlo, usando

las funciones *getWord* y *equalThreadsPerGPU* dentro del generador de plantillas, la primera función obtiene una palabra de los vectores de entrada en forma de cadena y la segunda función calcula equitativamente el número de hilos para cada uno de los GPUs que se quieran en el archivo generado de salida *.cu*.

| Nombre del código | Descripción breve | Enlace (Dar Click) |
|---|---|---|
| AdomPoly | Código usada para generar los polinomios de Adomian en MATLAB a partir de variables simbólicas | `AdomPoly.m` |
| vecAdom2vecNum | Código para convertir los polinomios de Adomian que están calculados en variables simbólicas, a vectores de enteros | `vecAdom2vecNum.m` |
| eigenValuesAdom | Código para evaluar los eigenvalores de una matriz en MATLAB usando polinomios en forma de vectores enteros y no en variables simbólicas | `eigenValuesAdom.m` |
| EvaluacionSimbolica | Código para evaluar de manera simbólica los polinomios, sin usar la conversión a vectores de enteros | `EvaluacionSimbolica.m` |
| Matlab2txt | Código para pasar los vectores de enteros, obtenidos a partir de los polinomios simbólicos, a un archivo .txt | `Matlab2text.m` |
| multiGPUFrame | Código para generar una plantilla para una función en $n$ GPUs | `multiGPUFrame.m` |
| getWord | Código para obtener unas palabra de un vector de cadenas que se introduce como entrada en el generador de plantillas | `getWord.m` |
| equalThreadsPerGPU | Código para calcular de manera equitativa los hilos por GPU para el generador de plantillas | `equalThreadsPerGPU.m` |

**Tabla** 4.1: Tabla de códigos usados en MATLAB

## 4.2. C++

Todos los códigos utilizados para ejecutar el algoritmo en C++ se encuentran incluidos en un disco aparte. Más adelante se explica brevemente cada una de los códigos, así como un link hacía el archivo en la tabla 4.2.
Se utilizaron diferentes funciones para la ejecución del algoritmo, las cuales se pueden ver en el código *funciones*. Debido a que los elementos que componen cada uno de los polinomios de Adomian fueron exportados de MATLAB a un archivo en formato .txt, se tienen que cargar éstos previamente para la ejecutar el algoritmo, la carga se realiza mediante el código *txt2vectors*. Un ejemplo del cálculo de los eigenvalores de una matriz deseada se puede realizar mediante el código *eigenValuesAdomian*.

| Nombre del código | Descripción breve | Enlace ((Dar Click)) |
|---|---|---|
| funciones | Código que contiene las funciones utilizadas en el programa (archivo .cpp) | `funciones.c` |
| txt2vectors | Código para precargar en vectores en contenido del archivo .txt, que contienen los valores de los polinomios de Adomian calculados en MATLAB | `txt2vectors.c` |
| eigenValuesAdomian | Código para realizar una prueba al algoritmo introduciendo la matriz deseada | `eigenValuesAdomian.c` |

**Tabla** 4.2: Tabla de códigos usados en C++

## 4.3. CUDA C

Todos los códigos utilizados para ejecutar el algoritmo en CUDA C se encuentran incluidos en un disco aparte. Más adelante se explica brevemente cada una de los códigos, así como un link hacía el archivo en la tabla 4.3.
La implementación para el cálculo de eigenvalores en el GPU a partir de los polinomios de Adomian,se muestra en el código *AdomianGPU*. El cual está dado con un ejemplo de un matriz de $3 \times 3$ para asegurarse de que funcione y se ejecute de manera correcta sobre en el GPU. Se realizaron simulaciones de la suma de los elementos de un vector en paralelo, el código $SumaVectorParalelo1GPU$ contiene esta simulación para vectores de $2^n$ elementos. Se realizaron simulaciones para la misma suma de elementos de un vector en paralelo pero esta vez para ser ejecutado en dos GPUs, el código es el $SumaVector - Paralelo1GPU$.
Un ejemplo del código que se espera de salida, una vez ejecutada el generador de plantillas para multiples GPUs propuesto, está dado en el código *testing.cu*. Donde los valores utilizados vienen como ejemplo dentro del generador de plantillas en el archivo que se propuso en la plataforma de MATLAB.

| Nombre del código | Descripción breve | Enlace (Dar Click) |
|---|---|---|
| AdomianGPU | Código para calcular los eigenvalores de una matriz (por defecto, viene el cálculo de una matriz de $3 \times 3$) | `AdomianGPU.cu` |
| SumaVectorParalelo1GPU | Código para la suma de los elementos de un vector en paralelo en 1 GPU | `SumaVectorParalelo1GPU.cu` |
| SumaVectorParalelo2GPU | Código para la suma de los elementos de un vector en paralelo en 2 GPU | `SumaVectorParalelo2GPU.cu` |

**Tabla** 4.3: Tabla de códigos usados en CUDA

## 4.4. Resultados

Para el análisis de resultados, se realizan diversas pruebas del algoritmo en MATLAB,C++ y en CUDA C, las condiciones para las pruebas son que los polinomios ya están precalculados y precargados en la memoria, los resultados para las simulaciones en M son los siguientes

| Orden del polinomio | Matlab Simbólico | Matlab Vectorial | C++ Vectorial | CUDA Vectorial | CUDA (1 Hilo) | Matlab eig() |
|---|---|---|---|---|---|---|
| 2 | 22.082 | 16.430 | 14.78 | 0.00577888 | 26.963 | 0.0024182 |
| 3 | 26.874 | 19.915 | 15.34 | 0.0057936 | 46.365 | 0.0040907 |
| 4 | 31.171 | 27.811 | 19.30 | 0.00582976 | 62.951 | 0.0050878 |
| 5 | 39.407 | 32.765 | 23.80 | 0.00583904 | 81.574 | 0.0069387 |
| 6 | 48.220 | 40.773 | 28.90 | 0.00586304 | 97.699 | 0.0085118 |
| 8 | 81.690 | 68.392 | 47.30 | 0.00591744 | 112.784 | 0.0128101 |

**Tabla** 4.4: Tabla de resultados de los tiempos de ejecución (en milisegundos) para Matlab Simbólico, Matlab Vectorial, C++, CUDA C y CUDA C (1 Hilo), con 12 Polinomios evaluados

La tabla 4.4 es la comparativa que muestra el desempeño del algoritmo para cuando se ejecuta en un sólo procesador, es decir cuando toda la ejecución es secuencial y nada se ha hecho en paralelo. También incluye tiempos para la versión implementada en paralelo en la arquitectura de los GPUs de Nvidia. Esta prueba fue realizada calculando el valor de 12 polinomios para cada uno de los valores en los que se descompuso la sumatoria (2.7).
Como medida para una comparación con métodos competitivos, se tomaron mediciones para la función *eig*() de Matlab, las cuales también están en la tabla 4.4.
En la figura 4.1 se muestra el desempeño del algoritmo. Se puede observar que cuando el algoritmo se ejecuta en MATLAB usando variables simbólicas es más lento con respecto a cuando es usando vectores en MATLAB y C++. De lo cual se deduce que el cambio de la estructura computacional para el manejo de los polinomios mejoró el

desempeño de tiempo de nuestro algoritmo. Se puede observar que el tiempo medido para la versión paralela en CUDA comparado con el que le tomó a la función *eig*() de MATLAB, no tiene mucha diferencia, por ello en la figura 4.2 se observa una comparación entre ambas.
Se observa que el comportamiento de nuestro algoritmo en CUDA es practicamente una constante mientras que el de la función de MATLAB es un recta, ésto debido a la paralelización realizada.

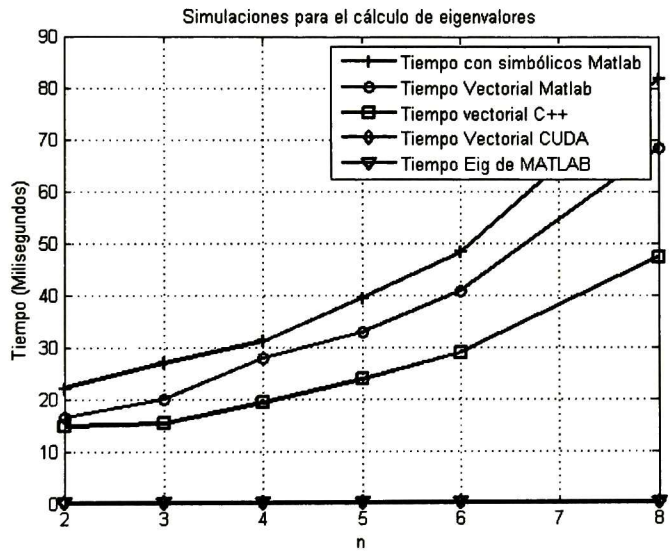



**Figura** 4.1: Gráfica de los tiempos de ejecución del cálcilo de eigenvalor vs el grado del polinomio de la tabla 4.4

El error comparado entre ambas, MATLAB y CUDA, se calculo mediante el error cuadrático medio, los errores obtenidos son los que están en la tabla 4.5 de la cual se obtiene la gráfica de la figura 4.3

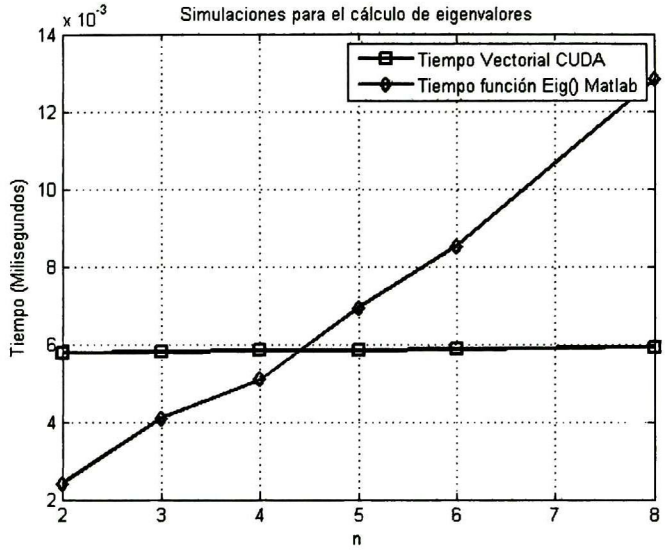



**Figura** 4.2: Gráfica de los tiempos de ejecución del cálcilo de eigenvalor vs el grado del polinomio de la tabla 4.4 para los casos: vectorial CUDA y operador *eig*() de MATLAB

| Orden del polinomio | Error |
|---|---|
| 2 | 8.41E-14 |
| 3 | 3.18E-13 |
| 4 | 9.86E-13 |
| 5 | 2.38E-12 |
| 6 | 7.72E-12 |
| 8 | 1.56E-11 |

**Tabla** 4.5: Error cuadrático medio obtenido

Se realizaron pruebas en la propuesta hecha para la suma de los elementos de

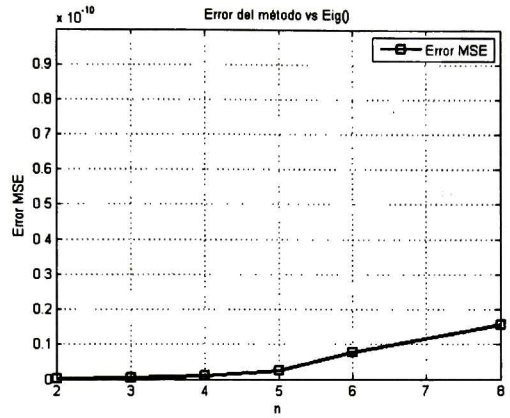



**Figura** 4.3: Gráfica del error cuadrático medio vs grado del polinomio

un vector en paralelo, se obtuvieron los resultados de la tabla 4.6 y las gráficas de la figuras 4.4,4.5,4.6,4.7

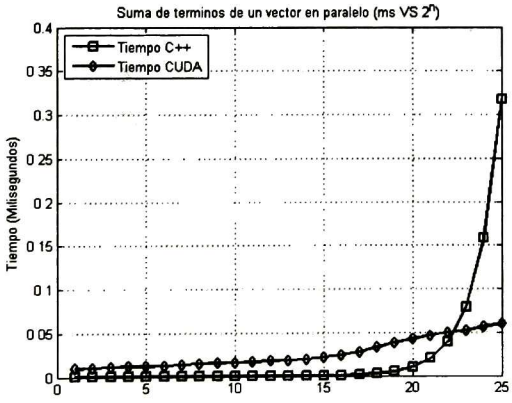



**Figura** 4.4: Gráfica de las pruebas de tiempo de la tabla 4.6

| Longitud del vector | C++ | C++ más Latencia | CUDA |
|---|---|---|---|
| $2^1$ | 0.000002979524322889 | 3.25032155088900e-06 | 0.0096752 |
| $2^2$ | 0.000003289465027724 | 4.37265393572400e-06 | 0.0103424 |
| $2^3$ | 0.000002997068136370 | 7.32982376837000e-06 | 0.0112189 |
| $2^4$ | 0.000003230985649453 | 2.05620081774530e-05 | 0.0118554 |
| $2^5$ | 0.000002029234425991 | 7.13533245379910e-05 | 0.0122762 |
| $2^6$ | 0.000002216368436457 | 0.000279512728948457 | 0.0127702 |
| $2^7$ | 0.000002959056540494 | 0.00111214449846049 | 0.0139606 |
| $2^8$ | 0.000004096480447858 | 0.00444083824812786 | 0.0148211 |
| $2^9$ | 0.000006339164604538 | 0.0177533062358365 | 0.0155088 |
| $2^{10}$ | 0.000011921021260470 | 0.0709997893051645 | 0.015928 |
| $2^{11}$ | 0.000021985322260848 | 0.283973458459925 | 0.0168048 |
| $2^{12}$ | 0.000043341991205286 | 1.13584923453777 | 0.0176605 |
| $2^{13}$ | 0.000084072878170788 | 4.54330764307260 | 0.0186864 |
| $2^{14}$ | 0.000166090206195364 | 18.1730603709675 | 0.0198019 |
| $2^{15}$ | 0.000329335390637861 | 72.6919064584360 | 0.0220358 |
| $2^{16}$ | 0.000654050910392341 | 290.766962543092 | 0.0242227 |
| $2^{17}$ | 0.001296692494084080 | 1163.06653066135 | 0.0279462 |
| $2^{18}$ | 0.002600768009674828 | 4652.26353664318 | 0.0329581 |
| $2^{19}$ | 0.005204641274335828 | 18609.0489481419 | 0.0384842 |
| $2^{20}$ | 0.010424063211529562 | 74436.1853980669 | 0.0424253 |
| $2^{21}$ | 0.0201318359103733 | 297744.720027851 | 0.0462544 |
| $2^{22}$ | 0.0397258282068848 | 1190978.83930988 | 0.0502746 |
| $2^{23}$ | 0.0793798788248803 | 4763915.27771610 | 0.0523898 |
| $2^{24}$ | 0.1588825614201600 | 19055660.9522274 | 0.0561805 |
| $2^{25}$ | 0.3181208324305579 | 76222643.4915004 | 0.0598983 |

**Tabla** 4.6: Tabla de resultados de los tiempos de ejecución (en milisegundos) para la suma de los elementos de un vector de forma paralela, en C++, C++ más latencia de la memoria y CUDA

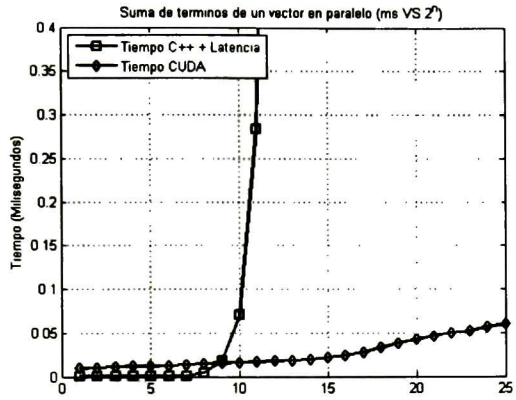



**Figura** 4.5: Gráfica de las pruebas de tiempo de la tabla 4.6

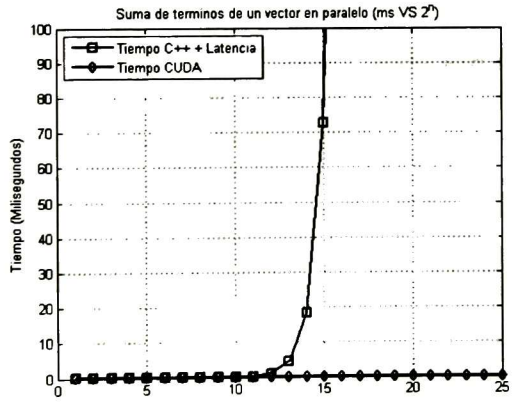



**Figura** 4.6: Gráfica de las pruebas de tiempo de la tabla 4.6

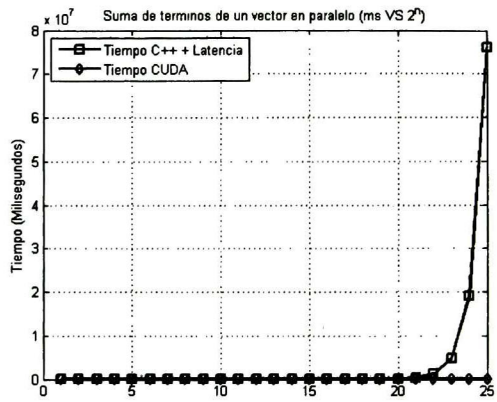



**Figura** 4.7: Gráfica de las pruebas de tiempo de la tabla 4.6

Los ejes de las gráficas, son el tiempo en milisegundos (eje Y) y el número de elementos que tiene un vector de longitud $2^n$.

En la gráfica de la figura 4.4 se observa la comparación entre el tiempo que tarda en realizarse la operación de la suma de los elementos de un vector en C++ y el tiempo que se tarda en CUDA. Se puede ver que C++ es más rápido que CUDA hasta que el número de elementos en el vector es del orden de $2^{23}$, pero hay que tener en cuenta que si los datos se tienen ya en el GPU, existe un costo de tiempo para regresarlos a la memoria del CPU. Entonces es necesario considerar que aunque sea más rápido en C++, cuando se toma en cuenta los tiempos de acceso a la memoria, ya sea para copiar o leer, se tendrá una comparación más justa ya que el tiempo que se gane en C++ podría resultar superado por la latencia. Tal es el caso de la gráfica de la figura 4.5 donde se toma en cuenta la latencia de la copia de los datos y se observa que ahora cuando el vector es de $2^{10}$ ya existe una diferencia de tiempo entre ambos, ya que CUDA sigue lineal y el tiempo de C++ más la latencia empieza a crecer conforme van creciendo el número de datos. En la gráfica 4.6 se observa la diferencia para un número más grande de longitud del vector y en la gráfica 4.7 se muestra la comparativa de la curva completa. Nótese que las escalas de tiempo cambian en cada gráfica, solo la primera y la segunda son iguales.

## 4.5. Resumen del capítulo

En este capítulo se presentan los resultados obtenidos durante el trabajo de tesis, explicando los códigos en diferentes lenguajes de programación usados durante el mismo. Así como la comparativa del algoritmo propuesto contra las diferentes versiones propuestas y con una función de MATLAB para comparar el desempeño de ambas. También se habla acerca de los resultados de la suma de los elementos de un vector de forma paralela, tomando en cuenta el tiempo de acceso a memoria del GPU y la cantidad de elementos que se quieren copiar, de esta forma analizar cuando es conveniente hacer uso de lo propuesto y cuando es mejor ejecutarlo de manera secuencial.

# Capítulo 5

# Conclusiones y trabajo futuro

## 5.1. Conclusiones

- Partiendo de una propuesta de implementación de los polinomios de Adomian en MATLAB simbólica, se implementaron las siguientes versiones no paralelas: MATLAB vectorial y C++ Vectorial.

- Se desarrolló una estructura computacional para el manejo de los polinomios de Adomian representados en forma de multiples vectores de diferentes dimensiones.

- Se paralelizó el algoritmo de los polinomios de Adomian empleando la forma vectorial (con vectores de diferentes dimensiones).

  Aplicaciones: Cálculo de los eigenvalores, no linealidades de la forma $f(x) = x^n$.

  Se propuso la suma de los elementos de un vector de forma paralela en una GPU, disminuyendo su complejidad de $n$ a $\log_2(n)$. Teniendo como propósito ser usado cuando los vectores a procesar estén ya en la memoria del GPU, ahorrando tiempo en la copia hacía la memoria del CPU.

  Se propone una reescritura de los polinomios de Adomian, almacenando las multiplicaciones entre los $u_i$ de la ecuación (2.7), teniendo como finalidad redu-

cir la cantidad de multiplicaciones con los mismos términos y por ende reducir la complejidad del algoritmo.

Se implementó un generador de plantillas para cuando se quiere usar una función en más de un GPU, ahorrando al programador el tiempo que tarda en preparar los GPUs y reduciendo posibles errores en este proceso.

La eficiencia alcanzada con la propuesta, se midió tomando tiempos de la ejecución del algoritmo. La versión que tuvo el mejor desempeño fue la paralela implementada en CUDA, que fue la comparada con la función *eig*() de MATLAB, presentado en la gráfica 4.2. La peor versión fue la implementación con variables simbólicas en MATLAB.

## 5.2. Trabajo futuro

- Se propone implementar la modificación propuesta para el algoritmo, que involucra el precálculo de algunos de los valores que habrá en los términos de los polinomios y su almacenamiento. Y así ver en qué medida la disminución de número de multiplicaciones ayuda a disminuir el tiempo de ejecución y si el cambio es significativo como para que se justifique el uso de memoria que se está usando para el almacenamiento de dichos valores.

- Este trabajo, fue el primer paso para el algoritmo, ya que sólo se uso para encontrar las raíces de un polinomio de orden $n$. Pero recordar que este algoritmo es capaz de evaluar ecuaciones matemáticas no-lineales sin aplicar ningún tipo de linealización o suponerlo lineal, por lo tanto es posible usarse con ecuaciones diferenciales, ecuaciones exponenciales, ecuaciones logarítmicas, etc. Por tanto, con esta forma de almacenar los polinomios en vectores, se piensa que se puede tener mejores resultados en cuanto a tiempo sin sacrificar la precisión. Se propone que como trabajo futuro este algoritmo se aplique a otras no-linealidades.

- Hacer más robusto el generador de plantillas para que sea capaz de generar

partes de las funciones a utilizar y así reducir más tiempo al programador en dicho proceso.

Implementar la suma del vector de forma paralela para alguna aplicación en específico y ver si se logra alguna mejora en el desempeño.

# Bibliografía


[1] George Adomian. A review of the decomposition method in applied mathematics. *Journal of mathematical analysis and applications*, 135(2):501–544, 1988.

[2] Grégoire Allaire, Sidi Mahmoud Kaber, and Karim Trabelsi. *Numerical linear algebra*, volume 55. Springer, 2008.

[3] ESMAEIL Babolian and J Biazar. On the order of convergence of adomian method. *Applied Mathematics and Computation*, 130(2):383–387, 2002.

[4] Utpal Banerjee. *Loop transformations for restructuring compilers: the foundations*, volume 1. Springer Science & Business Media, 1993.

[5] J Biazar, M Tango, E Babolian, and R Islam. Solution of the kinetic modeling of lactic acid fermentation using adomian decomposition method. *Applied mathematics and computation*, 144(2):433–439, 2003.

[6] Y Cherruault and G Adomian. Decomposition methods: a new proof of convergence. *Mathematical and Computer Modelling*, 18(12):103–106, 1993.

[7] Yves Cherruault. Convergence of adomian's method. *Kybernetes*, 18(2):31–38, 1989.

[8] Thomas H Cormen. *Introduction to algorithms*. MIT press, 2009.

[9] H Fatoorehchi and H Abolghasemi. Adomian decomposition method to study mass transfer from a horizontal flat plate subject to laminar fluid flow. *Advances in Natural and Applied Sciences*, 5(1):26–33, 2011.

[10] Hooman Fatoorehchi and Hossein Abolghasemi. On calculation of adomian polynomials by matlab. *Journal of Applied Computer Science & Mathematics*, 5(11), 2011.

[11] Hooman Fatoorehchi and Hossein Abolghasemi. Improving the differential transform method: a novel technique to obtain the differential transforms of nonlinearities by the adomian polynomials. *Applied Mathematical Modelling*, 37(8):6008–6017, 2013.

[12] Hooman Fatoorehchi and Hossein Abolghasemi. A more realistic approach toward the differential equation governing the glass transition phenomenon. *Intermetallics*, 32:35–38, 2013.

[13] Hooman Fatoorehchi and Hossein Abolghasemi. On computation of real eigenvalues of matrices via the adomian decomposition. *Journal of the Egyptian Mathematical Society*, 22(1):6–10, 2014.

[14] John B Fraleigh. *A first course in abstract algebra.* Pearson Education India, 2003.

[15] Gilbert Helmberg, Peter Wagner, and Gerhard Veltkamp. On faddeev-leverrier's method for the computation of the characteristic polynomial of a matrix and of eigenvectors. *Linear algebra and its applications*, 185:219–233, 1993.

[16] Alston S Householder. *The theory of matrices in numerical analysis.* Courier Corporation, 2013.

[17] Yong-Chang Jiao, Chuangyin Dang, and Yoshitsugu Yamamoto. An extension of the decomposition method for solving nonlinear equations and its convergence. *Computers & Mathematics with Applications*, 55(4):760–775, 2008.

[18] Balaram Kundu and Somchai Wongwises. A decomposition analysis on convecting–radiating rectangular plate fins for variable thermal conductivity and heat transfer coefficient. *Journal of the Franklin Institute*, 349(3):966–984, 2012.

[19] Randolph Rach. A convenient computational form for the adomian polynomials. *Journal of mathematical analysis and applications*, 102(2):415–419, 1984.

[20] Abdul-Majid Wazwaz. A new algorithm for calculating adomian polynomials for nonlinear operators. *Applied Mathematics and Computation*, 111(1):33–51, 2000.

[21] Abdul-Majid Wazwaz and Alice Gorguis. An analytic study of fisher's equation by using adomian decomposition method. *Applied Mathematics and Computation*, 154(3):609–620, 2004.

[22] EAA Ziada. Adomian solution of a nonlinear quadratic integral equation. *Journal of the Egyptian Mathematical Society*, 21(1):52–56, 2013.

# CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS DEL I.P.N.
# UNIDAD GUADALAJARA

El Jurado designado por la Unidad Guadalajara del Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional aprobó la tesis

Paralelización del Algoritmo del Método de Descomposición de Adomian y su Implementación en una GPU

del (la) C.

Gerardo RAMÍREZ ARREDONDO

el día 20 de Agosto de 2015.

Dr. Deni Librado Torres Román
Investigador CINVESTAV 3C
CINVESTAV Unidad Guadalajara

Dr. Yuriy Shkvarko
Investigador CINVESTAV 3C
CINVESTAV Unidad Guadalajara

Dr. Ramón Parra Michel
Investigador CINVESTAV 3C
CINVESTAV Unidad Guadalajara