

UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
VALPARAÍSO - CHILE



“IMPLEMENTACIÓN DE DIFERENCIACIÓN AUTOMÁTICA EN SOLVER NO LINEALES”

VALERIE ALEXANDRA EUGENIA JIMÉNEZ MEZA

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN INFORMÁTICA

Profesor Guía: Claudio Torres, Ph.D
Profesor Correferente: Roberto León, Ph.D
Profesora Correferente: Raquel Pezoa, Ph.D

Diciembre - 2023

DEDICATORIA

Dedicado a todos aquellos creen que no van a lograrlo, porque yo alguna vez estuve en el mismo lugar.

AGRADECIMIENTOS

Al llegar al final de esta etapa, deseo expresar mi más sincero agradecimiento a todas las personas que desempeñaron un papel significativo en este proceso.

En primer lugar, quiero agradecer a mi profesor guía, el Profesor Claudio Torres, por su constante orientación, inquebrantable apoyo y valiosas sugerencias a lo largo de esta investigación. Su vasta experiencia y dedicación fueron fundamentales para dar forma a este proyecto.

También quiero agradecer a mis amigos, algunos de los cuales conozco desde toda la vida y otros que conocí en la universidad. Han estado a mi lado sin importar cuántas veces tuve que declinar una invitación para reunirnos debido a mis responsabilidades académicas. Agradezco profundamente a mi familia por brindarme la oportunidad de estudiar en la universidad sin tener que preocuparme por nada más que mis estudios.

A mi mor, quien ha experimentado de primera mano mi nivel de concentración y a veces ha sido ignorado sin intención cuando estaba enfocada en obtener resultados.

Además, quiero expresar mi más sincero agradecimiento a la profesora Carolina Arbelaez, quien, sin saberlo, me brindó el apoyo académico necesario para que continuara en este camino.

Finalmente, quiero agradecerme a mí misma. A lo largo de este trayecto, me perdí incontables veces, pero siempre pude reencontrar mi rumbo gracias a mi capacidad de escuchar y aprender de todos ustedes.

Gracias a cada uno de ustedes por ser parte esencial de este viaje en mi vida.

RESUMEN

Resumen— Uno de los desafíos más significativos radica en la resolución de sistemas de ecuaciones no lineales sobredeterminados, principalmente debido a los altos requisitos computacionales y de memoria asociados a la obtención de la matriz Jacobiana. En este trabajo, se propone la implementación de la diferenciación automática a través de las funciones de *Jacobian-Vector Product* (jvp) y *Vector-Jacobian Product* (vjp). Además, se plantea una estrategia para obtener una aproximación de la matriz Jacobiana $\tilde{J}$ con la particularidad de ser una versión dispersa, lo que resulta fundamental para el manejo eficiente del producto $\tilde{J}^T \mathbf{w}$. Todo ello con el fin de implementar una versión alternativa del método nsLSQR que pueda obtener resultados satisfactorios sin requerir recursos computacionales excesivos

Palabras Clave— Sistemas de ecuaciones no lineales, Matriz Jacobiana, Métodos de resolución, diferenciación automática

ABSTRACT

Abstract— One of the most significant challenges lies in solving overdetermined nonlinear systems of equations, primarily due to the high computational and memory requirements associated with obtaining the Jacobian matrix. In this work, the implementation of automatic differentiation through the *Jacobian-Vector Product* (jvp) and *Vector-Jacobian Product* (vjp) functions is proposed. Additionally, a strategy is introduced to obtain an approximation of the Jacobian matrix $\tilde{J}$ with the particularity of being a sparse version, which is crucial for the efficient handling of the $\tilde{J}^T \mathbf{w}$ product. All of this is aimed at developing an alternative version of the nsLSQR method that can achieve satisfactory results without demanding excessive computational resources.

Keywords— Nonlinear systems of equations, Jacobian matrix, Solution methods, algorithmic differentiation

GLOSARIO

CPU: Central Processing Unit

GPU: Graphical Processing Unit

JAX: Just-Another-XLA.

JIT: Just-in-Time.

JVP: Jacobian-Vector-Product.

nsLSQR: non-Symmetric LSQR.

TPU:Tensor Processing Unit

VJP: Vector-Jacobian-Product.

XLA: Accelerated Linear Algebra.

ÍNDICE DE CONTENIDOS

RESUMEN	IV
ABSTRACT	IV
GLOSARIO	V
ÍNDICE DE FIGURAS	VIII
ÍNDICE DE TABLAS	VIII
INTRODUCCIÓN	1
CAPÍTULO 1: DEFINICIÓN DEL PROBLEMA	3
CAPÍTULO 2: MARCO CONCEPTUAL	5
2.1 Métodos de resolución de sistemas de ecuaciones no lineales	5
2.2 LSQR	10
2.3 Diferenciación automática	12
2.3.1 Forward mode	14
2.3.2 Reverse mode	16
CAPÍTULO 3: PROPUESTA DE SOLUCIÓN	20
3.1 Algoritmo nsLSQR	21
3.1.1 nsLSQR + JVP/VJP	25
3.2 Aproximación de J	28
CAPÍTULO 4: VALIDACIÓN DE LA SOLUCIÓN	30
4.1 Memoria y tiempo VJP	31
4.2 Tiempos de ejecución en iteraciones internas nsLSQR	33
4.3 Desempeño nsLSQR	36
CAPÍTULO 5: CONCLUSIONES	40
ANEXOS	42
A Pseudocódigos adicionales	42
A.1 Decomposición QR	42
A.2 Solución problema mínimos cuadrados	43
B Reporte uso de memoria memoria función VJP	43
B.1 2%	43
B.2 10%	44

B.3	25 %	45
B.4	30 %	46
B.5	50 %	47
B.6	100 %	48
C	Reporte uso de memoria nsLSQR con Diferenciación Automática	49
	REFERENCIAS BIBLIOGRÁFICAS	52

ÍNDICE DE FIGURAS

1	Tiempo y memoria que utiliza $\tilde{J}$ y el calculo asociado a ella (VJP). La abscisa indica el porcentaje de la matriz que se utilizó para obtener la aproximación. . . .	32
2	Variación del tiempo que demora en realizar el paso interno del algoritmo	34
3	Variación del tiempo que demora en realizar el paso interno del algoritmo	35
4	Residuo relativo y error relativo de la solución en función de las iteraciones internas	37
5	Residuo relativo dependiendo del valor de e	38

ÍNDICE DE TABLAS

1	Three-part F'	13
2	Grafo computacional F'	14
3	forward mode iteración 1	15
4	forward mode iteración 2	15
5	Reverse mode iteración 1	19
6	Memoria VJP	33
7	Caption	39

INTRODUCCIÓN

En la búsqueda constante de soluciones efectivas para una amplia gama de problemas en campos que van desde la física y la ingeniería hasta la ciencia de datos y la optimización, los métodos numéricos desempeñan un papel crucial. Estos métodos son la columna vertebral de muchas aplicaciones computacionales y científicas, proporcionando la capacidad de abordar problemas complejos que a menudo carecen de soluciones analíticas. Sin embargo, uno de los desafíos persistentes en la aplicación de estos métodos radica en la obtención de derivadas precisas y eficientes, que son fundamentales para el cálculo de gradientes, optimización y análisis de sensibilidad.

Uno de los tópicos más desafiantes bajo lo anterior es el de encontrar soluciones a sistemas de ecuaciones no lineales sobredeterminados, los cuales presentan dos adversidades: la no linealidad que a veces implica tener más de una solución y tener más ecuaciones que incógnitas. Esto provoca que el problema se transforme en encontrar la “mejor solución” que minimice la discrepancia entre las ecuaciones dadas y las soluciones propuestas. Para enfrentarse a estos problemas, los métodos que se implementan requieren de una u otra forma trabajar con la matriz Jacobiana J , cuyo requerimiento en memoria aumenta a medida que crecen las dimensiones del problema.

Este trabajo se centra en la modificación del algoritmo nsLSQR [Sanhueza, 2021], método utilizado para resolver los problemas antes mencionados. Se explorará una alternativa a las diferencias finitas: el método de diferenciación automática, la cual es una técnica que combina conceptos de cálculo numérico y simbólico. Esta permite obtener derivadas precisas, incluso cuando son no lineales con un grado alto y multidimensionales. Además, se presentará nueva forma de obtener una aproximación menos costosa de la matriz Jacobiana J con la cual se calculará el producto $\tilde{J}^T \mathbf{w}$, punto focal de estudio. Con ello se analizará como esta perturbación en la matriz J afecta al resultado de $\tilde{J}^T \mathbf{w}$ y a la convergencia del algoritmo. Lo anterior se puede entender bajo el siguiente *trade off*: utilizamos una aproximación $\tilde{J}$ que es peor que J , pero con ello obtenemos la reducción de los tiempos de computación y la memoria utilizada.

La estructura de este documento se divide en cinco secciones principales. La primera sección aborda la definición del problema, proporcionando un contexto integral sobre el tema en cuestión. A continuación, en la sección del marco conceptual, se presenta un análisis del estado actual de la investigación y se explica la teoría fundamental que servirá de base para comprender la solución propuesta. Luego, se detalla la propuesta de solución en la sección correspondiente, ofreciendo una visión completa y detallada de la solución planteada. Finalmente, se presenta una conclusión que resume los puntos clave del tema abordado.

CAPÍTULO 1

DEFINICIÓN DEL PROBLEMA

En diversos campos de investigación, nos enfrentamos a problemas no lineales. Estos desafíos van desde la simulación de eventos naturales o físicos que involucran una gran cantidad de términos y grados, hasta la aplicación de la inteligencia artificial para predecir eventos futuros o clasificar información. La resolución de problemas complejos es una constante en estos escenarios. Por lo general, estos problemas requieren una manera eficiente de manejar, estudiar y visualizar modelos que carecen de soluciones analíticas bajo condiciones específicas, lo que haría extremadamente complicado obtenerlas.

Podemos pensar en estos problemas no lineales de la siguiente forma:

$$\mathbf{F}(\mathbf{x}) = \begin{bmatrix} f_1(\mathbf{x}) & f_2(\mathbf{x}) & f_3(\mathbf{x}) & \cdots & f_m(\mathbf{x}) \end{bmatrix} \quad (1)$$

donde existen m ecuaciones que son evaluadas en un vector $\mathbf{x}$ con n componentes y al menos una de las funciones f_m es no lineal.

Pero, cuando buscamos una solución numérica para estos problemas, a menudo reformulamos estos problemas como sistemas de ecuaciones no lineales. Los cuales pueden ser definidos la siguiente forma:

$$\mathbf{F}(\mathbf{x}) = \mathbf{0}$$

donde $\mathbf{F} : \mathbb{R}^n \rightarrow \mathbb{R}^m$ es una función, $\mathbf{x} \in \mathbb{R}^n$ representa una raíz de la función $\mathbf{F}$, y $\mathbf{0} \in \mathbb{R}^m$ es un vector de ceros. En caso de que no exista una raíz, podemos construir este sistema utilizando una forma alternativa:

$$\mathbf{F}(\mathbf{r}) - \mathbf{F}(\mathbf{x}) = \mathbf{0}$$

donde ahora el vector $\mathbf{r}$ representa nuestra raíz.

Estos sistemas se clasifican según la relación entre la dimensión del dominio y el rango de la función $\mathbf{F}$:

1. Si $m < n$, el problema se denomina “sistema de ecuaciones no lineales subdeterminado”.
2. Si $m > n$, el problema se denomina “sistema de ecuaciones no lineales sobredeterminado”.
3. Si $m = n$, el problema se denomina “sistema de ecuaciones no lineales cuadrado”.

A lo largo de los años, se han estudiado diversas formas de obtener soluciones para sistemas de ecuaciones no lineales, muchas de las cuales requieren el cálculo de la matriz Jacobiana $J_{\mathbf{F}}$. Esta es esencial para el cálculo de la aproximación de una solución para el sistema, expresado frecuentemente como:

$$\begin{aligned}\mathbf{F}(\mathbf{x}_i + \Delta\mathbf{x}_i) &= \mathbf{F}(\mathbf{x}_i) + J(\mathbf{x}_i)\Delta\mathbf{x}_i + \dots \\ J(\mathbf{x}_i)\Delta\mathbf{x}_i &= -\mathbf{F}(\mathbf{x}_i)\end{aligned}$$

notando que $J(\mathbf{x}_i)$ es utilizado principalmente para obtener el producto entre este y un vector, el cual puede ser fácilmente obtenido, por ejemplo, a través de diferencias finitas, en donde se obtiene una expresión directa para obtenerlo.

El desafío radica en que en los métodos utilizados para resolver problemas sobredeterminados, se requiere además el cálculo del producto $J(\mathbf{x}_i)^T \mathbf{w}$, el cual, a diferencia del anterior, no tiene una forma explícita. Para problemas de pequeña escala, este producto podría obtenerse mediante el cálculo explícito de la matriz y la realización del producto, pero en problemas de mayor dimensión, esto se vuelve inviable debido al aumento en la demanda de memoria. Por esta razón, la implementación de este producto sigue siendo un desafío pendiente.

El objetivo principal de esta memoria es abordar la resolución de sistemas de ecuaciones sobredeterminados a través del método nsLSQR [Sanhueza, 2021], proporcionando por un lado, una nueva forma de obtener los productos $J(\mathbf{x}_i) \mathbf{v}$ y $\tilde{J}(\mathbf{x}_i)^T \mathbf{w}$ a través de la diferenciación automática; por el otro, una forma alternativa de obtener una aproximación de la matriz Jacobiana $\tilde{J}$. Con la combinación de ambos se espera lograr equilibrar la balanza entre el uso de memoria y la cantidad de operaciones elementales utilizadas.

CAPÍTULO 2

MARCO CONCEPTUAL

2.1. Métodos de resolución de sistemas de ecuaciones no lineales

La resolución de sistemas no lineales es un desafío que ha suscitado un gran interés, y para abordarlo existen varios enfoques. Es importante destacar que resolver estos sistemas no lineales es una tarea sumamente complicada en muchos casos, y la forma más común de abordarlos es a través de métodos iterativos. Estos métodos ofrecen una solución práctica dada la complejidad inherente de estos problemas, y permiten avanzar hacia soluciones aproximadas de manera efectiva.

El Método de Newton [Sauer, 2011, Chong y Zak, 2004], es la opción más destacada para abordar estos problemas.

Supongamos que tenemos una función $F : \mathbb{R}^n \rightarrow \mathbb{R}^n$ para la cual deseamos encontrar un vector $\mathbf{x} \in \mathbb{R}^n$ que cumpla con la condición $F(\mathbf{x}) = \mathbf{0}$. Si la matriz Jacobiana $J(\mathbf{x})$ es no singular entorno a una vecindad de la raíz $\mathbf{x}$, entonces es posible aplicar el Método de Newton. Este método consiste en construir una solución a través de una iteración definida de la siguiente manera:

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \Delta\mathbf{x}_i \quad (2)$$

donde $\Delta\mathbf{x}_i$ es llamado *Newton step* y representa la solución de la ecuación de Newton

$$J(\mathbf{x}_i)\Delta\mathbf{x}_i = -F(\mathbf{x}_i)$$

Este método presenta una característica esencial: cuando el punto de inicio se encuentra lo suficientemente próximo a la raíz del problema, exhibe una convergencia cuadrática, aunque esta solo es válida a nivel local.

Para abordar situaciones en las que se comienza desde una solución inicial distante $\mathbf{x}_0$, es necesario emplear técnicas que permitan al método lograr una convergencia global. Entre estas, destaca en popularidad y simplicidad la técnica *backtracking line search* [Nocedal y Wright, 2006].

Sin embargo, es importante destacar que este método implica el cálculo y almacenamiento de la matriz Jacobiana en cada iteración. Aunque esto puede no suponer un problema significativo en casos de problemas pequeños, a medida que la dimensión del problema aumenta, más memoria es requerida por el método.

Los métodos *line search* parten desde una iteración similar a la iteración de Newton presentada en la ecuación (2), de modo que se expresa de la siguiente manera:

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \alpha_i \mathbf{p}_i \quad (3)$$

para lo cual $\alpha_i > 0$ es un escalar llamado *step lenght* y $\mathbf{p}_i$ un vector de dirección. La idea es buscar los mejores valores para α_i y $\mathbf{p}_i$.

En particular, se presta especial atención al valor de α_i , que, aunque tiene un cálculo que puede obtener el óptimo, su alto costo a menudo lleva a explorar otras formas de determinarlo, que resultan igualmente efectivas, siendo una de estas la técnica *backtracking*. Es importante destacar que este α_i es crucial para la convergencia del método. Un valor muy pequeño resulta en una convergencia lenta, mientras que un valor muy grande puede no garantizar la convergencia. Una propiedad deseada es que para un α_i lo suficientemente pequeño no negativo [Armijo, 1966]:

$$f(\mathbf{x}_i + \alpha_i \mathbf{p}_i) \leq f(\mathbf{x}_i) + \alpha \rho \mathbf{p}_i^T f'(\mathbf{x}_i)$$

donde $\rho \in (0, 1)$. Esta inecuación nos indica que los valores α aceptados son todos los que permiten que $f(\mathbf{x}_i + \alpha_i \mathbf{p}_i)$ se encuentre bajo $f(\mathbf{x}_i) + \alpha \rho \mathbf{p}_i^T f'(\mathbf{x}_i)$. Esta condición garantiza que la longitud del paso calculada pueda disminuir suficientemente la función objetivo, pero no que converja en un número razonable de iteraciones.

Los métodos *Nonlinear Conjugate Gradient* [Nocedal y Wright, 2006, Cheng et al., 2009, Hager y Zhang, 2006] forman parte de los métodos *line search* y extienden el método del gradiente. Comienzan desde la expresión (3), en este caso α_i se obtiene a través de *lineal*

search y la dirección $\mathbf{p}_i$ se calcula utilizando la siguiente expresión:

$$\mathbf{p}_{i+1} = -g(\mathbf{x}_{i+1}) + \beta_i \mathbf{p}_i$$

donde g representa $\nabla F(\mathbf{x}_i)^T$ y β_i un escalar que asegura que $\mathbf{p}_{i+1}$ y $\mathbf{p}_i$ son conjugados, además de ser el parámetro a actualizar. La diferencia entre cada método corresponde a una distinta forma de elegir el escalar β_i . La selección de la expresión para obtener este valor es el desafío de este enfoque.

El método *Spectral Gradient* [Raydan, 1997, Birgin et al., 2014, Zhang y Zhou, 2006] es un enfoque que parte desde la iteración definida como:

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \alpha_i \mathbf{g}_i$$

el valor de α_i se obtiene a través de una expresión fija, la iteración *Barzilai-Borwein* [Barzilai y Borwein, 1988].

$$\alpha_i = \frac{(\mathbf{x}_{i+1} - \mathbf{x}_i)^T (\mathbf{x}_{i+1} - \mathbf{x}_i)}{(\mathbf{x}_{i+1} - \mathbf{x}_i)^T h(\mathbf{x}_{i+1}) - h(\mathbf{x}_i)}$$

donde h es el gradiente de la función F evaluado en el punto $\mathbf{x}_i$.

Una de las grandes ventajas que presenta *Spectral Gradient* es que el cálculo de α_i tiene un bajo costo computacional y requiere poca memoria. Con respecto a la convergencia, se ha demostrado que alcanza convergencia global para el caso cuadrático estrictamente convexo con cualquier número de variables [Raydan, 1993].

El método *quasi-Newton* [Birgin et al., 2003, Dennis y Moré, 1977] que se basa en la formulación de Newton, se distingue por obtener la dirección de búsqueda mediante la aproximación siguiente:

$$\mathbf{p}_i = -B_i^{-1} \nabla F(\mathbf{x}_i)$$

En esta oportunidad buscamos aproximar la matriz Hessiana a través de una matriz B_i definida positiva, la cual es actualizada en cada iteración usando información obtenida en pasos anteriores. Para ello, es imperativo que esta matriz cumpla con la ecuación de la secante:

$$B_{i+1}(\mathbf{x}_{i+1} - \mathbf{x}_i) = \nabla F(\mathbf{x}_{i+1}) - \nabla F(\mathbf{x}_i)$$

El más popular es el BFGS (Broyden, Fletcher, Goldfarb, and Shanno), el cual resuelve un problema de minimización sujeto a restricciones

$$\begin{aligned} & \underset{B_{i+1}}{\text{minimize}} && ||B_{i+1} - B_i|| \\ & \text{subject to} && B_{i+1}^T = B_{i+1}, \\ & && B_{i+1} \Delta \mathbf{x}_i = \nabla F(\mathbf{x}_{i+1}) - \nabla F(\mathbf{x}_i) \end{aligned}$$

Al resolver el problema de optimización, se obtienen una solución única para B_{i+1} .

La convergencia global de este método es incierta, principalmente debido a que la dirección obtenida mediante la aproximación mencionada previamente no siempre corresponde a una dirección de descenso con respecto a la norma del residuo [Birgin *et al.*, 2003].

El método de Newton inexacto [Dembo *et al.*, 1982, Nocedal y Wright, 2006] utiliza una aproximación distinta para $\mathbf{p}_i$, la cual debe satisfacer

$$||F(\mathbf{x}_i) + J_i \mathbf{p}_i|| \leq \eta_i ||F(\mathbf{x}_i)||$$

con $\eta_i > 0$ denominado como *forcing sequence* y se utiliza para controlar la precisión.

Una importante modificación que realizado al método de Newton es el enfoque conocido como *Jacobian-free Newton-Krylov* (JFNK) [Knoll y Keyes, 2004]. Este método es parte de los métodos del subespacio de Krylov. Se le denomina *Jacobian-free* porque no requiere almacenar o calcular matriz Jacobiana, sino que necesita una rutina para calcular el producto entre la matriz Jacobiana y un vector $J \mathbf{v}$. Una de las ventajas clave de este enfoque es que converge sin la necesidad de calcular la verdadera matriz Jacobiana. Además, obtener el producto matriz-vector no impone un alto costo computacional, ya que puede aproximarse utilizando un enfoque de diferencias finitas. Por ejemplo, una aproximación de segundo orden del producto $J(\mathbf{x}_i) \mathbf{v}$ se calcula mediante la siguiente expresión:

$$J(\mathbf{x}_i) \mathbf{v} \approx \frac{F(\mathbf{x}_i + \epsilon \mathbf{v}) - F(\mathbf{x}_i - \epsilon \mathbf{v})}{2\epsilon} \quad (4)$$

cuyo error es proporcional a ϵ . Es posible utilizar aproximaciones de orden superior, pero esto aumenta el costo computacional asociado con estas aproximaciones.

La ventaja sobre el método de Newton es que evita el almacenamiento de la matriz en memoria, reemplazándolo por el cálculo del producto en cada iteración.

Lamentablemente, la mayoría de los métodos mencionados han sido estudiados para el caso cuadrado $n = m$.

Una modificación a JNKF [Sanhueza y Torres, 2017] expande la idea de este método y permite su implementación en sistemas de ecuaciones no lineales sobredeterminados. Para ello el algoritmo utiliza el enfoque de diferencias finitas (4) para obtener una aproximación del producto $J\mathbf{v}$ y se otro que obtenga $J^T\mathbf{w}$. Se generan dos sugerencias para esta última. El primero utiliza una base ortonormal, donde $Q = \{\mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_i\}$, $\mathbf{q}_i \in \mathbb{R}^n$ es una base ortonormal de n , $J^T \in \mathbb{R}^{n \times m}$ y $\mathbf{w} \in \mathbb{R}^m$ donde la aproximación se calcula de la siguiente manera:

$$J^T(\mathbf{x})\mathbf{w} \approx \sum_{i=1}^k \langle J(\mathbf{x})\mathbf{q}_i, \mathbf{w} \rangle$$

La segunda forma propuesta es a través del uso del gradiente

$$\nabla g = J^T(\mathbf{x})\mathbf{v}$$

La diferencia principal es que en la primera expresión, si se utilizan n vectores de Q se puede obtener el resultado exacto del producto, pero esto implicaría utilizar más memoria para almacenar estos vectores.

2.2. LSQR

Se conoce como **LSQR** [Paige y Saunders, 1982] a un algoritmo iterativo para resolver sistemas de ecuaciones de la forma $A\mathbf{x} = \mathbf{b}$ asociados a un problema de mínimos cuadrados

$$\min_{\mathbf{x} \in \mathbb{R}^n} \|\mathbf{b} - A\mathbf{x}\|^2$$

donde la matriz $A \in \mathbb{R}^{m \times n}$ es grande y dispersa, mientras que es un vector tal que $\mathbf{b} \in \mathbb{R}^m$. Es similar al Gradiente Conjugado aplicado a problemas de mínimos cuadrados [Hestenes y Stiefel, 1952]. Este algoritmo es un método basado en el subespacio de Krylov que utiliza el proceso de bidiagonalización de Golub y Kahan [Golub y Kahan, 1965]. Por lo cual, en cada k -ésima iteración se realizan dos recurrencias:

$$\begin{aligned} AV_k &= U_{k+1}B_k \\ A^T U_{k+1} &= V_{k+1}\hat{B}_{k+1} \end{aligned}$$

donde V_k y U_k son bases del subespacio de Krylov de dimensión k .

Las matrices $B_k \in \mathbb{R}^{k+1,k}$ y $\hat{B}_{k+1} \in \mathbb{R}^{k,k+1}$ corresponden a una matriz bidiagonal inferior y una matriz bidiagonal superior respectivamente. Se le llama matriz bidiagonal a aquella que tiene elementos no nulos a lo largo de su diagonal principal y en la primera superdiagonal o subdiagonal. Particularmente, las matrices mencionadas tienen la siguiente forma:

$$B_k = \begin{pmatrix} \beta_{1,1} & & & & & \\ \beta_{2,1} & \beta_{2,2} & & & & \\ & \beta_{3,2} & \beta_{3,3} & & & \\ & & \ddots & \ddots & & \\ & & & \beta_{k,k-1} & \beta_{k,k} & \\ & & & & \beta_{k+1,k} & \end{pmatrix} \quad \hat{B}_{k+1} = \begin{pmatrix} \alpha_{1,1} & \alpha_{1,2} & & & & \\ & \alpha_{2,2} & \alpha_{2,3} & & & \\ & & \ddots & \ddots & & \\ & & & \alpha_{k,k} & \alpha_{k,k+1} & \end{pmatrix}$$

además, $V_k \in \mathbb{R}^{n \times k}$ y $U_k \in \mathbb{R}^{m \times k}$, tienen la característica de que en aritmética exacta cumplen con la propiedad $V_k^T V_k = U_k^T U_k = I_k$. Con esto, en cada iteración se puede aproximar

una solución $\mathbf{x}_k$ que reduzca monótonamente la norma del residuo.

$$\|\mathbf{b} - A\mathbf{x}_k\|$$

Al igual que otros métodos iterativos, la recurrencia requiere la inicialización de las matrices U_k y V_k con los vectores $\mathbf{u}_1$ y $\mathbf{v}_1$ respectivamente, los cuales están definidos por las expresiones,

$$\mathbf{u}_1 = \frac{\mathbf{b}}{\|\mathbf{b}\|}$$
$$\mathbf{v}_1 = \frac{A^T \mathbf{b}}{\|A^T \mathbf{b}\|}$$

A través del procedimiento anterior, se genera el nuevo subespacio de Krylov, dado por las columnas de V_k , mediante el cual se acota la solución a $\mathbf{x}_k = V_k \mathbf{c}_k$, para algún $\mathbf{c}_k \in \mathbb{R}^k$. Así,

$$\begin{aligned} \mathbf{b} - A\mathbf{x}_k &= \|\mathbf{b}\| u_1 - AV_k \mathbf{c}_k \\ &= \|\mathbf{b}\| U_{k+1} \mathbf{e}_1 - U_{k+1} B_k \mathbf{c}_k \\ &= U_{k+1} (\|\mathbf{b}\| \mathbf{e}_1 - B_k \mathbf{c}_k) \end{aligned}$$

De este modo, el problema de mínimos cuadrados se reduce a

$$\min_{\mathbf{x}_k \in \text{range}(V_k)} \|\mathbf{b} - A\mathbf{x}_k\|^2 = \min_{\mathbf{c}_k \in \mathbb{R}^k} \|\|\mathbf{b}\| \mathbf{e}_1 - B_k \mathbf{c}_k\|^2$$

siendo $\mathbf{c}_k$ es la solución que minimiza la ecuación.

El QR en el nombre se obtiene debido a que se obtiene la factorización QR de la matriz B_k con el fin de resolver el problema de mínimos cuadrados de forma eficiente.

2.3. Diferenciación automática

El concepto de diferenciación automática [Griewank y Walther, 2008], también conocida como *Algorithmic Differentiation* o *Automatic Differentiation*, es un método computacional diseñado para calcular la derivada de una función de manera precisa y eficiente mediante la aplicación sistemática de la regla de la cadena.

En matemáticas la regla de la cadena establece que si se tiene una función compuesta $f(g(x))$ donde g y f son funciones diferenciables, entonces la derivada de esta función con respecto a la variable x está dada por

$$\frac{d}{dx}f(g(x)) = f'(g(x))g'(x)$$

La diferenciación automática se basa en que todas las funciones están compuestas por una secuencia finita de operaciones y funciones elementales, como por ejemplo $\pm, \times, \div, \tan(\cdot), \sin(\cdot)$. El propósito de la descomposición es que, aunque no conocemos la derivada de F , si conocemos el de las operaciones en las que se descompone. Los valores obtenidos a través de esta descomposición se almacenan en variables intermedias v_i , cuyo subíndice está asociado al orden en que se realiza la operación. Esto se expresa a través del siguiente vector:

$$\mathbf{v} = \left[\underbrace{v_{1-n}, \dots, v_0}_{\mathbf{x}}, \underbrace{v_1, \dots, v_{l-m}}_{\text{valores intermedios}}, \underbrace{v_{l-(m+1)}, \dots, v_l}_{\mathbf{y}} \right]$$

donde subíndices entre $[1 - n, 0]$ representan los valores de entrada, entre $[1, l - m]$ representan los valores intermedios obtenidos a partir de las funciones elementales y entre $[l - (m + 1), l]$ representan los valores del vector de resultados. Además, n es el largo del vector de entrada $\mathbf{x}$, m el largo del vector de resultados $\mathbf{y}$; y l la cantidad de variables intermedias.

Para obtener cada uno de los valores de v_i es necesario aplicar una función elemental, que en este contexto es denominada φ_i , la cual cumple que:

$$v_i = \varphi_i(v_j)_{j < i}; \varphi_i : \mathbb{R}^{n_i} \rightarrow \mathbb{R} \quad (5)$$

donde cada valor v_i siguiente se obtiene a través de los valores v_j anteriores ($j < i$); cabe notar que n_i representa a la cantidad de argumentos que se necesita para evaluar φ_i , que comunmente suele ser 1 ó 2.

Para una mejor comprensión, ejemplificaremos a través de una F definida como:

$$F : \mathbb{R}^2 \rightarrow \mathbb{R}, F(\mathbf{x}) = \sin(x_1 + 1) + \log(x_2) \quad (6)$$

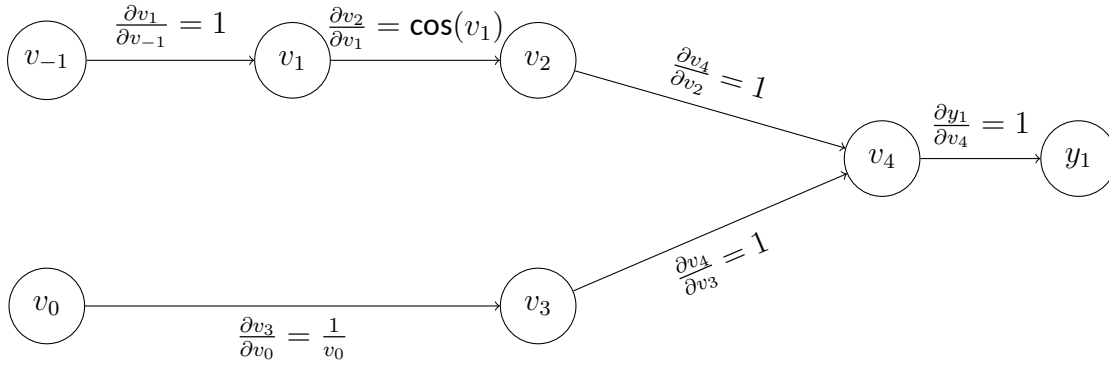
y el punto $\mathbf{x} = (2, 3)$, donde se quiere evaluar la derivada.

v_{-1}	=	x_1	=	2	
v_0	=	x_0	=	3	
v_1	=	$v_{-1} + 1$	=	$2 + 1$	= 3
v_2	=	$\sin(v_1)$	=	$\sin(3)$	= 0.141
v_3	=	$\log(v_0)$	=	$\log(3)$	= 1.09
v_4	=	$v_2 + v_3$	=	$0.141 + 1.09$	= 1.231
y_1	=	v_4	=	1.231	

Cuadro 1: Three-part F
Fuente: Elaboración Propia.

La Tabla (1) es una representación llamada *three-part form*, la cual muestra la descomposición de la función en las variables v_i , explicita las funciones elementales que componen la función y el trayecto que se realiza para computar el valor de la función F .

Además de la tabla anterior, otra forma en la que se puede representar la función es a través de un grafo computacional, el cual además de explicitar el recorrido y el orden, añade también la derivada que relaciona a estas variables.

Cuadro 2: Grafo computacional F

Fuente: Elaboración propia.

En el Grafo (2), cada vértice equivale a las variables v_i mientras que las aristas, la derivada que relaciona a estas variables.

Existen dos métodos distintos a través de los cuales se puede aplicar la diferenciación automática: *forward mode* y *reverse mode*.

2.3.1. Forward mode

También conocido como *tangent mode* o método de la tangente, es la forma más sencilla de aplicar AD. La idea es ir propagando la derivada a medida de que se obtiene el valor de la función. Desde un punto de vista matemático, este modo se puede interpretar como el cálculo de la tangente. Supongamos que tenemos una curva dependiente del tiempo $x(t)$, donde el punto $t = 0$ se representa simplemente como x . La tangente de este punto está dado por la expresión $\dot{x} = \partial x(t) / \partial t|_{t_0}$ que no es más que la derivada de x con respecto al tiempo en el punto dado. Luego, si quisiéramos obtener el valor de la tangente $\dot{y}$ de una función compuesta $y = F(x(t))$ con F suficientemente suave de la siguiente forma

$$\dot{y} = \frac{\partial}{\partial t} F(x(t))|_{t=0} = F'(x(t)) \frac{\partial x(t)}{\partial t}|_{t=0} = F'(x) \dot{x} \equiv F(x, \dot{x})$$

Visto desde la perspectiva de funciones elementales

$$\dot{\varphi}_i : \mathbb{R}^{2n_i} \rightarrow \mathbb{R}, \dot{v}_i = \nabla \varphi_i(u_i) \dot{u}_i \equiv \dot{\varphi}_i(u_i, \dot{u}_i)$$

$$\dot{v}_i = \sum_{j \prec i} \frac{\partial}{\partial v_j} \varphi_i(u_i) \dot{v}_j$$

donde $u_i = (v_j)_{j \prec i}$ y $\dot{u}_i = (\dot{v}_j)_{j \prec i}$

En el Cuadro (3) y Cuadro (4) se muestra el desarrollo del *forward mode* para el ejemplo (6)

F	F'
$v_{-1} = x_1 = 2$ $v_0 = x_2 = 3$	$\dot{v}_{-1} = \dot{x}_1 = 1$ $\dot{v}_0 = \dot{x}_2 = 0$
$v_1 = v_{-1} + 1 = 2 + 1$ $v_2 = \sin(v_1) = \sin(3)$ $v_3 = \log(v_0) = \log(3)$ $v_4 = v_2 + v_3 = 0.141 + 1.098$	$\dot{v}_1 = \dot{v}_{-1} = 1$ $\dot{v}_2 = \cos(v_1) \cdot \dot{v}_1 = \cos(3) \cdot 1$ $\dot{v}_3 = \frac{1}{v_0} \cdot \dot{v}_0 = \frac{1}{3} \cdot 0$ $\dot{v}_4 = \dot{v}_2 + \dot{v}_3 = -0.989 + 0$
$y_1 = v_4 = 1.238$	$\dot{y}_1 = \dot{v}_4 = -0.989$

Cuadro 3: forward mode iteración 1

Fuente: Elaboración propia.

F	F'
$v_{-1} = x_1 = 2$ $v_0 = x_2 = 3$	$\dot{v}_{-1} = \dot{x}_1 = 0$ $\dot{v}_0 = \dot{x}_2 = 1$
$v_1 = v_{-1} + 1 = 2 + 1$ $v_2 = \sin(v_1) = \sin(3)$ $v_3 = \log(v_0) = \log(3)$ $v_4 = v_2 + v_3 = 0.141 + 1.098$	$\dot{v}_1 = \dot{v}_{-1} = 0$ $\dot{v}_2 = \cos(v_1) \cdot \dot{v}_1 = \cos(3) \cdot 0$ $\dot{v}_3 = \frac{1}{v_0} \cdot \dot{v}_0 = \frac{1}{3} \cdot 1$ $\dot{v}_4 = \dot{v}_2 + \dot{v}_3 = 0 + 0.333$
$y_1 = v_4 = 1.238$	$\dot{y}_1 = \dot{v}_4 = 0.333$

Cuadro 4: forward mode iteración 2

Fuente: Elaboración propia.

A través de dos iteraciones obtenemos en su totalidad la matriz Jacobiana evaluada, cuyos valores son:

$$J_F = \begin{pmatrix} -0.989 & 0.333 \end{pmatrix}$$

Las flechas a la derecha de cada cuadro indica el orden en que se aplica la diferenciación algorítmica, en este caso se hace en el mismo orden en que se evalúa la función.

2.3.2. Reverse mode

También conocido como *adjoint mode*, es el segundo modo de aplicar AD. Este, en lugar de propagar la derivada junto a la evaluación de la función, lo realiza de forma inversa, comenzando desde el resultado de la función evaluada y hacia x luego de finalizada la evaluación. Matemáticamente se puede entender como dado un vector $\bar{y} \in \mathbb{R}^m$ y dado un valor $c \in \mathbb{R}$, el hiperplano $\{y \in \mathbb{R}^m | \bar{y}^T y = c\}$ en el rango de F , misma trabajada en el método anterior. El hiperplano tiene la inversa $\{x \in \mathbb{R}^n | \bar{y}^T F(x) = c\}$. Por el teorema de la función implícita sabemos que dicho conjunto es suave con una normal $\bar{x}^T = \bar{y}^T F'(x) \equiv \bar{F}(x, \bar{y})$ a x , demostrando que $\bar{x}^T$ no se desvanece. $\bar{y}$ y $\bar{x}$ se pueden ver geométricamente como normales o cotangentes.

Para cada función elemental $\varphi_i : \mathbb{R}^{n_i} \rightarrow \mathbb{R}$ se obtiene la función adjunta:

$$\bar{u}_i + = \bar{v}_i \nabla \varphi_i(\bar{u}_i) \text{ para } \bar{u}_i \equiv (\bar{v}_j)_{j \prec i} \in \mathbb{R}^{n_i}$$

Sin embargo, esta explicación es un poco más compleja de entender, por ello también se presenta una explicación alternativa.

Se define el vector v_i de la forma

$$v_i = [v_j, v_{j+1}, \dots, v_i, 0, \dots, 0]$$

es decir, es un vector que tiene valores desde v_j hasta v_i . Estos vectores están asociado a una función llamada *State Transform* Φ_i la cual es equivalente a aplicar la función φ_i pero

ahora, en lugar de trabajar sobre un número real, se realiza sobre un vector, es decir

$$\mathbf{v}_i = \Phi_i(\mathbf{v}_{i-1}), \text{ donde } \Phi_i : \mathbb{R}^{n+l} \rightarrow \mathbb{R}^{n+l}$$

Además, la derivada de las funciones elementales asociadas a las variables intermedias está dada por la expresión

$$c_{ij} \equiv c_{ij}(\mathbf{v}_i) = \frac{\partial \varphi_i}{\partial v_j} \text{ para } 1 - n \leq i, j \leq l$$

donde n representa el largo del vector de entrada y l el largo del vector de salida.

Con ello, podemos expresar la función $\mathbf{F}$ a través de la composición de todos los Φ_i que permiten obtener $\mathbf{v}$, recordando que los últimos m elementos de dicho vector entregan el valor Y

$$Y = Q_m \Phi_l \circ \Phi_{l-1} \circ \dots \circ \Phi_2 \circ \Phi_1(P_n^T X)$$

Hay que detenerse un momento debido a que es importante comprender las dimensiones sobre lo que se está trabajando. Se tiene que $X \in \mathbb{R}^n$ e $Y \in \mathbb{R}^m$, pero las operaciones Φ_i trabajan sobre un vector $\mathbb{R}^{n+l}$. Para ello, P_n se encarga de transformar X en el vector $\mathbf{v}$ de tamaño $\mathbb{R}^{n+l}$, mientras que Q_m extrae los últimos m valores del vector. Es por ello, que tienen la siguiente forma:

$$P_n = \begin{pmatrix} I_n & \mathbf{0}_{n \times l} \end{pmatrix}$$

$$Q_m = \begin{pmatrix} \mathbf{0}_{1 \times l-m} & I_m \end{pmatrix}$$

Luego, para conocer la derivada de la función compuesta, se requiere conocer las derivadas

de cada una de las funciones Φ_i que la componen

$$A_i = \Phi'_i = \begin{pmatrix} 1 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & 0 & 0 & \cdots & 0 \\ c_{i1-n} & c_{i1-n} & \cdots & c_{ii-1} & 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 1 \end{pmatrix}$$

$$A_i = I + e_{n+1} [\nabla\varphi(v_i) - e_{n+i}]^T$$

En palabras, la derivada de Φ_i es la matriz identidad I_{n+l} , pero en la fila i almacena los valores de las derivadas en dichas coordenadas, con un valor nulo en la diagonal. Con ello, podemos obtener la derivada:

$$\dot{Y} = Q_m A_l A_{l-1} \cdots A_2 A_1 P_n^T \dot{X}$$

A través de la identidad $\bar{y}^T \dot{y} = \bar{x}^T \dot{x}$ se obtiene la relación que da como resultado el método reverso

$$\bar{X} = P_n A_1^T A_2^T \cdots A_{l-1}^T A_l^T Q_m^T \bar{Y}$$

En el Cuadro (5) se muestra el desarrollo del *reverse mode* para el ejemplo (6)

En este caso la flecha indica que la evaluación de la derivada se hace una vez finalizada la evaluación de la función.

Finalmente, obtenemos el valor de la derivada

$$J_F = \begin{pmatrix} -0.989 & 0.333 \end{pmatrix}$$

Aunque los dos métodos retornan el mismo resultado, la diferencia está en cómo lo entregan. A través del *Forward Mode* el algoritmo retorna una a una las columnas de la matriz Jacobiana, en cambio, el *Reverse Mode* entrega una a una las filas de la matriz. Debido a esto, la elección sobre cuál de los dos métodos utilizar dependerá netamente de cómo es la

función sobre la cual se va a aplicar.

v_i	valor	φ_i	$\nabla \varphi_i$	F'	$A_l^T \bar{v}_1$
v_{-1}	x_1	—	—	$\bar{x}_1$	$\cos(v_1) = -0.989$
v_0	x_2	—	—	$\bar{x}_2$	$\frac{1}{v_0} = 0.333$
v_1	—	$v_{-1} + 1$	$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$	—	$\begin{bmatrix} \cos(v_1) & \frac{1}{v_0} & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$
v_2	—	$\sin(v_1)$	$\begin{bmatrix} 0 & 0 & \cos(v_1) & 0 & 0 & 0 & 0 \end{bmatrix}$	—	$\begin{bmatrix} 0 & \frac{1}{v_0} & \cos(v_1) & 0 & 0 & 0 & 0 \end{bmatrix}$
v_3	—	$\log(v_0)$	$\begin{bmatrix} 0 & \frac{1}{v_0} & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$	—	$\begin{bmatrix} 0 & \frac{1}{v_0} & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$
v_4	—	$v_2 + v_3$	$\begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix}$	—	$\begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix}$
v_5	—	v_4	$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$	—	$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$
y	—	v_5	—	$Q_m^T \bar{y}$	$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$

Cuadro 5: Reverse mode iteración 1

Fuente: Elaboración propia.

CAPÍTULO 3

PROPUESTA DE SOLUCIÓN

Antes de comenzar con la descripción de la propuesta de solución, es menester tener presentes el objetivo que da comienzo al desarrollo de este problema: encontrar un método para resolver problemas no lineales sobredeterminados en donde el uso de memoria y computación sea razonable.

Para ello, y por la naturaleza de este trabajo, se parte con la selección del lenguaje de programación en el que se desarrollará el algoritmo. En el mercado existen distintas opciones orientadas al análisis numérico y computación científica, dentro de los cuales se encuentran, por ejemplo, Python, Julia o R; cada uno con sus propias cualidades. Particularmente, se eligió el lenguaje más popular dentro del área, Python, no solo debido a la sencillez de la sintaxis y fácil lectura, sino porque tiene una comunidad activa y una gran cantidad de bibliotecas con diversos módulos, que lo vuelve un lenguaje muy versátil. Cabe destacar que se probaron otras opciones, pero estas iban asociadas a un tiempo de aprendizaje adicional.

A continuación, es necesario buscar una herramienta que permitiera la correcta implementación de *Algorithmic differentiation*. JAX [Johnson y Levskaia, 2023] es un *frameworks* que combina Autograd y XLA, lo que la convierte en una herramienta de alto rendimiento para la implementación de la diferenciación automática. La innovación se encuentra en el uso de XLA, lo cual permite compilar y ejecutar programas NumPy, además de sus propias funciones de Python, en aceleradores GPU y TPU. La gran ventaja que ofrece sobre otros *frameworks* se puede observar en su rendimiento: JAX utiliza compilación JIT para optimizar cálculos, lo que resulta en tiempos de ejecución más rápidos. Un ejemplo particular [O'Connor, 2023], demuestra que para una función definida como

$$F(A) = A + A \cdot A + A \cdot A \cdot A$$

donde A es una matriz de dimensiones 5000×5000 , JAX trabajando en CPU llega a ser 8.5 veces más veloz que NumPy, mientras que en TPU resulta 9.3 veces más rápida que su contraparte. Sin embargo, hay que recalcar que esta herramienta sigue en desarrollo, razón por la cuál aún tiene una comunidad pequeña y algunas funcionalidades no tienen una implementación robusta.

El corazón de este trabajo es la implementación de un algoritmo llamado nsLSQR [Sanhueza, 2021], incorporando funciones de diferenciación automática. Para ello, primero se requiere explicar el algoritmo con mayor profundidad

3.1. Algoritmo nsLSQR

Para explicar este algoritmo y las modificaciones propuestas, es necesario partir por el problema de mínimos cuadrados. Para ello comenzamos considerando el siguiente problema,

$$\min_{\mathbf{x} \in \mathbb{R}^n} \|\mathbf{b} - J\mathbf{x}\|^2 \quad (7)$$

Asociado a las recurrencias:

$$JV_k = U_{k+1}B_k \quad (8)$$

$$J^T U_{k+1} = V_{k+1}\hat{B}_{k+1} \quad (9)$$

El método nsLSQR [Sanhueza, 2021], está basado en el LSQR, levemente explicado en la sección anterior. Tengamos presente que este método requiere del uso de dos rutinas: la primera para calcular $J(\mathbf{x})\mathbf{v}$ y otra para obtener el valor de $J(\mathbf{x})^T \mathbf{w}$. El primer producto se puede calcular de distintas formas, por lo que es fácil de obtener; el segundo, por otra parte, plantea mayores dificultades en particular cuando se trabaja sobre problemas de mayor magnitud. Es en este punto es donde el nsLSQR plantea un cambio, alterando las rutinas de la siguiente forma

- Un procedimiento para calcular $J(\mathbf{x})\mathbf{v}$.
- Un procedimiento para aproximar $J(\mathbf{x})^T \mathbf{w}$.

La idea del cambio es que ahora se puedan implementar procedimientos que entreguen una aproximación más accesible, en temas de computación y memoria, y aún así lograr resolver el problema. No obstante, el uso de malas aproximaciones puede relentizar considerablemente la convergencia del método. De ahora en adelante usaremos simplemente J haciendo referencia a $J(\mathbf{x})$ y J^T , para referirnos a $J(\mathbf{x})^T$ para mayor simpleza.

Supongamos que tenemos una aproximación de J , denotada como $\tilde{J}$, la cual tiene la cualidad de requerir menos espacio de memoria para ser almacenada. En rigor, esta matriz puede resultar ser una muy buena aproximación tanto como una muy imprecisa, pero aún así es un medio por el cual se puede buscar una aproximación del producto $J^T \mathbf{w}$ de la forma $\tilde{J}^T \mathbf{w}$. El uso de esta aproximación implican algunos cambios importantes:

1. Las matrices B_k y $\hat{B}_k$ dejan de ser matrices bidiagonales.
2. Cambia el subespacio de solución definido por V_k .

El primer punto es importante debido a que la bidiagonalización de Golub y Kahan [Golub y Kahan, 1965] ya no será válida. El siguiente indica que, debido a un cálculo no exacto o aproximado, los vectores $\mathbf{v}_{k+1}$ generados en ambos algoritmos serán distintos y por ende, el espacio generado en ambos será diferente. Aún así es importante destacar que mientras se cumpla que las columnas de V_k sean linealmente independientes, entonces este subespacio o base es válido para resolver el problema de mínimos cuadrados.

Para resolver el problema de mínimos cuadrados con nsLSQR, primero hay que obtener los vectores iniciales:

$$\begin{aligned}\tilde{\mathbf{u}}_1 &= \frac{\mathbf{b}}{\|\mathbf{b}\|} \\ \tilde{\mathbf{v}}_1 &= \frac{\tilde{J}^T \mathbf{b}}{\|\tilde{J}^T \mathbf{b}\|}\end{aligned}$$

A pesar de que $\mathbf{u}_1 = \tilde{\mathbf{u}}_1$, los siguientes vectores $\tilde{\mathbf{u}}_{k+1}$ no necesariamente son iguales a sus contrapartes de LSQR. La k -ésima columna de (8) se computa da través de,

$$J\tilde{\mathbf{v}}_k = \beta_{1,k}\tilde{\mathbf{u}}_1 + \beta_{2,k}\tilde{\mathbf{u}}_2 + \cdots + \beta_{k+1,k}\tilde{\mathbf{u}}_{k+1} \quad (10)$$

Lo que modifica la recurrencia original, transformándolo a la expresión

$$J\tilde{V}_k = \tilde{U}_{k+1}\tilde{B}_k$$

donde $\tilde{B}_k$ adquiere la forma de una matriz de Hessenberg con dimensiones $(k+1) \times k$. De esta recurrencia obtenemos $\tilde{\mathbf{u}}_{k+1}$. De forma similar, la $k+1$ -ésima columna de (9) se obtiene a partir de la descomposición

$$J^T \tilde{\mathbf{u}}_{k+1} \approx \tilde{J}^T \tilde{\mathbf{u}}_{k+1} = \alpha_{1,k+1} \tilde{\mathbf{v}}_1 + \alpha_{2,k+1} \tilde{\mathbf{v}}_2 + \cdots + \alpha_{k+1,k+1} \tilde{\mathbf{v}}_{k+1} \quad (11)$$

De la cual obtenemos la recurrencia:

$$J\tilde{U}_k = \tilde{V}_{k+1}\bar{B}_k$$

donde la matriz $\bar{B}_k$ es una matriz triangular superior de tamaño $(k+1) \times (k+1)$. Notamos que $\tilde{U}_k \neq U_k$. Esta recurrencia nos permite obtener $\tilde{\mathbf{v}}_{k+1}$, bajo la condición $\alpha_{k+1,k+1} \neq 0$, en cada iteración.

Tanto (10) como (11) se pueden obtener mediante Gram-Schmidt modificado.

Ahora, para cada k -iteración restringimos el espacio de solución al espacio generado por las columnas de $\tilde{V}_k$, lo que significa $\tilde{\mathbf{x}}_k = \tilde{V}_k \tilde{\mathbf{c}}_k$

Luego, somos capaces de reducir el problema de mínimos cuadrados:

$$\min_{\mathbf{x}_k \in \text{range}(\tilde{V}_k)} \|\mathbf{b} - J\mathbf{x}_k\|^2 = \min_{\tilde{\mathbf{c}}_k \in \mathbb{R}^k} \|\|\mathbf{b}\| \mathbf{e}_1^{(k+1)} - \tilde{B}_k \tilde{\mathbf{c}}_k\|^2 \quad (12)$$

Finalmente, hay que notar que nsLSQR requiere almacenar algunas matrices densas: $\tilde{V}_k$, $\tilde{U}_k$ y $\tilde{B}_k$, cuyo requerimiento de memoria depende del número de iteraciones k , a mayor k más memoria es requerida para su almacenamiento. Para combatir esto, se puede utilizar una estrategia de reinicio a través de dos parámetros t_{in} y t_{out} . Con ello, el algoritmo itera t_{in} para encontrar una solución aproximada y luego, esta solución se utiliza como una nueva solución, liberando el espacio utilizado por las matrices antes mencionadas. Utilizando este enfoque, la matriz más grande almacenada tendrá el tamaño $m \times t_{in}$. Si $t_{in} < n$, el método utilizará menos memoria en comparación a un enfoque sin reinicio.

En Algorithm 1 se presenta la forma general del algoritmo nsLSQR, sin considerar el caso de

término cuando $\alpha_{k+1,k+1} = 0$ y $\beta_{k+1,k} = 0$

Algorithm 1 Algoritmo general nsLSQR

Require: Una función F , un vector $\mathbf{x}_i$ para evaluar la matriz jacobiana, una aproximación $\tilde{J}$, un vector $\mathbf{b}$, tolerancia η , cantidad de iteraciones t_{in}, t_{out} , una solución inicial $\mathbf{x}_0$ usualmente un vector de 0s

Ensure: Una solución aproximada de $\mathbf{x}$

```
1: for  $i$  de 1 hasta  $t_{out}$  do
2:    $\mathbf{r} := \mathbf{b} - J\mathbf{x}_0$ 
3:    $\tilde{\mathbf{u}}_1 := \frac{\mathbf{r}}{\|\mathbf{r}\|}$ 
4:    $\tilde{\mathbf{v}}_1 := \frac{\tilde{J}^T \mathbf{r}}{\|\tilde{J}^T \mathbf{r}\|}$ 
5:   for  $k$  de 1 hasta  $t_{in}$  do
6:     Calcular  $\tilde{\mathbf{u}}_{k+1}$  y  $b_k$  usando Gram-Schmidt modificado.
7:     Calcular  $\tilde{\mathbf{v}}_{k+1}$  usando Gram-Schmidt modificado.
8:     Resolver el problema de mínimos cuadrados para  $\mathbf{c}_k$ 
9:      $\mathbf{x}_k := V_k \tilde{\mathbf{c}}_k$ 
10:    if Se cumple algun criterio de parada utilizando la tolerancia  $\eta$  then
11:      Retorna  $\mathbf{x}_k$ 
12:    end if
13:  end for
14:   $\mathbf{x}_0 := \mathbf{x}_0 + \mathbf{x}_k$ 
15: end for
16: Retorna  $\mathbf{x}_k$ 
```

3.1.1. nsLSQR + JVP/VJP

Antes de precisar la incorporación de la diferenciación automática en nsLSQR, es relevante entender los conceptos JVP y VJP.

Jacobian-Vector-Product (JVP) es el paso fundamental de *forward-mode*. En la sección anterior revisamos los modos de aplicación de la diferenciación automática y un punto importante fue notar que cuando se aplica *forward-mode*, en cada paso se regresa una columna de la matriz Jacobiana. Cuando analizamos este proceso más a profundidad se puede apreciar que se puede representar a través del siguiente producto,

$$\underbrace{\begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial x_3} & \dots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial x_3} & \dots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \frac{\partial f_m}{\partial x_3} & \dots & \frac{\partial f_m}{\partial x_n} \end{pmatrix}}_{\partial f} \underbrace{\begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ \vdots \\ v_n \end{pmatrix}}_v$$

Cuando el vector v es una base canónica de $\mathbb{R}^n$ entonces obtenemos exactamente el resultado de aplicar *forward-mode*. Lo interesante ocurre cuando este es un vector cualquiera, pues en ese caso el resultado es exactamente el producto entre la matriz Jacobiana con un vector Jv .

Por otra parte, de forma símil existe Vector-Jacobian-Product (VJP), que es la base del *reverse-mode*. En este caso, si profundizamos un poco en la forma que adquiere tenemos lo siguiente,

$$\underbrace{\begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ \vdots \\ w_m \end{pmatrix}}_w^T \underbrace{\begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial x_3} & \dots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial x_3} & \dots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \frac{\partial f_m}{\partial x_3} & \dots & \frac{\partial f_m}{\partial x_n} \end{pmatrix}}_{\partial f}$$

Al igual que en el caso anterior, si el vector w es una base canónica de $\mathbb{R}^m$ entonces obtenemos exactamente el resultado de aplicar *reverse-mode*. En cambio, si utilizamos un vector

cualquiera, entonces obtenemos el producto entre un vector y una matriz jacobina $\mathbf{w}^T J$. Quizás con esa forma no queda completamente claro su utilización dentro de nsLSQR, pero si reescribimos la función se puede lograr una expresión más clara $J^T \mathbf{w}$.

Es importante mencionar que no es necesario obtener las matrices Jacobianas de forma explícita para calcular los productos respectivos y es por ello que estas operaciones son tan poderosas.

Retomando la primera idea del algoritmo, enfoquémonos en las rutinas necesarias para llevarlo a cabo, pues es aquí donde este algoritmo se puede beneficiar de la diferenciación automática:

- Un procedimiento para calcular $J \mathbf{v}$.
- Un procedimiento para aproximar $J^T \mathbf{w}$.

Para abarcar dichos puntos utilizamos la diferenciación automática a través de los métodos mencionados, con lo cual, el primero estará dado por JVP; mientras que el segundo será retornado al utilizar VJP.

A continuación, el Algoritmo (2) presenta una versión completa del algoritmo con los cambios respectivos para adicionar diferenciación automática. En Anexo A se encontrarán los detalles que no son parte de esta memoria, pero si relevantes para la implementación.

Algorithm 2 nsLSQR modificado

Require: Una función jvp , una función vjp , un vector $\mathbf{x}_1$ para evaluar los productos jacobian-vector y vector-jacobian, un vector $\mathbf{b}$, tolerancia η_t , η_d y η_s , enteros s y d , cantidad de iteraciones t_{in} , t_{out} , una solución inicial $\mathbf{x}_0$ usualmente un vector de 0s

Ensure: Una solución aproximada de $\mathbf{x}$

```
1: for  $i$  de 1 hasta  $t_{out}$  do
2:    $\mathbf{r} := \mathbf{b} - \text{jvp}(\mathbf{x}_0)$ 
3:    $\tilde{\mathbf{u}}_1 := \frac{\mathbf{r}}{\|\mathbf{r}\|}$ 
4:    $\tilde{\mathbf{v}}_1 := \frac{\text{vjp}(\mathbf{r})}{\|\text{vjp}(\mathbf{r})\|}$ 
5:   for  $k$  de 1 hasta  $t_{in}$  do
6:     Calcular  $\tilde{\mathbf{u}}_{k+1}$  y  $b_k$  usando Gram-Schmidt modificado.
7:     Calcular  $\tilde{\mathbf{v}}_{k+1}$  usando Gram-Schmidt modificado.
8:     if  $k$  es 1 then
9:       Construir la descomposición QR inicial,  $Q_1$  y  $R_1$ 
10:      Calcular  $\tilde{\mathbf{c}}_1 = \|\mathbf{b}\| R^{-1} Q_1^T \mathbf{e}_1^{(2)}$  por computación directa
11:    else
12:      Resolver el problema de mínimos cuadrados para obtener  $\mathbf{c}_k$ 
13:    end if
14:     $\mathbf{x}_k = V_k \tilde{\mathbf{c}}_k$ 
15:    if Se cumple algun criterio de parada then
16:      Retorna  $\mathbf{x}_k$ 
17:    end if
18:  end for
19:   $\mathbf{x}_0 := \mathbf{x}_0 + \mathbf{x}_k$ 
20: end for
21: Retorna  $\mathbf{x}_k$ 
```

3.2. Aproximación de J

Para concluir nuestra propuesta de solución, es esencial abordar la forma de obtener una estimación del producto entre la matriz Jacobiana transpuesta y un vector, es decir, $\tilde{J}^T \mathbf{w}$.

Para lograrlo, es importante recordar que utilizaremos la diferenciación automática para obtener estos productos. Por lo tanto, comenzaremos examinando la firma de las funciones necesarias para lograr este objetivo:

```
primals, tangents = jax.jvp(func, x, v)
primals, func_vjp = jax.vjp(func, x)
```

donde `func` es un objeto *callable* que representa a la función F , x un vector donde se evaluará la derivada. Cabe mencionar que los retornos de las funciones son diferentes, la primera función retorna el producto en el primera llamado, mientras que la segunda retorna una función que, al evaluarla en el vector $\mathbf{w}$ se obtiene el producto, esto hace que realizar la operación $J^T \mathbf{w}$ se transforme en simplemente evaluar una función.

Si ambas funciones son llamadas con F , entonces obtendremos $J\mathbf{v}$ y $J^T \mathbf{w}$ respectivamente, por esta razón es que se propone redefinir la función realizando una modificación a F de tal forma en que cuando se realice el llamado a la función `jax.vjp` el resultado que entregue sea $\tilde{J}^T \mathbf{w}$. De esta forma, se creará una nueva función F definida como:

$$F(x) = \begin{cases} F_1(x) & \text{si } J\mathbf{v} \\ F_2(x) & \text{si } \tilde{J}^T \mathbf{w} \end{cases}$$

donde F_1 representa la función original a partir de la cual obtenemos el producto $J\mathbf{v}$, mientras que F_2 es una versión modificada que introduce diversificación en la matriz Jacobiana. En otras palabras, nuestro objetivo es que un cierto porcentaje de los elementos que componen J sean nulos. A continuación, con esta aproximación procederemos a calcular el producto $\tilde{J}^T \mathbf{w}$.

Para lograrlo, retomemos la ecuación (1) que es una forma de expresar un problema sobre-determinado. La idea subyacente es que no todas las funciones f_m que componen a F se elijan para formar parte de la nueva función F_2 ; en su lugar, conservaremos solo un subconjunto pequeño de valores. Esto se logra mediante la introducción de una probabilidad p_m , la

cual estará asociada a cada función f_m , y determinará si el candidato es seleccionado o no para la nueva Fx_2 . Esto provoca que solo un porcentaje de filas se mantenga y el resto de ellas, estarán compuestas por elementos 0.

Esta variación en la función ofrece dos ventajas destacables: en primer lugar, permite el uso de formas especiales para almacenar matrices diversas que ocupan menos espacio en memoria; en segundo lugar, requiere menos operaciones para realizar cálculos, ya que muchas de estas operaciones darán como resultado valores ceros.

CAPÍTULO 4

VALIDACIÓN DE LA SOLUCIÓN

Esta sección se enfoca en verificar el rendimiento del algoritmo nsLSQR en conjunto con la diferenciación automática. A través de pruebas definidas, evaluaremos el funcionamiento del método, el tiempo de ejecución, la memoria necesaria para computar $\tilde{J}^T \mathbf{w}$ a través de la función vjp y la memoria que ocupa las aproximaciones $\tilde{J}$ obtenidas a través de $\mathbf{F}_2$. De esta manera, analizaremos si las modificaciones cumplen con nuestras expectativas.

Para el entorno de pruebas es importante destacar que todas las pruebas se llevaron a cabo en un entorno de ejecución de Google Colab, el cual está equipado con un acelerador de hardware NVIDIA GPU Tesla T4, aprovechando así las ventajas de JAX.

Las funciones seleccionadas para llevar a cabo la experimentación fueron las siguientes:

1. Una función lineal $\mathbf{F}(\mathbf{x}) = \mathbf{b} - A\mathbf{x}$, donde cada elemento de la matriz A y vector $\mathbf{b}$ siguen una distribución normal $\mathcal{N}(0, 1)$. La matriz A es de dimensiones $m \times n$, denominado *Caso lineal*
2. Una función trigonométrica modificada basada en [Moré, 2006], obtenida en [Sanhueza, 2021]. Está definida por m funciones y n incógnitas, donde solo se requiere que $m \geq n$. Se define

$$f_k(\mathbf{x}) = n + k(1 - \cos(x_{i+1})) - \sin(x_{i+1}) - \sum_{j=1}^n \cos^{p+1}(x_j)$$

donde p es la división entera entre k y n . Este se denominó *Caso no lineal*

Cada uno de los experimentos se repitió 5 veces, cambiando el porcentaje que afecta la diversidad de la matriz Jacobiana en los siguientes valores: 50 %, 30 %, 25 %, 10 % y 2 %. El

objetivo principal de estos experimentos es analizar cómo se ven afectados los resultados al utilizar aproximaciones más o menos densas, asociadas a mejores o peores aproximaciones, y cómo esto influye en el cálculo del resultado.

4.1. Memoria y tiempo VJP

El primer punto importante a validar es el uso de la diferenciación algorítmica. Los criterios que se revisarán para esta sección serán la memoria que utiliza la aproximación de $\tilde{J}$ y el tiempo que demora el cálculo del producto vector matriz. Esto se realizó sobre la función VJP de forma aislada.

Las dimensiones que se seleccionaron para cada una de las funciones son:

1. *Caso lineal*: $m = 10000$ y $n = 8000$
2. *Caso no lineal*: $m = 8000$ y $n = 2000$

El vector $\mathbf{x}$ utilizado para evaluar la matriz Jacobiana y su aproximación está distribuido uniformemente en el intervalo $[-1, 1]$, situación que se repite con el vector $\mathbf{w}$.

Es importante destacar que el módulo de JAX destinado al manejo de matrices *sparse* se encuentra actualmente en una fase de desarrollo. En consecuencia, no existe una manera precisa de determinar la cantidad de memoria que utiliza este tipo de matrices de forma directa. Por esta razón, se optó por emplear la biblioteca *SciPy* para obtener una estimación de este valor, dado que cuenta con un sólido soporte para matrices dispersas. Por lo tanto, lo que se obtuvo es una aproximación del espacio que esta matriz podría estar utilizando.

En la Figura 1, se puede observar un comportamiento similar en la disminución del tiempo a medida que disminuye el porcentaje de elementos no nulos. Sin embargo, en (1c), se aprecia una disminución más pronunciada, lo que se refleja en la mayor inclinación de la pendiente. Por otro lado, en cuanto a la memoria en el caso no lineal, se observa una disminución más marcada a partir del 50 % de elementos no nulos. A partir de este punto, se mantiene un comportamiento lineal similar al mostrado en (1b). A pesar de estas similitudes en el comportamiento, es importante destacar las diferencias en las magnitudes. El tiempo necesario

para el cálculo en el caso no lineal resulta ser mayor, incluso cuando las dimensiones del problema son menores. Esta discrepancia podría estar relacionada con la mayor complejidad del cálculo en este escenario.

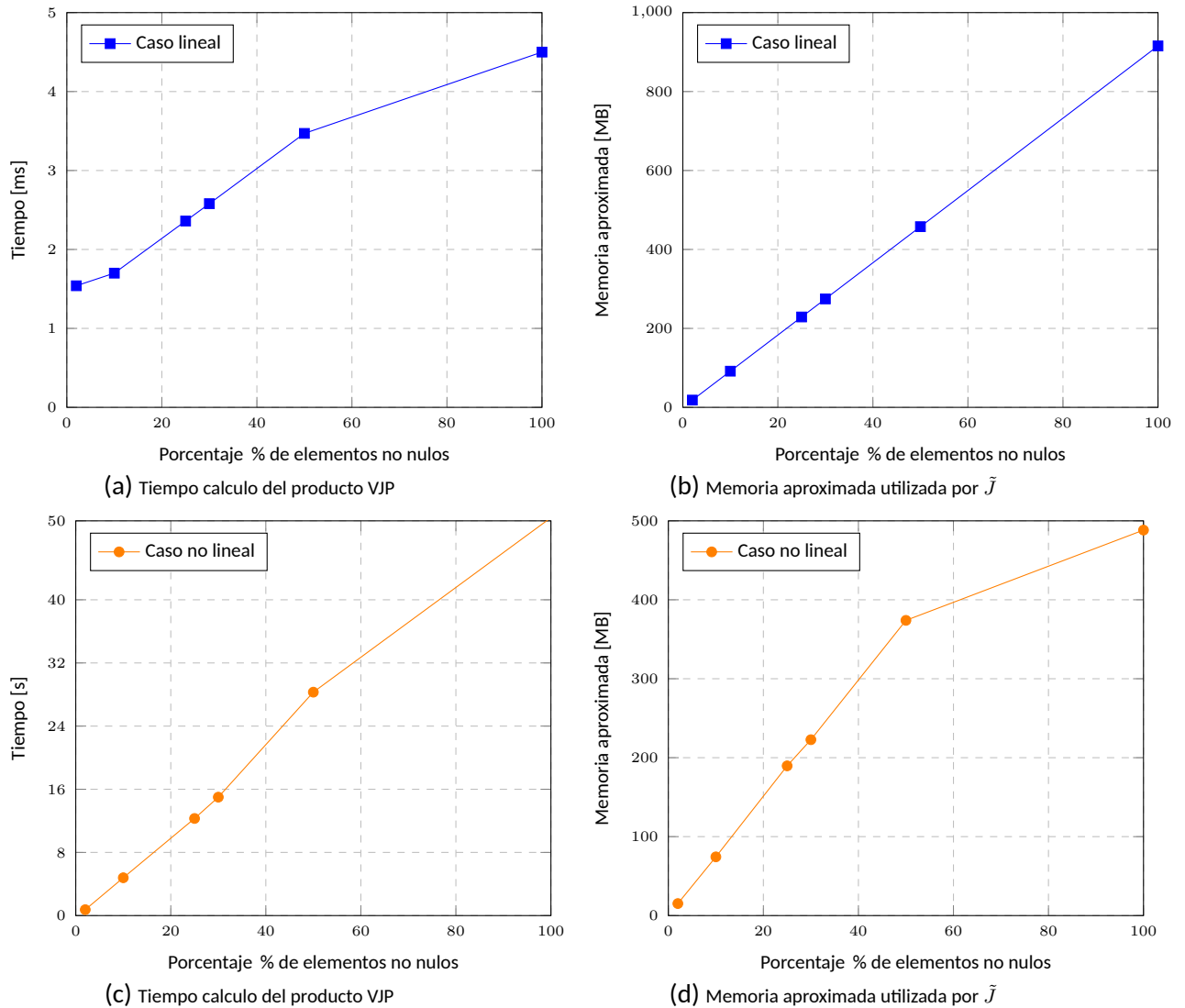


Figura 1: Tiempo y memoria que utiliza $\tilde{J}$ y el calculo asociado a ella (VJP). La abscisa indica el porcentaje de la matriz que se utilizó para obtener la aproximación.

A continuación, en el Cuadro 6, se presenta un registro de la memoria asignada por Python al método VJP para diferentes aproximaciones de la función F . Es importante mencionar que esta medición se realizó únicamente para el caso no lineal. La recopilación de datos se llevó a cabo mediante la función *magic function %mprun*, que permite realizar un perfilado línea

por línea del código. Para comprender este cuadro, es necesario entender la función VJP, que se define de la siguiente manera:

```
primals, f_vjp = jax.vjp(F, x)
```

Esta función devuelve una tupla compuesta por un vector y un objeto callable. Al realizar el perfilado de la función, se asigna inicialmente memoria a la propia función VJP, y esta información se muestra en la columna “Memoria utilizada”. En cambio, el incremento de memoria necesario para la función VJP se registra en la columna “Incremento” en relación con el objeto callable `f_vjp`. Finalmente, es posible apreciar la memoria implementada en reducción a medida de que se acota la función original.

Porcentaje	Memoria usada [MiB]	Incremento [MiB]	peak [MiB]
100 %	1657.7	953.0	2610.7
50 %	1644.0	504.2	2148.2
30 %	1655.7	292.8	1948.5
25 %	1657.4	242.7	1900.1
10 %	1645.4	111.2	1756.6
2 %	1656.4	24.3	1680.6

Cuadro 6: Memoria VJP
Fuente: Elaboración propia

4.2. Tiempos de ejecución en iteraciones internas nsLSQR

El segundo punto a considerar es el desempeño de la implementación de la función `vjp`, cuyo mayor impacto es en las iteraciones internas del algoritmo. Es por esta razón que resulta interesante analizar su comportamiento dentro de estas. El siguiente experimento se diseñó con el propósito de medir el tiempo que utiliza el algoritmo nsLSQR en combinación con la aproximación $J\mathbf{v}$ y $\tilde{J}^T\mathbf{u}$ obtenidas a través de la diferenciación algorítmica. Esto permitirá observar cómo varía el tiempo de cómputo al emplear estas diferentes aproximaciones de la función.

Para calcular este tiempo, se empleará el módulo `time`, el cual forma parte de la biblioteca estándar de Python

Las dimensiones que se seleccionaron para cada una de las funciones son:

1. *Caso lineal*: $m = 10000$ y $n = 8000$
2. *Caso no lineal*: $m = 4000$ y $n = 1000$

La disminución de la dimensión del caso no lineal está asociada a la capacidad limitada de uso de colab.

El vector $\mathbf{x}$ utilizado para evaluar la matriz Jacobiana y su aproximación está distribuido uniformemente en el intervalo $[-1, 1]$, situación que se repite con el vector $\mathbf{w}$.

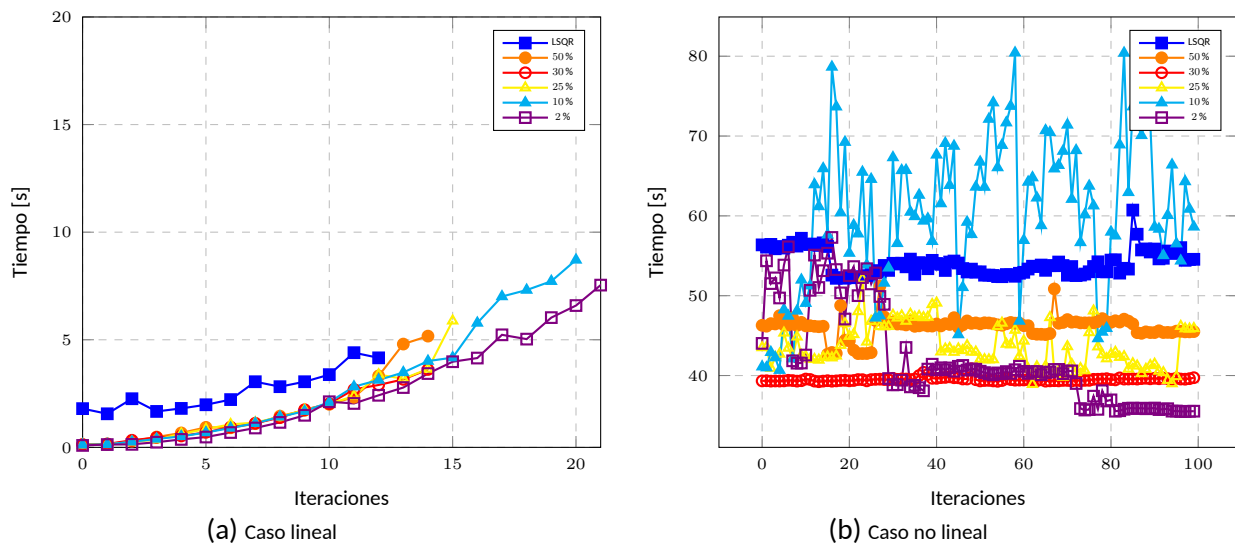


Figura 2: Variación del tiempo que demora en realizar el paso interno del algoritmo

En 2a se observa un aumento en el tiempo a medida que avanzan las iteraciones. En el caso de la función LSQR, los tiempos son inicialmente más altos desde la primera iteración, pero logra obtener un resultado en un menor número de iteraciones. Sin embargo, a lo largo del tiempo, el rendimiento de las aproximaciones 50 %, 30 %, 25 % y 10 %, aunque requieren un mayor número de iteraciones para obtener un resultado, se destaca por el menor tiempo promedio que toma en comparación con LSQR. En particular, la aproximación 30 % muestra un mejor

desempeño en términos de tiempo. Estas tendencias se pueden apreciar de manera gráfica en la Figura 3, donde los puntos por debajo de la línea punteada representan tiempos más cortos en comparación con nsLSQR, y entre ellos, destaca la mencionada aproximación.

Dentro del marco del caso no lineal, es interesante analizar cómo la elección de la aproximación afecta el tiempo requerido para cada iteración interna del algoritmo. Observamos que, en los casos anteriores al 25 % de aproximación, la curva que representa el tiempo de cálculo exhibe un comportamiento relativamente constante. Esto sugiere que, independientemente de la aproximación utilizada dentro de esta brecha, el algoritmo requiere un tiempo similar para cada paso iterativo.

Sin embargo, a medida que disminuye la precisión de la aproximación, se vuelve evidente un patrón más errático en los tiempos de cálculo. Este fenómeno podría deberse a que el algoritmo debe realizar ajustes más significativos en cada iteración cuando se trabaja con aproximaciones menos precisas, lo que conlleva un mayor tiempo de cómputo en ciertos casos.

Es relevante destacar que, al observar detenidamente la Figura 3, se destaca la aproximación del 30 %. Esta aproximación se caracteriza por mantener un tiempo de cálculo relativamente estable y, lo que es aún más notable, mantenerse por debajo de la línea punteada de referencia. Este resultado sugiere que, la aproximación del 30 % podría ser la opción atractiva.

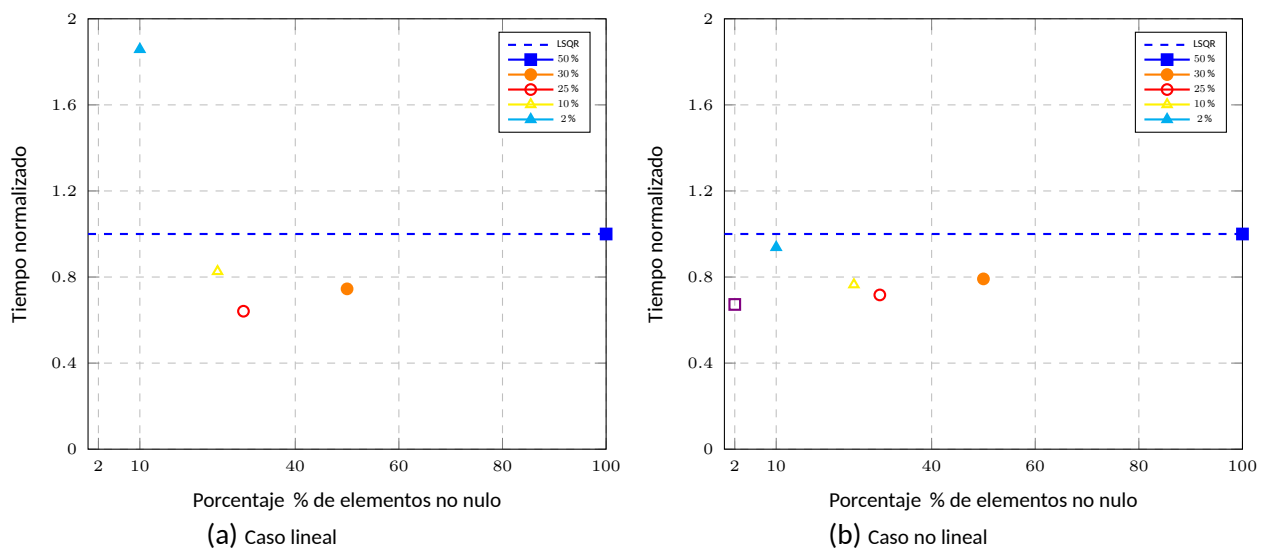


Figura 3: Variación del tiempo que demora en realizar el paso interno del algoritmo

4.3. Desempeño nsLSQR

La última sección de experimentos se centrará en abordar el aspecto que aún no hemos explorado: el rendimiento del algoritmo. Para lograr esto, es crucial tener en cuenta que evaluaremos su funcionamiento mediante la observación del “residuo relativo” y del “error relativo de la solución”. El primero dado por la expresión,

$$\frac{\|\mathbf{b} - J\mathbf{x}_k\|}{\|\mathbf{b}\|}$$

donde $\mathbf{b}$ es el valor de la función F en el punto de solución y $\mathbf{x}_k$ el valor obtenido en cada iteración.

Luego, se comparó la similitud de la solución a través del error relativo de la solución descrito en la siguiente expresión:

$$\frac{\|\mathbf{x}_{\text{sol}} - \mathbf{x}_k\|}{\|\mathbf{x}_{\text{sol}}\|}$$

donde $\mathbf{x}_k$ es el valor obtenido a través de la función en cada iteración, mientras que $\mathbf{x}_{\text{sol}}$ es la solución real del problema.

Nuevamente, la dimensiones que se seleccionaron para cada una de las funciones son:

1. *Caso lineal*: $m = 10000$ y $n = 8000$
2. *Caso no lineal*: $m = 4000$ y $n = 1000$

El vector $\mathbf{x}$ está nuevamente distribuido uniformemente en el intervalo $[-1, 1]$, situación que se repite con el vector $\mathbf{w}$.

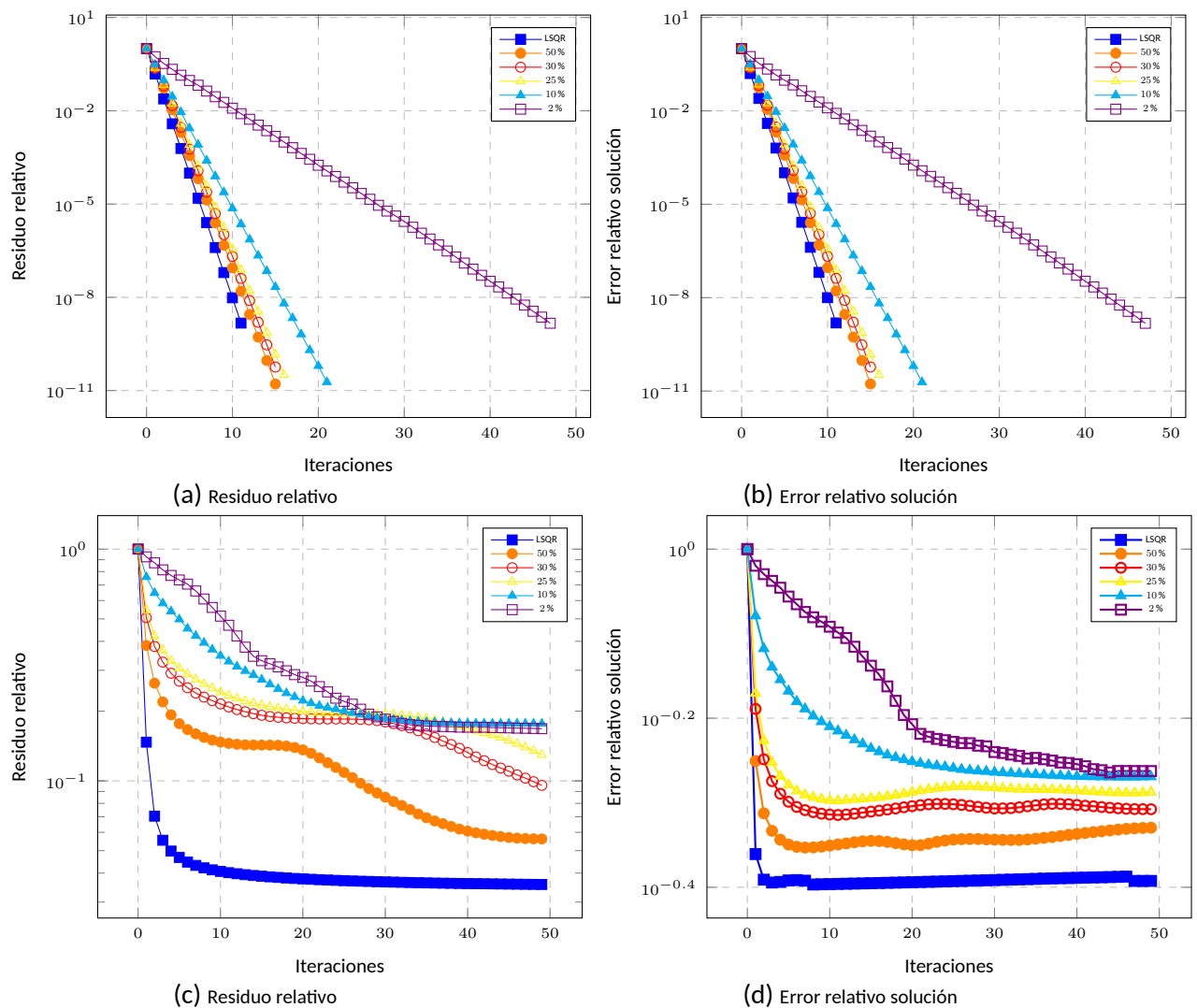


Figura 4: Residuo relativo y error relativo de la solución en función de las iteraciones internas

En este contexto, se puede observar que en el caso lineal, el algoritmo converge rápidamente hacia una solución a medida que avanza a través de las iteraciones internas. Sin embargo, en el caso no lineal, el algoritmo requiere trabajar a través de ambas iteraciones para reducir el residuo y mejorar la solución, pero esto resulta en un estancamiento en una solución óptima local, que puede no ser la que se está buscando. Esta diferencia podría atribuirse a la solución inicial entregada.

A continuación en la Figura 5, se presenta la influencia de la elección de la solución inicial en la convergencia del algoritmo. Partimos de la siguiente expresión para calcular la solución

inicial:

$$\mathbf{x}_0 = \mathbf{x}_i + e * \frac{\mathbf{r}}{\|\mathbf{r}\|}$$

Donde $\mathbf{x}_i$ representa la solución del problema, e es un entero, y $\mathbf{r}$ es un vector distribuido uniformemente. En estos experimentos, variamos el valor de e , y observamos cómo este valor afecta la velocidad de disminución del residuo. En general, cuanto menor es el valor de e , más rápido disminuye el residuo.

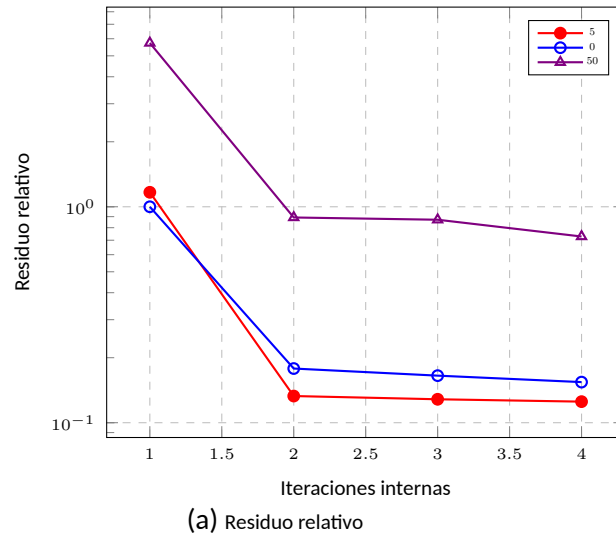
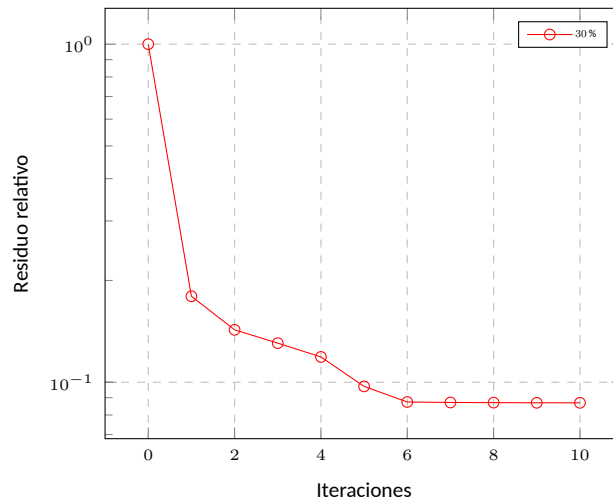


Figura 5: Residuo relativo dependiendo del valor de e

Cabe destacar que, en este caso, el valor 0 indica que se inicializó el algoritmo con el vector de 0, el cual está más lejano que el valor 5

Finalmente, utilizando solo en una aproximación (30 %) el algoritmo demuestra una marcada reducción en el residuo durante las primeras iteraciones, aunque esta mejora se vuelve gradual a medida que avanzamos. Sin embargo, es importante destacar que en ciertas iteraciones específicas se observa una disminución más sustancial del residuo, lo cual no descarta que esta tendencia se repita en iteraciones posteriores.



Junto a ello, se registra un peak de memoria en la operaciones

Operación	Memoria Peak [MiB]
$qu = J(V[:, k])$	4951.4
$qv = Jt(U[:, k+1])$	4673.6

Cuadro 7: Caption

Es importante tener presente que existen componentes de uso de memoria dentro del algoritmo que escapan a nuestro control, ya que son inherentes a su funcionamiento. Estos componentes incluyen las matrices V_k , U_k y $\tilde{B}_k$.

CAPÍTULO 5

CONCLUSIONES

Nuestra búsqueda de una metodología para obtener una aproximación de la matriz Jacobiana transpuesta en la implementación del algoritmo nsLSQR nos condujo a una profunda exploración de la diferenciación algorítmica. Esta técnica, que permite manejar de manera más eficiente las derivadas de funciones definidas, se convirtió en un pilar fundamental de nuestra investigación. Su aplicación, llevada a cabo mediante la biblioteca JAX, no solo corroboró la rapidez con la que se pueden calcular productos de matriz Jacobiana-vector (a través de JVP) y productos de vector-matriz Jacobiana (a través de VJP), sino que también enfatizó la importancia de este último en el contexto de nuestro trabajo, ya que es mediante esta operación que trabajamos con la matriz Jacobiana transpuesta.

Sin embargo, es imperativo destacar que la eficacia de esta técnica no solo depende de su implementación, sino también de la aproximación de la función subyacente, denotada como F . En particular, para esta técnica, es esencial tener acceso explícito a esta función para realizar modificaciones que conduzcan a una versión simplificada. Esto plantea una oportunidad de mejora, ya que sería deseable que el algoritmo fuera capaz de trabajar con la aproximación de la función de manera inmediata, sin requerir una implementación por separado.

Un aspecto adicional a considerar es el framework JAX, que se destaca por su robusta capacidad en cuanto a diferenciación automática, muestra ciertas limitaciones en otros aspectos, como el manejo de matrices dispersas (sparse matrices), que son esenciales debido a su eficiente almacenamiento comprimido. Aquí, se vislumbra un amplio potencial de mejora, ya que se pueden explorar diferentes enfoques al implementar la diferenciación algorítmica a través de otros frameworks o considerar opciones de compresión más sofisticadas.

Como pudimos observar a través de nuestros experimentos, la inclusión de la diferenciación algorítmica en el algoritmo nsLSQR no tuvo un impacto negativo en la convergencia del algo-

ritmo. Más bien, se evidenció que la convergencia está mayormente influida por la elección de la solución inicial y aproximación $\tilde{J}$. Esto adquiere una importancia particular dado que el método está diseñado para lograr una convergencia local, no necesariamente para encontrar el óptimo global que resuelva el problema de mínimos cuadrados, sino más bien para alcanzar una solución lo suficientemente precisa para satisfacer los requisitos planteados. En consecuencia, una elección inicial del vector muy alejada de la solución real puede resultar en un estancamiento en una solución alternativa que podría ser menos precisa que la solución esperada. Es importante destacar que, aunque nuestros experimentos han arrojado luz sobre este aspecto, no se realizaron pruebas más exhaustivas en torno a la elección de la solución inicial en esta etapa, pues esto escapa del estudio de esta memoria.

En resumen, la incorporación estratégica de la diferenciación algorítmica en la aproximación de derivadas no solo revela un inmenso potencial para la mejora de algoritmos empleados en una amplia variedad de problemas, sino que también emerge como una herramienta invaluable en disciplinas interdisciplinarias, como la inteligencia artificial. Su aplicación en diversos dominios, más allá de la resolución de problemas, demuestra cómo esta técnica puede influir positivamente en la optimización de procesos y en el avance de tecnologías relacionadas con la inteligencia artificial, donde el cálculo preciso de derivadas desempeña un papel esencial en la mejora del rendimiento y la eficiencia.

El debate constante en torno al equilibrio entre la gestión eficiente de la memoria y la reducción del tiempo de procesamiento no llega a su fin con esta investigación, sino que ofrece una perspectiva adicional que puede considerarse como una estrategia complementaria para abordar este desafío fundamental en la optimización de algoritmos y la mejora del rendimiento en cálculos computacionales.

ANEXOS

Pseudocódigos adicionales

A.1. Descomposición QR

Algorithm 3 Obtener la nueva descomposición QR

Require: La nueva columna b_k de $\tilde{B}_k$, la descomposición QR Q_{k-1} y R_{k-1} previa y la iteración actual k

Ensure: La nueva descomposición QR Q_k y R_k

- 1: Construir una matriz Q_k de tamaño $(k+1) \times k$ con coeficientes 0
 - 2: Construir una matriz R_k de tamaño $k \times k$ con coeficientes 0
 - 3: $Q_k[1:k, 1:k-1] := Q_{k-1}$
 - 4: $R_k[1:k-1, 1:k-1] := R_{k-1}$
 - 5: $y := b_k$
 - 6: **for** i desde 1 hasta $k-1$ **do**
 - 7: $r_{i,k} := \langle y, q_i \rangle$
 - 8: $y := y - r_{i,k} q_i$
 - 9: **end for**
 - 10: $r_{k,k} := \|y\|$
 - 11: $q_k = \frac{y}{r_{k,k}}$
 - 12: Retornar Q_k y R_k
-

A.2. Solución problema mínimos cuadrados

Algorithm 4 Algoritmo para resolver el problemas de mínimos cuadrados

Require: La nueva columna b_k de $\tilde{B}_k$, la descomposición QR Q_{k-1} y R_{k-1} previa, el escalar $\|b\|$ y la iteración actual k

Ensure: La nueva descomposición QR Q_k y R_k y la solución $\tilde{c}_k$

- 1: Usar el Algoritmo 3 para calcular la descomposición Q_k y R_k desde b_k , Q_{k-1} , R_{k-1} y k
 - 2: Obtener la primera fila de Q_k , denotado como q
 - 3: Resolver $R_k \tilde{c}_k = \|b\|q$ usando backward substitution
 - 4: Retornar x_k
-

Para más detalle de cómo se definen los criterios de parada y el background de los algoritmos anteriores, revisar [Sanhueza, 2021]

Reporte uso de memoria memoria función VJP

B.1. 2 %

Line #	Mem usage	Increment	Occurrences	Line Contents
=====				
2176	1656.4 MiB	1656.4 MiB	1	def vjp(# type: ignore
2177				fun: Callable, *primals, has_aux: bool = False, reduce_axes=()
2178				) -> tuple[Any, Callable] tuple[Any, Callable, Any]:
2179				"""Compute a (reverse-mode) vector-Jacobian product of ``fun``.
2180				
2181				:py:func:`grad` is implemented as a special case of :py:func:`vjp`.
2182				
2183				Args:
2184				fun: Function to be differentiated. Its arguments should be arrays, scalars,
2185				or standard Python containers of arrays or scalars. It should return an
2186				array, scalar, or standard Python container of arrays or scalars.
2187				primals: A sequence of primal values at which the Jacobian of ``fun``
2188				should be evaluated. The number of ``primals`` should be equal to the
2189				number of positional parameters of ``fun``. Each primal value should be
2190				an array, a scalar, or a pytree (standard Python containers) thereof.
2191				has_aux: Optional, bool. Indicates whether ``fun`` returns a pair where the
2192				first element is considered the output of the mathematical function to be
2193				differentiated and the second element is auxiliary data. Default False.
2194				reduce_axes: Optional, tuple of axis names. If an axis is listed here, and
2195				``fun`` implicitly broadcasts a value over that axis, the backward pass
2196				will perform a ``psum`` of the corresponding gradient. Otherwise, the
2197				VJP will be per-example over named axes. For example, if ``'batch'``
2198				is a named batch axis, ``vjp(f, *args, reduce_axes=('batch',))`` will
2199				create a VJP function that sums over the batch while ``vjp(f, *args)``
2200				will create a per-example VJP.
2201				
2202				Returns:

```

2203                                     If ``has_aux`` is ``False``, returns a ``(primals_out, vjpfun)`` pair, where
2204                                     ``primals_out`` is ``fun(*primals)``. If ``has_aux`` is ``True``, returns a
2205                                     ``(primals_out, vjpfun, aux)`` tuple where ``aux`` is the auxiliary data
2206                                     returned by ``fun``.
2207
2208                                     ``vjpfun`` is a function from a cotangent vector with the same shape as
2209                                     ``primals_out`` to a tuple of cotangent vectors with the same number and
2210                                     shapes as ``primals``, representing the vector-Jacobian product of ``fun``
2211                                     evaluated at ``primals``.
2212
2213                                     >>> import jax
2214                                     >>>
2215                                     >>> def f(x, y):
2216                                     ...     return jax.numpy.sin(x), jax.numpy.cos(y)
2217                                     ...
2218                                     >>> primals, f_vjp = jax.vjp(f, 0.5, 1.0)
2219                                     >>> xbar, ybar = f_vjp((-0.7, 0.3))
2220                                     >>> print(xbar)
2221                                     -0.61430776
2222                                     >>> print(ybar)
2223                                     -0.2524413
2224                                     """
2225     1656.4 MiB      0.0 MiB      1     check_callable(fun)
2226     1656.4 MiB      0.0 MiB      1     reduce_axes = _ensure_str_tuple(reduce_axes) # type: ignore
2227     1680.6 MiB     24.3 MiB      4     return _vjp(
2228     1656.4 MiB      0.0 MiB      3         lu.wrap_init(fun), *primals, has_aux=has_aux, reduce_axes=reduce_axes)

```

B.2. 10%

Line #	Mem usage	Increment	Occurrences	Line Contents
2176	1645.4 MiB	1645.4 MiB	1	def vjp(# type: ignore
2177				fun: Callable, *primals, has_aux: bool = False, reduce_axes=()
2178				) -> tuple[Any, Callable] tuple[Any, Callable, Any]:
2179				"""Compute a (reverse-mode) vector-Jacobian product of ``fun``.
2180				
2181				:py:func:`grad` is implemented as a special case of :py:func:`vjp`.
2182				
2183				Args:
2184				fun: Function to be differentiated. Its arguments should be arrays, scalars,
2185				or standard Python containers of arrays or scalars. It should return an
2186				array, scalar, or standard Python container of arrays or scalars.
2187				primals: A sequence of primal values at which the Jacobian of ``fun``
2188				should be evaluated. The number of ``primals`` should be equal to the
2189				number of positional parameters of ``fun``. Each primal value should be
2190				an array, a scalar, or a pytree (standard Python containers) thereof.
2191				has_aux: Optional, bool. Indicates whether ``fun`` returns a pair where the
2192				first element is considered the output of the mathematical function to be
2193				differentiated and the second element is auxiliary data. Default False.
2194				reduce_axes: Optional, tuple of axis names. If an axis is listed here, and
2195				``fun`` implicitly broadcasts a value over that axis, the backward pass
2196				will perform a ``psum`` of the corresponding gradient. Otherwise, the
2197				VJP will be per-example over named axes. For example, if ``'batch'``
2198				is a named batch axis, ``vjp(f, *args, reduce_axes=('batch',))`` will
2199				create a VJP function that sums over the batch while ``vjp(f, *args)``
2200				will create a per-example VJP.
2201				
2202				Returns:
2203				If ``has_aux`` is ``False``, returns a ``(primals_out, vjpfun)`` pair, where
2204				``primals_out`` is ``fun(*primals)``. If ``has_aux`` is ``True``, returns a
2205				``(primals_out, vjpfun, aux)`` tuple where ``aux`` is the auxiliary data
2206				returned by ``fun``.
2207				
2208				``vjpfun`` is a function from a cotangent vector with the same shape as

```

2209         ``primals_out`` to a tuple of cotangent vectors with the same number and
2210         shapes as ``primals``, representing the vector-Jacobian product of ``fun``
2211         evaluated at ``primals``.
2212
2213     >>> import jax
2214     >>>
2215     >>> def f(x, y):
2216     ...     return jax.numpy.sin(x), jax.numpy.cos(y)
2217     ...
2218     >>> primals, f_vjp = jax.vjp(f, 0.5, 1.0)
2219     >>> xbar, ybar = f_vjp((-0.7, 0.3))
2220     >>> print(xbar)
2221     -0.61430776
2222     >>> print(ybar)
2223     -0.2524413
2224     """
2225     1645.4 MiB      0.0 MiB      1   check_callable(fun)
2226     1645.4 MiB      0.0 MiB      1   reduce_axes = _ensure_str_tuple(reduce_axes) # type: ignore
2227     1756.6 MiB     111.2 MiB      4   return _vjp(
2228     1645.4 MiB      0.0 MiB      3       lu.wrap_init(fun), *primals, has_aux=has_aux, reduce_axes=reduce_axes)

```

B.3. 25%

Line #	Mem usage	Increment	Occurrences	Line Contents
2176	1657.4 MiB	1657.4 MiB	1	def vjp(# type: ignore
2177				fun: Callable, *primals, has_aux: bool = False, reduce_axes=()
2178				) -> tuple[Any, Callable] tuple[Any, Callable, Any]:
2179				"""Compute a (reverse-mode) vector-Jacobian product of ``fun``.
2180				
2181				:py:func:`grad` is implemented as a special case of :py:func:`vjp`.
2182				
2183				Args:
2184				fun: Function to be differentiated. Its arguments should be arrays, scalars,
2185				or standard Python containers of arrays or scalars. It should return an
2186				array, scalar, or standard Python container of arrays or scalars.
2187				primals: A sequence of primal values at which the Jacobian of ``fun``
2188				should be evaluated. The number of ``primals`` should be equal to the
2189				number of positional parameters of ``fun``. Each primal value should be
2190				an array, a scalar, or a pytree (standard Python containers) thereof.
2191				has_aux: Optional, bool. Indicates whether ``fun`` returns a pair where the
2192				first element is considered the output of the mathematical function to be
2193				differentiated and the second element is auxiliary data. Default False.
2194				reduce_axes: Optional, tuple of axis names. If an axis is listed here, and
2195				``fun`` implicitly broadcasts a value over that axis, the backward pass
2196				will perform a ``psum`` of the corresponding gradient. Otherwise, the
2197				VJP will be per-example over named axes. For example, if ``'batch'``
2198				is a named batch axis, ``vjp(f, *args, reduce_axes=('batch',))`` will
2199				create a VJP function that sums over the batch while ``vjp(f, *args)``
2200				will create a per-example VJP.
2201				
2202				Returns:
2203				If ``has_aux`` is ``False``, returns a ``(primals_out, vjpfun)`` pair, where
2204				``primals_out`` is ``fun(*primals)``. If ``has_aux`` is ``True``, returns a
2205				``(primals_out, vjpfun, aux)`` tuple where ``aux`` is the auxiliary data
2206				returned by ``fun``.
2207				
2208				``vjpfun`` is a function from a cotangent vector with the same shape as
2209				``primals_out`` to a tuple of cotangent vectors with the same number and
2210				shapes as ``primals``, representing the vector-Jacobian product of ``fun``
2211				evaluated at ``primals``.
2212				
2213				>>> import jax
2214				>>>

```

2215                                     >>> def f(x, y):
2216                                     ...     return jax.numpy.sin(x), jax.numpy.cos(y)
2217                                     ...
2218                                     >>> primals, f_vjp = jax.vjp(f, 0.5, 1.0)
2219                                     >>> xbar, ybar = f_vjp((-0.7, 0.3))
2220                                     >>> print(xbar)
2221                                     -0.61430776
2222                                     >>> print(ybar)
2223                                     -0.2524413
2224                                     """
2225 1657.4 MiB      0.0 MiB      1      check_callable(fun)
2226 1657.4 MiB      0.0 MiB      1      reduce_axes = _ensure_str_tuple(reduce_axes) # type: ignore
2227 1900.1 MiB     242.7 MiB      4      return _vjp(
2228 1657.4 MiB      0.0 MiB      3          lu.wrap_init(fun), *primals, has_aux=has_aux, reduce_axes=reduce_axes)

```

B.4. 30%

Line #	Mem usage	Increment	Occurrences	Line Contents
2176	1655.7 MiB	1655.7 MiB	1	def vjp(# type: ignore
2177				fun: Callable, *primals, has_aux: bool = False, reduce_axes=()
2178				) -> tuple[Any, Callable] tuple[Any, Callable, Any]:
2179				"""Compute a (reverse-mode) vector-Jacobian product of ``fun``.
2180				
2181				:py:func:`grad` is implemented as a special case of :py:func:`vjp`.
2182				
2183				Args:
2184				fun: Function to be differentiated. Its arguments should be arrays, scalars,
2185				or standard Python containers of arrays or scalars. It should return an
2186				array, scalar, or standard Python container of arrays or scalars.
2187				primals: A sequence of primal values at which the Jacobian of ``fun``
2188				should be evaluated. The number of ``primals`` should be equal to the
2189				number of positional parameters of ``fun``. Each primal value should be
2190				an array, a scalar, or a pytree (standard Python containers) thereof.
2191				has_aux: Optional, bool. Indicates whether ``fun`` returns a pair where the
2192				first element is considered the output of the mathematical function to be
2193				differentiated and the second element is auxiliary data. Default False.
2194				reduce_axes: Optional, tuple of axis names. If an axis is listed here, and
2195				``fun`` implicitly broadcasts a value over that axis, the backward pass
2196				will perform a ``psum`` of the corresponding gradient. Otherwise, the
2197				VJP will be per-example over named axes. For example, if ``'batch'``
2198				is a named batch axis, ``vjp(f, *args, reduce_axes=('batch',))`` will
2199				create a VJP function that sums over the batch while ``vjp(f, *args)``
2200				will create a per-example VJP.
2201				
2202				Returns:
2203				If ``has_aux`` is ``False``, returns a ``(primals_out, vjpfun)`` pair, where
2204				``primals_out`` is ``fun(*primals)``. If ``has_aux`` is ``True``, returns a
2205				``(primals_out, vjpfun, aux)`` tuple where ``aux`` is the auxiliary data
2206				returned by ``fun``.
2207				
2208				``vjpfun`` is a function from a cotangent vector with the same shape as
2209				``primals_out`` to a tuple of cotangent vectors with the same number and
2210				shapes as ``primals``, representing the vector-Jacobian product of ``fun``
2211				evaluated at ``primals``.
2212				
2213				>>> import jax
2214				>>>
2215				>>> def f(x, y):
2216				... return jax.numpy.sin(x), jax.numpy.cos(y)
2217				...
2218				>>> primals, f_vjp = jax.vjp(f, 0.5, 1.0)
2219				>>> xbar, ybar = f_vjp((-0.7, 0.3))
2220				>>> print(xbar)

```

2221                                     -0.61430776
2222                                     >>> print(ybar)
2223                                     -0.2524413
2224                                     """
2225 1655.7 MiB      0.0 MiB      1      check_callable(fun)
2226 1655.7 MiB      0.0 MiB      1      reduce_axes = _ensure_str_tuple(reduce_axes) # type: ignore
2227 1948.5 MiB      292.8 MiB     4      return _vjp(
2228 1655.7 MiB      0.0 MiB      3          lu.wrap_init(fun), *primals, has_aux=has_aux, reduce_axes=reduce_axes)

```

B.5. 50%

Line #	Mem usage	Increment	Occurrences	Line Contents
2176	1644.0 MiB	1644.0 MiB	1	def vjp(# type: ignore
2177				fun: Callable, *primals, has_aux: bool = False, reduce_axes=()
2178				) -> tuple[Any, Callable] tuple[Any, Callable, Any]:
2179				"""Compute a (reverse-mode) vector-Jacobian product of ``fun``.
2180				
2181				:py:func:`grad` is implemented as a special case of :py:func:`vjp`.
2182				
2183				Args:
2184				fun: Function to be differentiated. Its arguments should be arrays, scalars,
2185				or standard Python containers of arrays or scalars. It should return an
2186				array, scalar, or standard Python container of arrays or scalars.
2187				primals: A sequence of primal values at which the Jacobian of ``fun``
2188				should be evaluated. The number of ``primals`` should be equal to the
2189				number of positional parameters of ``fun``. Each primal value should be
2190				an array, a scalar, or a pytree (standard Python containers) thereof.
2191				has_aux: Optional, bool. Indicates whether ``fun`` returns a pair where the
2192				first element is considered the output of the mathematical function to be
2193				differentiated and the second element is auxiliary data. Default False.
2194				reduce_axes: Optional, tuple of axis names. If an axis is listed here, and
2195				``fun`` implicitly broadcasts a value over that axis, the backward pass
2196				will perform a ``psum`` of the corresponding gradient. Otherwise, the
2197				VJP will be per-example over named axes. For example, if ``'batch'``
2198				is a named batch axis, ``vjp(f, *args, reduce_axes=('batch',))`` will
2199				create a VJP function that sums over the batch while ``vjp(f, *args)``
2200				will create a per-example VJP.
2201				
2202				Returns:
2203				If ``has_aux`` is ``False``, returns a ``(primals_out, vjpfun)`` pair, where
2204				``primals_out`` is ``fun(*primals)``. If ``has_aux`` is ``True``, returns a
2205				``(primals_out, vjpfun, aux)`` tuple where ``aux`` is the auxiliary data
2206				returned by ``fun``.
2207				
2208				``vjpfun`` is a function from a cotangent vector with the same shape as
2209				``primals_out`` to a tuple of cotangent vectors with the same number and
2210				shapes as ``primals``, representing the vector-Jacobian product of ``fun``
2211				evaluated at ``primals``.
2212				
2213				>>> import jax
2214				>>>
2215				>>> def f(x, y):
2216				... return jax.numpy.sin(x), jax.numpy.cos(y)
2217				...
2218				>>> primals, f_vjp = jax.vjp(f, 0.5, 1.0)
2219				>>> xbar, ybar = f_vjp((-0.7, 0.3))
2220				>>> print(xbar)
2221				-0.61430776
2222				>>> print(ybar)
2223				-0.2524413
2224				"""
2225	1644.0 MiB	0.0 MiB	1	check_callable(fun)
2226	1644.0 MiB	0.0 MiB	1	reduce_axes = _ensure_str_tuple(reduce_axes) # type: ignore

```

2227 2148.2 MiB  504.2 MiB      4  return _vjp(
2228 1644.0 MiB   0.0 MiB      3      lu.wrap_init(fun), *primals, has_aux=has_aux, reduce_axes=reduce_axes)

```

B.6. 100%

Line #	Mem usage	Increment	Occurrences	Line Contents
2176	1657.7 MiB	1657.7 MiB	1	def vjp(# type: ignore
2177				fun: Callable, *primals, has_aux: bool = False, reduce_axes=()
2178				) -> tuple[Any, Callable] tuple[Any, Callable, Any]:
2179				"""Compute a (reverse-mode) vector-Jacobian product of ``fun``.
2180				
2181				:py:func:`grad` is implemented as a special case of :py:func:`vjp`.
2182				
2183				Args:
2184				fun: Function to be differentiated. Its arguments should be arrays, scalars,
2185				or standard Python containers of arrays or scalars. It should return an
2186				array, scalar, or standard Python container of arrays or scalars.
2187				primals: A sequence of primal values at which the Jacobian of ``fun``
2188				should be evaluated. The number of ``primals`` should be equal to the
2189				number of positional parameters of ``fun``. Each primal value should be
2190				an array, a scalar, or a pytree (standard Python containers) thereof.
2191				has_aux: Optional, bool. Indicates whether ``fun`` returns a pair where the
2192				first element is considered the output of the mathematical function to be
2193				differentiated and the second element is auxiliary data. Default False.
2194				reduce_axes: Optional, tuple of axis names. If an axis is listed here, and
2195				``fun`` implicitly broadcasts a value over that axis, the backward pass
2196				will perform a ``psum`` of the corresponding gradient. Otherwise, the
2197				VJP will be per-example over named axes. For example, if ``'batch'``
2198				is a named batch axis, ``vjp(f, *args, reduce_axes=('batch',))`` will
2199				create a VJP function that sums over the batch while ``vjp(f, *args)``
2200				will create a per-example VJP.
2201				
2202				Returns:
2203				If ``has_aux`` is ``False``, returns a ``(primals_out, vjpfun)`` pair, where
2204				``primals_out`` is ``fun(*primals)``. If ``has_aux`` is ``True``, returns a
2205				``(primals_out, vjpfun, aux)`` tuple where ``aux`` is the auxiliary data
2206				returned by ``fun``.
2207				
2208				``vjpfun`` is a function from a cotangent vector with the same shape as
2209				``primals_out`` to a tuple of cotangent vectors with the same number and
2210				shapes as ``primals``, representing the vector-Jacobian product of ``fun``
2211				evaluated at ``primals``.
2212				
2213				>>> import jax
2214				>>>
2215				>>> def f(x, y):
2216				... return jax.numpy.sin(x), jax.numpy.cos(y)
2217				...
2218				>>> primals, f_vjp = jax.vjp(f, 0.5, 1.0)
2219				>>> xbar, ybar = f_vjp((-0.7, 0.3))
2220				>>> print(xbar)
2221				-0.61430776
2222				>>> print(ybar)
2223				-0.2524413
2224				"""
2225	1657.7 MiB	0.0 MiB	1	check_callable(fun)
2226	1657.7 MiB	0.0 MiB	1	reduce_axes = _ensure_str_tuple(reduce_axes) # type: ignore
2227	2610.7 MiB	953.0 MiB	4	return _vjp(
2228	1657.7 MiB	0.0 MiB	3	lu.wrap_init(fun), *primals, has_aux=has_aux, reduce_axes=reduce_axes)

Reporte uso de memoria nsLSQR con Diferenciación Automática

Line #	Mem usage	Increment	Occurrences	Line Contents
95	364.0 MiB	364.0 MiB	1	def nsLSQR(J, Jt, b, x0, rtol = 1e-8, dtol = 1e-5, d = 5, stol = 1.0, s = 5, tin = 50, tout = 1, show = False):
96				'''
97				Args:
98				
99				J : función que retorna el producto Jv
100				Jt : función que retorna el producto J ^t w
101				b : vector de largo m resultado del sistema Ax = b
102				x0 : solución inicial sobre la cual se comienza a iterar
103				rtol : tolerancia de la solución
104				dtol : tolerancia de la diversidad
105				d : entero que controla la diversidad
106				stol : tolerancia de la pendiente
107				s : entero que controla la pendiente
108				tin : número de iteraciones antes de un restart
109				tout : número de iteraciones para buscar una solución
110				show : booleano que indica la impresión de valores
111				
112				Returns:
113				x : aproximación de la solución del sistema de ecuación Ax = b
114				rnorm : norma b - Jxk
115				x_list: lista de valores anteriores
116				'''
117				
118				# Tamaño del problema
119	364.0 MiB	0.0 MiB	1	m, n = len(b), len(x0)
120				
121				# Contador no progreso
122	364.0 MiB	0.0 MiB	1	sin_progreso = 0
123				
124				# Flag para el término del algoritmo
125	364.0 MiB	0.0 MiB	1	flag = False
126				
127				# Listas para almacenar valores a retornar
128	364.0 MiB	0.0 MiB	1	p_list = [] #pendientes
129	364.0 MiB	0.0 MiB	1	t_list = [] #tiempo
130	364.0 MiB	0.0 MiB	1	x_list = [] #valores x
131	364.0 MiB	0.0 MiB	1	r_list = [] #residuos
132				
133	364.0 MiB	0.0 MiB	1	rnorm = jnp.zeros(tin)
134				
135				# Iteración de re-start
136	364.1 MiB	0.1 MiB	2	for i in jnp.arange(tout):
137				
138	364.1 MiB	0.0 MiB	1	if show:
139				titulo = ' ----- nsLSQR ----- '
140				info = f'x0 = {x0[0]} norma = {jnp.linalg.norm(b - J(x0))/jnp.linalg.norm(b):10.3e}'
141				encabezado = ' k x rlnorm '
142				print(titulo)
143				print(info)
144				print(" ")
145				print(encabezado)
146				
147	364.1 MiB	0.0 MiB	1	rnorm = jnp.zeros(tin)
148				
149				# El problema que tenemos es F(xi) - J(xi)*xk
150				# Calculamos el residuo inicial con x0
151	368.9 MiB	4.8 MiB	1	r0 = b - J(x0)
152	369.8 MiB	0.9 MiB	1	r0_norm = jnp.linalg.norm(r0)
153	369.9 MiB	0.1 MiB	1	r_list.append(r0_norm/jnp.linalg.norm(b))
154	370.2 MiB	0.4 MiB	1	rnorm = rnorm.at[0].set(r0_norm/r0_norm)
155				
156				#almacenamos la primera solución
157	370.2 MiB	0.0 MiB	1	x_list.append(x0)

```

158 370.2 MiB 0.0 MiB 1 xk = jnp.zeros(n)
159
160 # Construimos las matrices U y V de LSQR, con tin columnas
161 # Construimos las matrices Bt y Br para las recurrencias
162
163 371.7 MiB 1.5 MiB 1 U = jnp.zeros((m, tin+1))
164 372.2 MiB 0.5 MiB 1 V = jnp.zeros((n, tin+1))
165
166 372.3 MiB 0.1 MiB 1 Bt = jnp.zeros((tin+1, tin))
167 372.3 MiB 0.1 MiB 1 Br = jnp.zeros((tin, tin+1))
168
169 377.8 MiB 5.4 MiB 1 hat_r0 = Jt(r0)
170 378.4 MiB 0.7 MiB 1 hat_r0_norm = jnp.linalg.norm(hat_r0)
171
172 # Inicializamos vectores u1 y v1
173 # u1 = r0/||r0||_2
174 # v1 = (J.T r0)/||(J.T r0)||_2
175 380.4 MiB 1.9 MiB 1 U = U.at[:,0].set(r0/r0_norm)
176 381.1 MiB 0.7 MiB 1 V = V.at[:,0].set(hat_r0/hat_r0_norm)
177
178
179 #Inicializamos Br
180 #Este paso no es realmente necesario porque podemos construir esta matriz sobre la marcha
181 381.2 MiB 0.1 MiB 1 Br = Br.at[0,0].set(hat_r0_norm/r0_norm)
182
183 416.2 MiB -4902.8 MiB 31 for k in jnp.arange(tin):
184 # Ahora calculamos Gram-Schmidt modificado
185 '''
186 Primera recursión Jvk = Uk+1 Bk (idem a Aqk = Qk+1 Hk) a través de Gram-Schmidt modificado
187 '''
188 416.2 MiB -4951.4 MiB 30 qu = J(V[:, k])
189
190 416.2 MiB -86228.7 MiB 495 for j in jnp.arange(k+1):
191 416.2 MiB -81333.0 MiB 465 Bt = Bt.at[j, k].set(jnp.dot(qu, U[:, j]))
192 416.2 MiB -81334.4 MiB 465 qu = qu - Bt[j, k]*U[:, j]
193 416.2 MiB -4891.6 MiB 30 c = jnp.linalg.norm(qu)
194
195 416.2 MiB -4890.4 MiB 30 if not jnp.allclose(c, 0):
196 # Almacenamos el U obtenido
197 416.2 MiB -4891.4 MiB 30 Bt = Bt.at[k+1,k].set(c)
198 # Si quedan iteraciones
199 416.2 MiB -4891.4 MiB 30 if k < n-1:
200 417.0 MiB -4863.0 MiB 30 U = U.at[:,k+1].set(qu/c)
201
202 '''
203 Ahora debemos trabajar en la segunda recursión Juk+1 = Vk+1 Bk+1
204 '''
205
206 # Aquí construimos V
207 417.1 MiB -4673.6 MiB 30 qv = Jt(U[:, k+1])
208
209 417.1 MiB -83934.2 MiB 495 for j in jnp.arange(k+1):
210 417.1 MiB -79257.3 MiB 465 Br = Br.at[j, k+1].set(jnp.dot(qv, V[:, j]))
211 417.1 MiB -79257.1 MiB 465 qv = qv - Br[j, k+1]*V[:, j]
212 417.1 MiB -4675.4 MiB 30 c = jnp.linalg.norm(qv)
213
214 # Si c es muy pequeño, podría ser un problema
215 417.1 MiB -4675.4 MiB 30 if not jnp.allclose(c, 0):
216 417.1 MiB -4675.4 MiB 30 Br = Br.at[k+1, k+1].set(jnp.linalg.norm(qv))
217 417.1 MiB -4672.0 MiB 30 V = V.at[:, k+1].set(qv/c)
218
219 else:
220 flag = True
221
222 #Ahora, resolvemos el problema de mínimos cuadrados a través de la descomposición QR propuesta
223 417.1 MiB -4673.0 MiB 30 if k == 0:
224 390.4 MiB 1.8 MiB 1 R = jnp.array([jnp.linalg.norm(Bt[:,0])])
225 391.5 MiB 1.0 MiB 1 Q = jnp.array(Bt[:,2, 0]/R)
226 391.9 MiB 0.5 MiB 1 aux = r0_norm*Q[0]
227 392.4 MiB 0.4 MiB 1 ck = aux/R

```

```

228
229
230 421.9 MiB -4528.9 MiB 29
231
232
233 421.9 MiB -4666.0 MiB 30
234 422.2 MiB -4644.2 MiB 30
235 422.2 MiB -4654.4 MiB 30
236
237
238 416.2 MiB -5111.6 MiB 30
239
240 416.2 MiB -4954.6 MiB 30
241 416.2 MiB -4954.4 MiB 30
242
243 416.2 MiB -4954.4 MiB 30
244
245
246
247
248 416.2 MiB -4953.6 MiB 30
249 250.2 MiB -166.0 MiB 1
250
251
252 416.2 MiB -4787.9 MiB 29
253
254
255 416.2 MiB -4787.9 MiB 30
256 416.2 MiB -4953.8 MiB 30
257 286.6 MiB -4777.7 MiB 25
258 286.6 MiB -159.5 MiB 25
259
260 416.2 MiB -286.8 MiB 30
261
262
263
264
265
266
267
268
269
270
271 223.4 MiB -192.8 MiB 1
272
273 223.5 MiB -140.6 MiB 1

```

```

else:
    Q, R, ck = LS_solver(Bt[:k+2,k], Q, R, r0_norm, k)

# Guardamos la solución anterior y calculamos la nueva xk
prev_x = xk.copy()
xk = jnp.dot(V[:, :k+1], ck)
x_list.append(x0 + xk)

# Calculamos el residuo como ||b - j*xk||_2/||r0_norm||
xk_norm = jnp.linalg.norm(r0 - J(xk))
rnorm = rnorm.at[k+1].set(xk_norm)
r_list.append(xk_norm/r0_norm)

if show:
    str1 = f'{k:6g} {xk[0]:12.5e} {jnp.linalg.norm(r0 - J(xk))/r0_norm:10.3e}'
    print(str1)

# Test progreso de la solución
if jnp.linalg.norm(xk - prev_x)/jnp.linalg.norm(xk) <= dtol:
    sin_progreso += 1
else:
    sin_progreso = 0

# Test de la pendiente
pendiente = 0
if k >= s:
    pendiente = regresion_lineal(rnorm[k-s:k+1], np.arange(k-s,k+1))
    p_list.append(pendiente)

if pendiente > stol or (r_list[-1]) < rtol or sin_progreso == d or flag:
    if show:
        print("")
        print("Solución encontrada x = {}".format((x0 + xk)[0]))
        print("")
        print("Criterio pendiente :",pendiente > stol, pendiente)
        print("Criterio norma :",r_list[-1] <= rtol)
        print("Criterio diversidad :", sin_progreso == d)
        print("Criterio cero :",flag)
    return xk, r_list, x_list

x0 = x0 + xk

return x0, r_list, x_list

```

REFERENCIAS BIBLIOGRÁFICAS

- [Armijo, 1966] Armijo, L. (1966). Minimization of functions having lipschitz continuous first partial derivatives. *Pacific Journal of mathematics*, 16(1):1–3.
- [Barzilai y Borwein, 1988] Barzilai, J. y Borwein, J. M. (1988). Two-point step size gradient methods. *IMA journal of numerical analysis*, 8(1):141–148.
- [Birgin et al., 2003] Birgin, E. G., Krejić, N., y Martínez, J. M. (2003). Globally convergent inexact quasi-newton methods for solving nonlinear systems. *Numerical algorithms*, 32(2):249–260.
- [Birgin et al., 2014] Birgin, E. G., Martínez, J. M., y Raydan, M. (2014). Spectral projected gradient methods: review and perspectives. *Journal of Statistical Software*, 60(1):1–21.
- [Cheng et al., 2009] Cheng, W., Xiao, Y., y Hu, Q.-J. (2009). A family of derivative-free conjugate gradient methods for large-scale nonlinear systems of equations. *Journal of Computational and Applied Mathematics*, 224(1):11–19.
- [Chong y Zak, 2004] Chong, E. K. y Zak, S. H. (2004). *An introduction to optimization*. John Wiley & Sons.
- [Dembo et al., 1982] Dembo, R. S., Eisenstat, S. C., y Steihaug, T. (1982). Inexact newton methods. *SIAM Journal on Numerical analysis*, 19(2):400–408.
- [Dennis y Moré, 1977] Dennis, Jr, J. E. y Moré, J. J. (1977). Quasi-newton methods, motivation and theory. *SIAM review*, 19(1):46–89.
- [Golub y Kahan, 1965] Golub, G. y Kahan, W. (1965). Calculating the singular values and pseudo-inverse of a matrix. *Journal of the Society for Industrial and Applied Mathematics, Series B: Numerical Analysis*, 2(2):205–224.
- [Griewank y Walther, 2008] Griewank, A. y Walther, A. (2008). *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM.

- [Hager y Zhang, 2006] Hager, W. W. y Zhang, H. (2006). A survey of nonlinear conjugate gradient methods. *Pacific journal of Optimization*, 2(1):35–58.
- [Hestenes y Stiefel, 1952] Hestenes, M. R. y Stiefel, Eduard, e. a. (1952). Methods of conjugate gradients for solving linear systems. *Journal of research of the National Bureau of Standards*, 49(6):409–436.
- [Johnson y Levskaya, 2023] Johnson, M. y Levskaya, A. (2023). Jax quickstart.
- [Knoll y Keyes, 2004] Knoll, D. A. y Keyes, D. E. (2004). Jacobian-free newton–krylov methods: a survey of approaches and applications. *Journal of Computational Physics*, 193(2).
- [Moré, 2006] Moré, J. J. (2006). The levenberg-marquardt algorithm: implementation and theory. En *Numerical analysis: proceedings of the biennial Conference held at Dundee, June 28–July 1, 1977*, pp. 105–116. Springer.
- [Nocedal y Wright, 2006] Nocedal, J. y Wright, S. (2006). *Numerical optimization*. Springer Science & Business Media.
- [O’Connor, 2023] O’Connor, R. (2023). Why you should (or shouldn’t) be using google’s jax in 2023.
- [Paige y Saunders, 1982] Paige, C. C. y Saunders, M. A. (1982). Lsq: An algorithm for sparse linear equations and sparse least squares. *ACM Transactions on Mathematical Software (TOMS)*, 8(1):43–71.
- [Raydan, 1993] Raydan, M. (1993). On the barzilai and borwein choice of steplength for the gradient method. *IMA Journal of Numerical Analysis*, 13(3):321–326.
- [Raydan, 1997] Raydan, M. (1997). The barzilai and borwein gradient method for the large scale unconstrained minimization problem. *SIAM Journal on Optimization*, 7(1):26–33.
- [Sanhueza, 2021] Sanhueza, A. (2021). A limited-memory levenberg-marquardt algorithm for solving large-scale nonlinear least-square problems. Tesis de máster, Universidad Técnica Federico Santa María.
- [Sanhueza y Torres, 2017] Sanhueza, A. y Torres, C. E. (2017). A matrix-free algorithm based on newton’s method for overdetermined nonlinear system of equations. En *2017 36th International Conference of the Chilean Computer Science Society (SCCC)*. IEEE.

[Sauer, 2011] Sauer, T. (2011). *Numerical Analysis*. Addison-Wesley Publishing Company, USA, 2nd edición.

[Zhang y Zhou, 2006] Zhang, L. y Zhou, W. (2006). Spectral gradient projection method for solving nonlinear monotone equations. *Journal of Computational and Applied Mathematics*, 196(2):478–484.