UNIVERSIDAD POLITECNICA DE VALENCIA

DSIC
DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Trabajo Final de Máster

Cálculo de la exponencial de una matriz en SLEPc

Máster en Computación Paralela y Distribuida

Autor: Javier Nadal Durá
Director: José E. Román Moltó

Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia

2 de julio de 2013

Índice general

Introducción

En muchas aplicaciones, aparece la necesidad de calcular para una matriz A una cierta función $f(A)$, como puede ser la exponencial, el logaritmo, la raíz cuadrada, la función signo u otras. En algunas de estas aplicaciones, especialmente en el contexto de matrices dispersas de gran dimensión, no es necesario obtener de forma explícita la matriz $f(A)$ sino que se requiere únicamente el cálculo de $f(A)v$ para un cierto vector v. En este caso, es posible plantear un enfoque del problema basado en proyecciones sobre subespacios de Krylov, con lo que los algoritmos de Arnoldi y Lanczos constituyen el principal núcleo computacional. En particular, el cálculo de $\exp(tA)v$, para un cierto escalar t, es la base de algunos métodos para resolver ecuaciones diferenciales ordinarias que gobiernan el comportamiento de muchos fenómenos físicos. El software EXPOKIT implementa este cálculo de forma secuencial, y no existe ningún software paralelo que ofrezca una funcionalidad similar. El objetivo de este trabajo final de máster consiste en implementar una nueva clase de métodos en SLEPc que permitan realizar dicho cálculo, aprovechando los métodos de Krylov de que dispone.

El presente documento está estructurado tal y como se indica a continuación: en el capítulo 1 se resumen aspectos claves sobre la teoría de las funciones matriciales y se describen algunos métodos de resolución. El capítulo 2 presenta el algoritmo empleado por EXPOKIT para el cálculo de la función exponencial. El capítulo 3 realiza una breve introducción a las librerías PETSc y SLEPc. En el capítulo 4 se muestran los detalles más relevantes de la implementación en SLEPc.

Después, la descripción de los experimentos que comprueban la funcionalidad y miden las prestaciones de la implementación, son recogidos en el capítulo 5 junto a los resultados.

Por último, se encuentran las conclusiones del trabajo en el capíitulo 6 y al final se adjunta tanto la bibliografía referida durante el documento como las referencias más importantes consultadas durante su elaboración.

Capítulo 1

Funciones matriciales

La necesidad de evaluar una función $f(A) \in \mathbb{C}^{n\times n}$ de una matriz $A \in \mathbb{C}^{n\times n}$ surge en un amplio y creciente número de aplicaciones, que van desde la solución numérica de ecuaciones diferenciales a la medida de la complejidad de las redes. En este capítulo se estudian métodos numéricos para la evaluación de las funciones de una matriz, junto con una breve descripción de la teoría subyacente.

1.1. Definición y propiedades

Se entiende por función matricial aquella función $f : \mathbb{C}^{n\times n} \mapsto \mathbb{C}^{n\times n}$ que se define en términos de una función escalar. De una manera informal, dada una función $f(z)$ definida sobre el espectro de una matriz cuadrada A, la matriz $f(A)$ se puede obtener sustituyendo la variable z por la matriz A en la expresión que define a $f(z)$. En consecuencia, por ejemplo, $\det(A)$, la matriz adjunta y la matriz $p(X) = AX^2 + BX + C$ (donde todas las matrices son $n \times n$) no son funciones matriciales en este sentido.

Una de las propiedades más importantes es que $f(A)$ es un polinomio en $A \in \mathbb{C}^{n\times n}$, los coeficientes del polinomio dependen de A. Esta propiedad no sorprende en vista del teorema de Cayley-Hamilton.

Teorema 1.1 (Cayley-Hamilton) *Sea $A \in \mathbb{C}^{n\times n}$ y $q(t) = det(tI - A)$ el polinomio característico de A. Entonces, $q(A) = 0$.*

El teorema implica que la potencia n-ésima de A, y por inducción todas las potencias superiores, se pueden expresar como una combinación lineal de $I, A, \cdots, A^{n-1}$. Así, cualquier serie de potencias en A puede ser reducida a un polinomio de A de grado, a lo sumo, $n - 1$.

Otras propiedades se recogen en el siguiente teorema.

Teorema 1.2 *Sea $A \in \mathbb{C}^{n\times n}$ y f definida en el espectro de A, entonces*

(a) $f(A)$ *conmuta con* A

(b) $f(A^T) = f(A)^T$

(c) $f(XAX^{-1}) = Xf(A)X^{-1}$

(d) los valores propios de $f(A)$ *son* $f(\lambda_i)$*, donde* λ_i *son los valores propios de* A

(e) si $A = (A_{ij})$ *es triangular, entonces* $F = f(A)$ *es triangular con la misma estructura de* A *y* $F_{ii} = f(A_{ii})$

(f) si $A = diag(A_{11}, A_{22}, \cdots, A_{mm})$ *es diagonal por bloques, entonces*

$$f(A) = diag(f(A_{11}), f(A_{22}), \cdots, f(A_{mm}))$$

A menudo conviene representar a la función matricial como una serie de potencias o serie de Taylor.

Teorema 1.3 (Convergencia de la serie de Taylor de una matriz) *Si f tiene una serie de Taylor*

$$f(z) = \sum_{k=0}^{\infty} a_k(z-\alpha)^k \qquad \left(a_k = \frac{f^{(k)}(\alpha)}{k!}\right)$$

con radio de convergencia r. Si $A \in \mathbb{C}^{n\times n}$ entonces $f(A)$ se define mediante

$$f(A) = \sum_{k=0}^{\infty} a_k(A-\alpha I)^k$$

si y solo si cada uno de los valores propios $\lambda_1, \cdots, \lambda_s$ de A, donde s es el número de valores propios, satisface una de las condiciones

(a) $|\lambda_i - \alpha| < r$,

(b) $|\lambda_i - \alpha| = r$ *y la serie para* $f^{(n_i-1)}(\lambda)$ *(donde* n_i *es el indice de* λ_i*) es convergente en el punto* $\lambda = \lambda_i, i = 1 : s$.

De este modo, la exponencial de una matriz se define mediante la evaluación de la siguiente serie de potencias.

$$\mathrm{e}^A = I + A + \frac{A^2}{2!} + \frac{A^3}{3!} + \cdots$$

y el seno y coseno mediante

$$\sin(A) = A - \frac{A^3}{3!} + \frac{A^5}{5!} - \frac{A^7}{7!} + \cdots$$

$$\cos(A) = I - A + \frac{A^2}{2!} + \frac{A^4}{4!} - \frac{A^6}{6!} + \cdots$$

Otras funciones importantes son: la función signo, raíz cuadrada, ráiz p-ésima y logaritmo.

1.2. Técnicas generales

Existen diferentes técnicas para calcular o aproximar funciones matriciales, algunas de ellas muy generales y otras especializadas en determinadas funciones.

1.2.1. Transformaciones de semejanza

El uso de transformaciones de semejanza se basa en la siguiente relación: $f(XBX^{-1}) = Xf(B)X^{-1}$. La idea es factorizar $A = XBX^{-1}$, con B de una forma que permita fácilmente calcular $f(B)$. Entonces $f(A) = Xf(B)X^{-1}$ se obtiene rápidamente.

La opción más obvia de semejanza es una diagonalización, suponiendo que A es diagonalizable: $A = X\mathrm{diag}(\lambda_i)X^{-1}$, donde los λ_i son los valores propios de A y las columnas de X sus vectores propios.

En aritmética de precisión finita, este planteamiento es fiable solamente si X está bien condicionada, esto es, si el número de condición $\kappa(X) = \|X\|\,\|X^{-1}\|$ no es demasiado grande. En la clase de matrices normales ($A^*A = AA^*$, $A \in \mathbb{C}^{n\times n}$) se garantiza que la diagonalización esta bien condicionada. Esta clase incluye las matrices ortogonales, simétricas, antisimétricas y sus análogas complejas, unitarias, hermitianas y antihermitianas. De hecho, las matrices normales son precisamente esas que son diagonalizables mediante una matriz unitaria, y las matrices unitarias están perfectamente condicionadas en la 2-norma $\kappa_2(X) = 1$. La descomposición espectral $A = QDQ^*$ con Q unitaria y D diagonal es siempre posible y si esta descomposición

se puede hallar, entonces la fórmula $f(A) = Qf(D)Q^*$ ofrece una excelente manera de calcular $f(A)$.

En matrices generales, si X debe ser unitaria, lo más cerca a la forma diagonal que A puede ser reducida es la forma Schur: $A = QTQ^*$ donde Q es unitaria y T triangular superior. Los valores proprios de A aparecen en la diagonal de T. La descomposición Schur se puede hallar mediante el algoritmo QR, entonces $f(A) = Qf(T)Q^*$ se reduce ahora a evaluar f con una matriz triangular superior.

Surge un problema con el método de la diagonalización cuando A es real con algún valor propio complejo y $f(A)$ es real, entonces errores de redondeo pueden causar que $f(A)$ tenga una pequeña parte imaginaria distinta de cero (Para las matrices simétricas el sistema de valores propios es real y estas consideraciones no se aplican). Una forma de solventar este problema es emplear la diagonalización por bloques $A = XDX^{-1}$ con $D, X \in \mathbb{R}^{n\times n}$ y D tiene bloques diagonales 2×2 correspondiente a los valores propios complejos conjugados.

1.2.2. Aproximaciones racionales y polinomiales

La forma natural de aproximar $f(A)$ es imitar lo que se hace con las funciones escalares: aproximar $f(A)$ mediante $r(A)$ donde r es alguna aproximación racional o polinomial a f. De la teoría sobre la aproximación escalar se sabe que $f(z) \approx r(z)$ es una buena aproximación en cierta región de $\mathbb{C}$. Sin embargo, no hay garantía de que la aproximación matricial $f(A) \approx r(A)$ sea igual de buena si el espectro de A se encuentra en esta región. En realidad, si A es diagonalizable con $A = XDX^{-1}$, entonces $f(A) - r(A) = X(f(D) - r(D))X^{-1}$, por lo que $\|f(A) - r(A)\| \leq \kappa(X)\,\|f(D) - r(D)\|$. Por consiguiente, el error en la aproximación matricial es $\kappa(X)$ veces mayor que el error en la aproximación escalar. Si A es normal, $\kappa_2(X) = 1$, la aproximación escalar y matricial son esencialmente la misma.

Una aproximación polinomial cuando f tiene una serie de Taylor $f(z) = \sum_{i=0}^{\infty} a_i z^i$ es la serie de Taylor truncada

$$T_k(A) = \sum_{i=0}^{k} a_i A^i$$

Dentro de las aproximaciones racionales, la aproximación de Padé resulta ser particularmente útil. $r_{km}(z) = p_{km}(z)/q_{km}(z)$ es una aproximación de Padé $[k/m]$ de f si p_{km} y q_{km} son polinomios de grado k y m, respectivamente , $q_{km}(0) = 1$ y $f(z) - r_{km}(z) = \mathcal{O}(z^{k+m+1})$. Un aproximante de Padé

reproduce tantos términos como sean posibles de la serie de Taylor sobre el origen. Si un aproximante $[k/m]$ de Padé existe, entonces es único.

1.2.3. Iteraciones matriciales

Algunas funciones matriciales son susceptibles de calcular mediante iteraciones de la forma

$$X_{k+1} = g(X_k)$$

donde g es una función polinómica o racional. En la práctica, la matriz de inicio X_0 se toma como $X_0 = I$ o $X_0 = A$ y la función de iteración g puede ser o no dependiente de A.

Tales iteraciones son fáciles de implementar mediante la multiplicación de matrices y la solución de sistemas lineales con múltiples lados derechos. Sin embargo, la estabilidad numérica en aritmética de precisión finita es un problema.

1.3. El problema $f(A)b$

En muchas aplicaciones, especialmente en el contexto de matrices dispersas de gran dimensión, no es necesario calcular $f(A)$ sino que se requiere el cálculo de la acción $f(A)b$ para un determinado vector b. Esto es análogo a resolver el sistema lineal $Ax = b$ sin calcular A^{-1}.

1.3.1. Métodos de Krylov

Estos métodos consisten, básicamente, en obtener una aproximación de $f(A)b$ a partir del subespacio de Krylov generado por los vectores $\{b, Ab, A^2b, \cdots, A^{m-1}b\}$ siendo m un número entero positivo relativamente pequeño con relación a la dimensión de la matriz A.

$$\mathcal{K}_m(A, b) = \text{span}\{b, Ab, \cdots, A^{m-1}b\}$$

Entre estos métodos se encuentra el procedimiento de Arnoldi.

Procedimiento de Arnoldi

El procedimiento de Arnoldi para una matriz $A \in \mathbb{C}^{n\times n}$ obtiene una reducción a la forma Hessenberg, $Q^*AQ = H$, donde $Q \in \mathbb{C}^{n\times n}$ es unitaria y $H \in \mathbb{C}^{n\times n}$ es Hessenberg superior. Escribiendo $Q = [q_1, \cdots, q_n]$ y equiparando k columnas en $AQ = QH$ tenemos que

$$Aq_k = \sum_{i=1}^{k+1} h_{ik} q_i, \qquad k = 1 : n-1 \tag{1.1}$$

Esto se puede reescribir como

$$h_{k+1,k} q_{k+1} = Aq_k - \sum_{i=1}^{k} h_{ik} q_i =: r_k \tag{1.2}$$

donde, usando el hecho de que Q es unitaria,

$$h_{ik} = q_i^* A q_k, \qquad i = 1 : k \tag{1.3}$$

siempre que $r_k \neq 0$, $q_{k+1} = r_k / h_{k+1,k}$, con $h_{k+1,k} = \|r_k\|_2$.

A partir de (1.1) se deduce por inducción que

$$\text{span}\{q_1, \cdots, q_k\} = \text{span}\{q_1, Aq_1, \cdots, A^{k-1} q_1\}$$

es decir, los vectores de Arnoldi forman una base ortonormal para el subespacio de Krylov $\mathcal{K}_k(A, q_1)$. El procedimiento de Arnoldi produce la factorización

$$AQ_k = Q_k H_k + h_{k+1,k} q_{k+1} e_k^*$$

donde $Q_k = [q_1, \cdots, q_k]$ y $H_k = (h_{ij})$ es una matriz $k \times k$ Hessenberg superior. Las columnas 1 hasta $k-1$ de esta factorización son justo (1.1), mientras que la columna k es (1.2). La ecuación (1.3) puede ser escrita como

$$Q_k^* A Q_k = H_k$$

que dice que H_k es la proyección ortogonal de A en $\text{span}\{q_1, \cdots, q_k\} = \mathcal{K}_k(A, q_1)$.

Tras generar la base ortonormal $q_1, \cdots, q_m$ para $\mathcal{K}_m(A, q_1)$ el procedimiento termina si el *residuo* es igual a cero

$$h_{m+1,m} q_{m+1} = 0 := r_m,$$

ya que, entonces $\mathcal{K}_m(A, q_1) = \mathcal{K}_{m+1}(A, q_1)$.

Dada una matriz $A \in \mathbb{C}^{n \times n}$ y $q_1 \in \mathbb{C}^n$ de 2-norma unidad, el algoritmo 1.1 usa el procedimiento de Arnoldi para calcular la factorización $AQ = QH$, donde $Q \in \mathbb{C}^{n \times m}$ $(m \leq n)$ con columnas ortonormales y $H \in \mathbb{C}^{m \times m}$ es Hessenberg supeior.

El algoritmo 1.1 usa el método de Gram-Schmidt modificado para ortogonalizar Aq_k contra $q_1, \cdots, q_k$. El método Gram-Schmidt clásico se puede

Algoritmo 1.1 Procedimiento de Arnoldi

```
1: for k = 1 : n do
2:    z = Aq_k
3:    for i = 1 : k do
4:       h_ik = q_i^* z
5:       z = z − h_ik q_i
6:    end for
7:    h_{k+1,k} = ‖z‖_2
8:    if h_{k+1,k} = 0 ∨ m = k then
9:       quit
10:   end if
11:   q_{k+1} = z/h_{k+1,k}
12: end for
```

usar en su lugar, remplazando la línea 2 por las asignaciones $r_k = Aq_k$, $z = r_k$ y z por r_k en la línea 5 y en adelante. La perdida de ortogonalidad en los vectores de Arnoldi puede ocurrir en aritmética de precisión finita para ambos métodos de Gram-Schmidt (más para el clásico Gram-Schmidt), pero se puede solucionar mediante reortogonalización. Otra alternativa es usar una implementacion del procedimiento de Arnoldi basado en las reflexiones de Householder.

Cuando A es Hermitiana muchos de los productos interiores del bucle interno del procedimiento de Arnoldi son cero y no necesitan ser calculados. En realidad, para matrices Hermitianas el procedimiento de Arnoldi se reduce al procedimiento de Lanczos, el cual produce una matriz H_k tridiagonal simétrica real.

Aproximación Arnoldi de $f(A)b$

Si el procedimiento de Arnoldi, con la matriz A y el vector inicial $q_1 = b/\|b\|_2$, completa k pasos, se puede aproximar $f(A)b$ mediante

$$\begin{aligned} f_k &:= \|b\|_2 Q_k f(H_k) e_1 \\ &= Q_k f(H_k) Q_k^* b \end{aligned}$$

La evaluación de f se lleva a cabo con la matriz $k \times k$ H_k, donde en la práctica $k \ll n$ y se puede hacer con cualquiera de los métodos disponibles para matrices densas de pequeña dimensión. Realmente, se esta evaluando f en el subespacio de Krylov $\mathcal{K}(A, q_1)$ y hay que llevar el resultado al espacio original $\mathbb{C}^n$.

1.3.2. Cuadratura y aproximación racional

Si tenemos una representación integral $f(A) = \int r(A,t)dt$, donde r es una función racional de A, entonces se puede aplicar cuadratura y aproximar $f(A)b$ mediante $\sum_i r(A,t_i)b$. Dependiendo de como r se represente, la evaluación de esta aproximación se reduce a resolver uno o más sistemas lineales.

Para una función analítica f se puede emplear la fórmula integral de Cauchy

$$f(A) = \frac{1}{2\pi i}\int_\Gamma f(z)(zI - A)^{-1}dz \tag{1.4}$$

donde Γ es un contorno cerrado que se encuentra en la región de f y rodea el espectro en el sentido contrario a las agujas del reloj. De (1.4) se obtiene

$$f(A)b = \frac{1}{2\pi i}\int_\Gamma f(z)(zI - A)^{-1}bdz \tag{1.5}$$

Cualquier método basado en (1.5) necesita ser especializado para una f y matriz A en particular, ya que la selección del contorno Γ es crucial para la efectividad y fiabilidad del método.

1.4. La función exponencial

Especial interés tiene la función exponencial por su papel en la solución de ecuaciones diferenciales. El clásico problema escalar

$$\frac{dy}{dt} = ay, \qquad y(0) = b,$$

tiene solución $y(t) = \mathrm{e}^{at}b$, mientras el problema análogo vectorial

$$\frac{dy}{dt} = Ay, \qquad y(0) = b, \qquad y \in \mathbb{C}^n, A \in \mathbb{C}^{n\times n},$$

tiene solución $y(t) = \mathrm{e}^{At}b$.

Han sido propuestos numerosos métodos para calcular e^A. En "Nineteen dubious ways to Compute the Exponential of a Matrix"[10] de Moler y Van Loan se clasifican una amplia variedad de ellos. La aproximación de Padé, combinada con el método de escalado y potenciación, es el método más usado.

La aproximación de Padé de orden (p,q) para e^A se define mediante la expresión

$$R_{pq}(A) = [D_{pq}(A)]^{-1} N_{pq}(A),$$

donde

$$N_{pq}(A) = \sum_{j=0}^{p} \frac{(p+q-j)!p!}{(p+q)!j!(p-j)!} A^j,$$

y

$$D_{pq}(A) = \sum_{j=0}^{q} \frac{(p+q-j)!q!}{(p+q)!j!(q-j)!} (-A)^j.$$

El problema es que únicamente proporcionan buenas aproximaciones cerca del origen. Una manera de evitar esto es hacer uso de la propiedad

$$\mathrm{e}^A = \left(\mathrm{e}^{\frac{A}{m}}\right)^m$$

El método de escalado y potenciación consiste en seleccionar un valor m como potencia de 2, $m = 2^j$, de manera que $\mathrm{e}^{A/m}$ pueda calcularse con precisión y eficiencia, y entonces calcular $(\mathrm{e}^{A/m})^m$ mediante potenciación repetida. Eligiendo j como el menor entero positivo que cumpla

$$\frac{\|A\|_\infty}{2^j} \leq \frac{1}{2},$$

se obtiene el valor de m más adecuado.

Existen varias razones para utilizar aproximantes diagonales de Padé ($p = q$) en lugar de los no diagonales ($p \neq q$). Por ejemplo, si todos los valores propios de A están en el semiplano $\mathrm{Re}(z) < 0$, entonces los aproximantes calculados con $p > q$ suelen tener errores de redondeo debido a problemas de cancelación, mientras que con $p < q$ se suelen producir debido al mal condicionamiento del denominador $D_{pq}(A)$.

En muchas ocasiones puede ser interesante calcular directamente el producto $\mathrm{e}^{At}b$ sin hallar expresamente la matriz e^{At}. Los métodos que calculan este producto están basados en los subespacios de Krylov, como el procedimiento de Lanczos para matrices simétricas, y el procedimiento de Arnoldi para el caso en que A no sea simétrica. Partiendo de $A \in \mathbb{C}^{n\times n}$ y del vector inicial $b \in \mathbb{C}^m$ se obtiene la reducción de Hessenberg,

$$AV_m = V_m H_m + h_{m+1,m} v_{m+1} e_p^*$$

donde V_m es una base ortonormal de m vectores, H_m es una matriz Hessenberg superior de dimensión m, e_m es la columna m-ésima de la matrix I_n, y v_{m+1} es un vector unitario que satisface $V_m^* v_{m+1} = 0$. La aproximación que se realiza es

$$\mathrm{e}^{At} b \approx \beta V_m \mathrm{e}^{H_m} e_1^*$$

siendo $\beta = \|b\|_2$ y e_1 la primera columna de I_n. Se pueden obtener muy buenas aproximaciones con valores de m relativamente pequeños.

1.5. Software disponible

Sofware disponible para el cálculo de funciones matriciales.

MATLAB En MATLAB se puede usar `funm` para evaluar funciones matriciales. La tabla 1.1 muestra la sintaxis para las funciones predefinidas.

Función	Sintaxis
exponencial	`funm (A, @exp)`
logaritmo	`funm (A, @log)`
seno	`funm (A, @sin)`
coseno	`funm (A, @cos)`
seno hiperbólico	`funm (A, @sinh)`
coseno hiperbólico	`funm (A, @cosh)`

Tabla 1.1: Sintaxis de `funm`

Para calcular la raíz cuadrada se puede usar `sqrtm(A)`. La función `expm` implementa el método de escalado y potenciación en el cálculo de la función exponencial y la función `logm` implementa una versión especifica para el logaritmo.

Octave GNU Octave es un programa libre similar a MATLAB. Incluye la función `expm` y la función `thfm` para evaluar funciones trigonométricas e hiperbólicas.

SPARSKIT Es un paquete de subrutinas escritas en FORTRAN para trabajar con matrices dispersas. Incluye algunas rutinas de manipulación de matrices dispersas así como algunos *solvers* iterativos. Las rutinas relacionadas con la matriz exponencial son `exppro`, que calcula $\mathrm{e}^{tA}v$ y `phipro` para $\mathrm{e}^{tA}v + t\varphi(tA)u$.

EXPOKIT Es el software más extendido para calcular la matriz exponencial. Calcula e^{tA} para matrices pequeñas densas y $e^{tA}v$ para matrices dispersas de gran dimensión. Las rutinas para matrices dispersas utilizan métodos de Krylov. Está disponible para FORTRAN y MATLAB.

Capítulo 2

EXPOKIT

En este capítulo se describe el paquete de software EXPOKIT y se analiza, partiendo de los aspectos clave sobre funciones matriciales vistos en el capítulo 1, el algoritmo para el cálculo de $e^{tA}v$.

2.1. Descripción

EXPOKIT es un paquete de software que provee un conjunto de rutinas dirigidas al cálculo de la exponencial de una matriz. Más concretamente, calcula la exponencial de pequeñas matrices densas, la exponencial de grandes matrices dispersas por un vector o la solución de un sistema lineal de ecuaciones diferenciales ordinarias (EDO) de la forma

$$\begin{cases} \frac{dw(t)}{dt} = Aw(t) + u, & t \in [0, T] \\ w(0) = v, & \text{valor inicial} \end{cases}$$

Su solución analítica se sabe que es

$$w(t) = e^{tA}v + t\varphi(tA)u$$

donde $\varphi(x) = (e^x - 1)/x$.

Cuando $u = 0$, la solución es simplemente $w(t) = e^{tA}v$. EXPOKIT provee rutinas en FORTRAN 77 y MATLAB para abordar ambas situaciones (caso $u = 0$ y caso $u \neq 0$) tanto con matrices reales como con matrices complejas y suministra rutinas especificas para matrices simétricas o Hermitianas.

2.2. Fundamentos

A continuación se detallan los fundamentos del algoritmo propuesto por EXPOKIT para el cálculo de la exponencial de una matriz por un vector.

Algoritmo 2.1 Calcula w(t) = exp(tA)v

$w = v; t_k = 0;$
$\bar{H}_{m+2} = \text{zeros}[m+2, m+2];$
while $t_k < t$ **do**
  $v = w; \beta = \|v\|_2;$
  $v_1 = v/\beta;$
  for $j = 1 : m$ **do**
    $p = Av_j;$
    for $i = 1 : j$ **do**
      $h_{ij} = v_i^* p;$
      $p = p - h_{ij} v_i;$
    end for
    $h_{j+1,j} = \|p\|_2;$
    if $h_{j+1,j} \leq tol\,\|A\|$ **then**
      happy-breakdown
    end if
    $v_{j+1} = p/h_{j+1,j};$
  end for
  $\bar{H}(m+2, m+1) = 1;$
  repeat
    $\tau = stepsize;$
    $F_{m+2} = \exp(\tau \bar{H}_{m+2});$
    $w = \beta V_{m+1} F(1 : m+1, 1);$
    $err\_loc = local\ error\ estimate;$
  until $err\_loc \leq \delta tol;$
  $t_k = t_k + \tau;$
end while

Considérese una matriz grande y dispersa A de $n \times n$ elementos (real o compleja), un vector v y un escalar $t \in \mathbb{R}$. El algoritmo 2.1 propuesto por EXPOKIT calcula una aproximación de $w(t) = \exp(tA)v$. El principio fundamental es aproximar

$$w(t) = \mathrm{e}^{tA} v = v + \frac{(tA)}{1!} v + \frac{(tA)^2}{2!} v + \cdots$$

Dado un entero m, la expansión de Taylor, puede ser truncada en orden $m-1$, produciendo de esta forma una aproximación polinómica de grado

$m-1$

$$c_0 v + c_1(tA)v + c_2(tA)^2 v + \cdots + c_{m-1}(tA)^{m-1}v,$$

con coeficientes $c_i = 1/i!$ que aproxima al vector. Sin embargo, estos no son necesariamente los mejores coeficientes y se sabe que todas las aproximaciones polinómicas de grado a lo sumo $m-1$ (incluyendo el polinomio truncado de Taylor, así como la aproximación polinómica óptima) son elementos del subespacio de Krylov definido como

$$\mathcal{K}_m(tA, v) = \text{Span}\{v, (tA)v, \cdots, (tA)^{m-1}v\}.$$

Por lo tanto, se reformula el problema al de como encontrar un elemento de $\mathcal{K}_m(tA, v)$ que aproxime $w(t)$.

Los elementos del subespacio de Krylov son mejor manipulados mediante su representación en una base ortonormal. El procedimiento Arnoldi construye convenientemente esta base sobre la secuencia de potencias $[v, (tA)v, \cdots, (tA)^{m-1}v]$.

Partiendo del vector inicial v, si Arnoldi se aplica simplemente con A (más que con tA) se construyen dos matrices

$$V_{m+1} = [v_1, v_2, \cdots, v_{m+1}] \in \mathbb{C}^{n\times(m+1)}$$

y

$$\bar{H} = [h_{ij}] \in \mathbb{C}^{(m+1)\times m}$$

satisfaciendo la relación

$$AV_m = V_{m+1}\bar{H} = V_m H_m + h_{m+1,m} v_{m+1} e_m^* \quad (2.1)$$

$$V_m^* A V_m = H_m \quad (2.2)$$

Para $j = 1, \cdots, m+1$, el subconjunto $V_j = [v_1, \cdots, v_j] \in \mathbb{C}^{n\times j}$ es una base ortonormal de $\mathcal{K}_j(A, v)$, es decir, $V_j^* V_j = I$. El primer vector de la base v_1 es v normalizado, particularmente $v = \beta V_m e_1$, donde $\beta = \|v\|_2$ y e_i es el i-ésimo vector unitario.

Las matrices $\bar{H}$ y H_m tienen una forma especial

$$\bar{H} = \begin{pmatrix} & H_m & \\ 0 & \cdots \quad 0 & h_{m+1,m} \end{pmatrix} \in \mathbb{C}^{(m+1)\times m}$$

$$H_m = \begin{pmatrix} h_{11} & h_{12} & h_{13} & \cdots & h_{1m} \\ h_{21} & h_{22} & h_{23} & \cdots & h_{2m} \\ & \ddots & \ddots & & \vdots \\ & & \ddots & \ddots & \vdots \\ 0 & & & h_{m,m+1} & h_{mm} \end{pmatrix} \in \mathbb{C}^{m\times m}$$

$H_m = V_m^* A V_m$ es una matriz Hessenberg (triangular con una subdiagonal extra) y representa la proyección del operador A sobre el subespacio de Krylov con respecto a la base V_m.

Para cualquier escalar τ, $\mathcal{K}_m(\tau A, v) = \mathcal{K}_m(A, v)$, de hecho, multiplicando (2.1) y (2.2) por τ se muestra que es fácil extender estas relaciones a τA usando $\tau\bar{H}$ y τH_m sin cambiar V_{m+1}. Por lo tanto no hay pérdida de generalidad cuando se aplica el procedimiento Arnoldi directamente a A.

Sea w_{opt} la aproximación óptima de Krylov a $w(\tau) = \exp(\tau A)v$, donde V_m es una base del subespacio de Krylov, $w_{opt} = V_m y_{opt}$ con $y_{opt} \in \mathbb{C}^m$. Por definición

$$\|w_{opt} - w\tau\|_2 = \min_{x\in\mathcal{K}_m(\tau A,v)} \|x - w\tau\|_2 = \min_{y\in\mathbb{C}^m} \|V_m y - w\tau\|_2$$

El problema lineal de mínimos cuadrados definido por la última expresión es de rango completo y es sabido que su solución y_{opt} se puede expresar en términos de V_m^+ (inversa de Moore-Penrose de V_m) como

$$y_{opt} = V_m^+ w(\tau) = (V_m^* V_m)^{-1} V_m^* w(\tau) = V_m^* \exp(\tau A)v$$

Usando el hecho que $v = \beta V_m e_1$ tenemos que

$$w_{opt} = V_m y_{opt} = \beta V_m (V_m^* \exp(\tau A) V_m) e_1$$

Esta relación caracteriza a la aproximación óptima pero es incómoda ya que requiere $\exp(\tau A)$. Sin embargo, se puede aproximar $V_m^*\exp(\tau A)V_m$ por $\exp(\tau V_m^* A V_m)$. En otras palabras, la proyección del operador exponencial $\exp(\tau A)$ con respecto a la base V_m es aproximada por la exponencial de la proyección del operador τA con respecto a la misma base. Ahora sustituyendo (2.2), se obtiene la aproximación

$$\exp(\tau A)v \approx \beta V_m \exp(\tau H_m) e_1$$

Por otra parte, se puede hacer un mayor uso de los componentes calculados por el procedimiento de Arnoldi para definir una aproximación mejorada

$$\exp(\tau A)v \approx \beta V_{m+1} \exp(\tau \bar{H}_{m+1}) e_1 \tag{2.3}$$

donde

$$\bar{H}_{m+1} = \left(\bar{H}|0\right) \in \mathbb{C}^{(m+1)\times(m+1)}$$

Así, la característica distintiva de esta aproximación de Krylov es que el problema original de gran tamaño n se convierte en un problema más pequeño (por lo general $m \ll 50$, mientras que n puede superar muchos miles). El cálculo del problema de tamaño reducido ($\exp(t\bar{H}_{m+1})$) se hace con los métodos clásicos densos, tales como el método de Padé.

Debido a que en realidad $t\,\|A\|$ puede ser muy grande, $w(t) = \exp(tA)v$ no se calcula de una sola vez. En su lugar se emplea una estrategia paso a paso junto a una estimación de error. Típicamente, el algoritmo evoluciona con el esquema de integración

$$\left\{ \begin{array}{lcl} w(0) & = & v \\ w(t_{k+1}) & = & \exp((t_k + \tau_k)A)v = \exp(\tau_k A)w(t_k) \end{array} \right.$$

donde

$$k = 0, 1, \cdots, s \quad \tau_k = t_{k+1} - t_k \quad 0 = t_0 < t_1 < \cdots < t_s < t_{s+1} = t.$$

La selección de un determinado tamaño de paso τ se hace de modo que $\exp(\tau A)v$ sea eficazmente aproximada por (2.3) usando la información del actual proceso de Arnoldi. Siguiendo el procedimiento para solucionar EDOs, un control *a posteriori* de error se lleva a cabo para asegurar que la aproximación intermedia sea aceptable con respecto a las expectativas del error global. El punto de partida de esta crítica cuestión es la siguiente expansión en serie

$$\begin{aligned} \exp(\tau A)v &= \beta V_m \exp(\tau H_m)e_1 + \beta\tau h_{m+1,m} \sum_{j=1}^{\infty} e_m^* \varphi_j(\tau H_m) e_1 (\tau A)^{j-1} v_{m+1} \\ &= \beta V_{m+1} \exp(\tau \bar{H}_{m+1})e_1 + \beta \sum_{j=2}^{\infty} h_{m+1,m}\tau^j e_m^* \varphi_j(\tau H_m) e_1 A^{j-1} v_{m+1} \end{aligned}$$

donde $\varphi_0(z) \equiv \mathrm{e}^z$, $\varphi_j(z) \equiv (\varphi_{j-1}(z) - \varphi_{j-1}(0))/z = \sum_{i=0}^{\infty} z^i/(i+j)!$, $j \geq 1$. Las funciones φ_j son positivas, crecientes, y $\varphi_{j+1} \leq \varphi_j/j$ en $[0, +\infty)$. Una forma de controlar el error viene dada por el algoritmo 2.2, donde los coeficientes $h_{m+1,m}\tau^j e_m^* \varphi_j(\tau H_m)e_1$ se calculan eficientemente basándose en el teorema 2.1.

Algoritmo 2.2 Estimación del error local

$err1 = \beta|h_{m+1,m}\tau e_m^* \varphi_1(\tau H_m)e_1|$;
$err2 = \beta|h_{m+1,m}\tau^2 e_m^* \varphi_2(\tau H_m)e_1| \, \|Av_{m+1}\|_2$;
if $err1 \gg err2$ **then**
  $err = err2$;
else if $err1 > err2$ **then**
  $err = err2 * \frac{1}{1-\frac{err2}{err1}}$;
else
  $err = err1$;
end if
$err\_loc = \max(err, roundoff)$;

Teorema 2.1 *Sea* $c \in \mathbb{C}^m$ *y*

$$\tilde{H}_{m+p} = \begin{pmatrix} H_m & c & 0 & \cdots & 0 \\ & 0 & 1 & \ddots & \vdots \\ & & 0 & \ddots & 0 \\ & & & \ddots & 1 \\ 0 & & & & 0 \end{pmatrix} \in \mathbb{C}^{(m+p)\times(m+p)}$$

entonces

$$exp(\tau\tilde{H}_{m+p}) = \begin{pmatrix} exp(\tau H_m) & \tau\varphi_1(\tau H_m)c & \tau^2\varphi_2(\tau H_m)c & \cdots & \tau^p\varphi_p(\tau H_m)c \\ & 1 & \frac{\tau}{1!} & \ddots & \frac{\tau^{p-1}}{(p-1)!} \\ & & 1 & \ddots & \vdots \\ & & & \vdots & \frac{\tau}{1!} \\ 0 & & & & 1 \end{pmatrix}$$

Demostración 2.1 *Considérese la descomposición triangular superior*

$$\tilde{H}_{m+p} = \begin{pmatrix} H_m & ce_1^* \\ 0 & J \end{pmatrix}, J = \begin{pmatrix} 0 & 1 & \cdots & 0 \\ & 0 & \ddots & \vdots \\ & & \ddots & 1 \\ 0 & & & 0 \end{pmatrix} = \begin{pmatrix} e_2^* \\ \vdots \\ e_p^* \\ 0 \end{pmatrix}$$

entonces

$$e^{\tau\tilde{H}_{m+p}} = \begin{pmatrix} e^{\tau H_m} & F \\ 0 & e^{\tau J} \end{pmatrix}, e^{\tau J} = \begin{pmatrix} 1 & \frac{\tau}{1!} & \cdots & \frac{\tau^{p-1}}{(p-1)!} \\ & 1 & \ddots & \vdots \\ & & \ddots & \frac{\tau}{1!} \\ 0 & & & 1 \end{pmatrix}, F = [f_1, \cdots, f_p]$$

A partir de $\tilde{H}_{m+p}e^{\tau\tilde{H}_{m+p}} = e^{\tau\tilde{H}_{m+p}}\tilde{H}_{m+p}$ *tenemos que* $H_mF - FJ = e^{\tau H_m}ce_1^* - ce_1^*e^{\tau J}$. *Multiplicando por la derecha por* e_j *para extraer la j-ésima columna, se obtiene la relación*

$$\begin{cases} f_1 &= \tau\varphi_1(\tau H_m)c \\ H_m f_j &= f_{j-1} - \frac{\tau^{j-1}}{(j-1)!}c \qquad 1 < j \leq p \end{cases}$$

y por inducción $f_j = \tau^{j-1}\left(\varphi_{j-1}(\tau H_m) - \varphi_{j-1}(0)I\right)H_m^{-1}c = \tau^j\varphi_j(\tau H_m)c$.

Cololario 2.1 *Sea* $c \in \mathbb{C}^m$ *y*

$$\bar{H}_{m+p} = \begin{pmatrix} H_m & & & & 0 \\ c^* & 0 & & & \\ 0 & 1 & 0 & & \\ \vdots & \ddots & \ddots & \ddots & \\ 0 & \cdots & 0 & 1 & 0 \end{pmatrix} \in \mathbb{C}^{(m+p)\times(m+p)}$$

entonces

$$exp(\tau\bar{H}_{m+p}) = \begin{pmatrix} exp(\tau H_m) & & & & 0 \\ \tau c^*\varphi_1(\tau H_m) & 1 & & & \\ \tau^2 c^*\varphi_2(\tau H_m) & \frac{\tau}{1!} & 1 & & \\ \vdots & \vdots & \ddots & \ddots & \\ \tau^p c^*\varphi_p(\tau H_m) & \frac{\tau^{p-1}}{(p-1)!} & \cdots & \frac{\tau}{1!} & 1 \end{pmatrix}$$

Asignando $c^* = h_{m+1,m}e_m^*$ en particular, los coeficientes deseados pueden ser recuperados de la primera columna justo debajo de la m-ésima fila de $\exp(\tau\bar{H}_{m+p})$.

Si *tol* indica la tolerancia establecida, el primer *stepsize* se elige para satisfacer un límite teórico conocido y dentro del proceso de integración se selecciona por medio de la fórmula

$$\tau_k = \gamma(tol/err_k)^{1/r}\tau_{k-1}$$

El valor de r es $m-1$ o m dependiendo de si el error estimado err proviene de $err1$ o $err2$ (ver algoritmo 2.2). Un paso es desestimado si $err_{k+1} > \delta tol$. Los escalares γ y δ son unos factores de seguridad destinados a reducir el riesgo de desestimación del paso. Se han asignado los clásicos valores 0,9 y 1,2 respectivamente.

2.3. Rutinas del paquete

Los nombres de las rutinas de FORTRAN 77 en EXPOKIT siguen el patrón `TXYYYZ` de acuerdo con la descripción de la tabla 2.1. No todas las combinaciones son posibles.

`T`	indica el tipo de dato
`D`	doble precisión
`Z`	complejo*16

`Z`	indica el alcance
`M`	Matriz
`V`	Vector

`X`	indica la matriz
`G`	General
`H`	Hermitiana
`S`	Simétrica
`M`	Markov
`N`	Hessenberg

`YYY`	indica la operación
`PAD`	Padé
`CHB`	Chebyshev
`EXP`	Exponencial
`PHI`	φ (problema no homogéneo)

Tabla 2.1: Nomenclatura

Algunas de las rutinas de alto nivel más relevantes son:

`__EXPV` calcula $w = \exp(tA)v$, t puede ser positivo o negativo. Las variantes que se ocupan de matrices simétricas o Hermitianas usan el proceso de Lanczos en lugar del proceso de Arnoldi.

`__PHIV` calcula $w = \exp(tA)v + t\varphi(tA)u$ que es la solución a la EDO lineal no homogénea. El parámetro de entrada t puede ser positivo o negativo. Si $u = 0$ el procedimiento es matemáticamente equivalente a `__EXPV` y si $v = 0$ calcula $t\varphi(tA)u$. Las variantes que se ocupan de matrices simétricas o Hermitianas usan el proceso de Lanczos en lugar del proceso de Arnoldi.

`__PADM` calcula la exponencial de una matriz $\exp(tH)$ donde H es una matriz relativamente pequeña. El método empleado es la aproximación racional de Padé para la función exponencial combinada con escalado y potenciación.

`__CHBV` calcula $\exp(tH)y$ donde H es una matriz relativamente pequeña usando la aproximación racional de Chebyshev del tipo (14,14) para la función exponencial.

La versión homóloga a los algoritmos escrita en MATLAB está también incluida en la distribución. Por ejemplo, `expv.m` es homóloga a `__EXPV`.

Capítulo 3

SLEPc

En este capítulo se realiza una breve introducción a la librería SLEPc, su estructura básica y sus clases.

3.1. Las librerías SLEPc y PETSc

SLEPc (*Scalable Library for Eigenvalue Problem Computation*) [14] es una librería orientada a la resolución de problemas de valores propios y valores singulares grandes y dispersos en entornos paralelos. La mayoría de los métodos que ofrece SLEPc son métodos de proyección como Iteración del Subespacio, Arnoldi, Lanczos o Krylov-Schur. Soporta problemas estándares y generalizados, tanto Hermitianos como no Hermitianos, y en aritmética real o compleja.

SLEPc se apoya en PETSc (*Portable, Extensible Toolkit for Scientific Computation*) [12], proporcionándole la filosofía y las estructuras de datos básicas para resolver problemas numéricos de ecuaciones en derivadas parciales.

Como el propio acrónimo indica, PETSc es *Portable* porque es fácilmente utilizable en varios entornos, incluyendo una amplia variedad de compiladores y sistemas operativos. Por otra parte es *Extensible* debido a la estructura modular, capaz de aceptar extensiones con nuevas funcionalidades.

Otras características que cabe destacar:

- usa un estilo de programación orientado a objetos, facilitando su extensión y mantenibilidad;
- la implementación de la estructura de datos es neutra permitiendo, por ejemplo, ser compilado para precisión simple, doble o aritmética compleja;

- es muy flexible en tiempo de ejecución, con el objetivo de poder ajustar ciertos parámetros sin necesidad de recompilar;
- proporciona herramientas para la introespección, midiendo el tiempo de ejecución y el número de operaciones en coma flotante;
- es portable a una gran variedad de plataformas paralelas, que soporten MPI;
- ofrece interfaz a Fortran y a otros lenguajes que puedan interoperar con C

Las familias de objetos (matrices, vectores, etc) y las operaciones disponibles para cada uno de estos, están ordenadas en sucesivas capas de abstracción, con la base fundamental de BLAS, LAPACK y MPI como pilares primitivos.

- BLAS (*Basic Linear Algebra Subprograms*) está formado por una serie de rutinas creadas para operar vectores y matrices.
- LAPACK (*Linear Algebra PACKage*) proporciona rutinas para resolver sstemas lineales de ecuaciones, sistemas lineales generados por mínimos cuadrados, problemas de valor propio y problemas de valor singular.
- MPI (*Message Passing Interface*) es una especificación estándar para definir un protocolo de comunicación entre varias computadoras que trabajan en paralelo, independiente del lenguaje con el que trabajen internamente.

La figura 3.1 representa la funcionalidad proporcionada por SLEPc y cómo se relaciona con PETSc.

Los objetos `Matriz`, `Vector` y `Index Set` son la base de PETSc.

`Matriz` Un gran conjunto de estructuras de datos y código para manipular matrices dispersas paralelamente.

`Vector` Proporciona las operaciones con vectores necesarias para manipular y resolver problemas lineales y no lineales de gran escala. Facilita además su uso en entornos paralelos.

`Index Set` Herramientas de permutación, renumeración, etc.

SLEPc incorpora toda la funcionalidad necesaria para la resolución de problemas de valores propios:

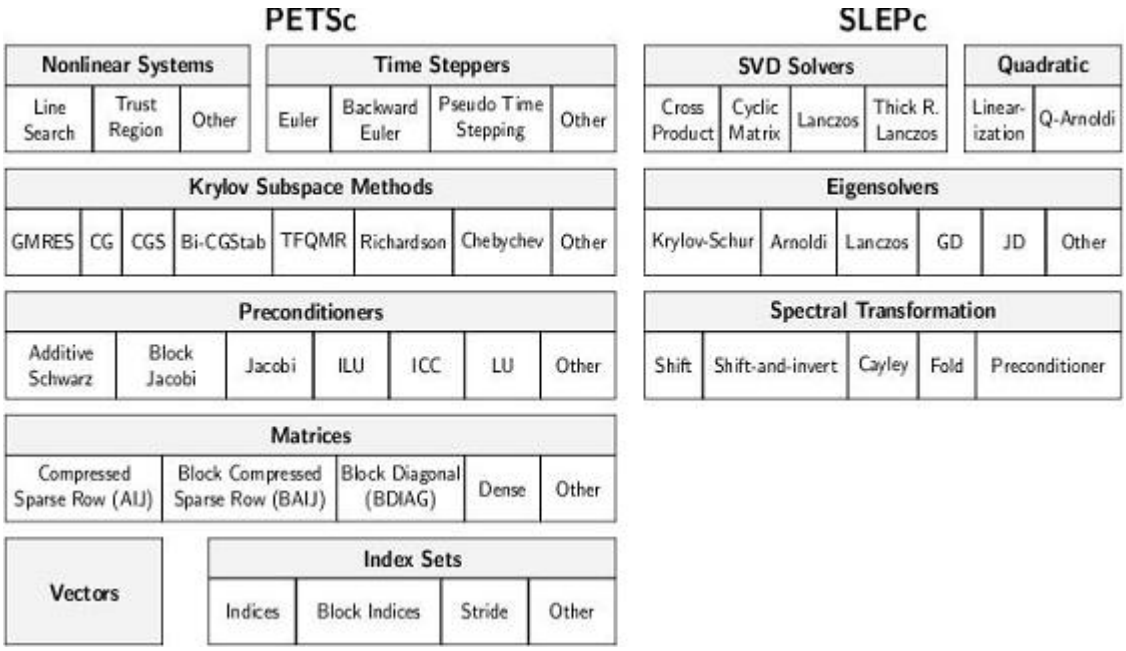

Figura 3.1: Organización de los paquetes PETSc y SLEPc

`EPS` El objeto *Eigenvalue Problem Solver* especifica un problema de valores propios, ya sea en su forma estándar o generalizada. Proporciona una interfaz uniforme a diversos *solvers* de valores propios.

`ST` La clase *Spectral Transformation* encapsula técnicas de aceleración basadas en la transformación del espectro. Los objetos `ST` están siempre relacionados con los `EPS`. Un usuario no debería crear un objeto `ST` independiente.

`SVD` El *Singular Value Decomposition* es muy similar al objeto `EPS`, pero destinado al cálculo de la descomposición parcial en valores singulares.

`QEP` El *Quadratic Eigenvalue Problem* es el objeto proporcionado por SLEPc para especificar un problema de valores propios cuadrático.

`DS` Los objetos *Direct Solver* proporcionan rutinas auxiliares que utilizan internamente los diferentes *solvers*. Representan problemas de dimensión más pequeña que son resueltos dentro de los *solvers* iterativos con métodos directos. Se pueden ver como una estructura contenedora de la funcionalidad de LAPACK.

`IP` *Inner Product* es un objeto auxiliar que encapsula diferentes aplicaciones del producto escalar, y varias implementaciones del procedimiento de ortogonalización de Gram-Schmidt.

La programación con SLEPc y PETSc se realiza principalmente mediante las clases de interfaz mostradas en la figura 3.1. Cada uno de los bloques se corresponde con clases abstractas que encapsulan tanto estructuras de datos (matrices y vectores) como métodos de resolución. En la parte inferior de los bloques se muestran algunas de las subclases encargadas de la implementación específica de una estrucutra de datos o de un método de resolución. La librería está organizada jerárquicamente de forma que cada clase depende para su implementación únicamente de la interfaz abstracta de las clases en los niveles inferiores del diagrama.

Los algoritmos que resuelven el mismo tipo de problema derivan de la misma clase abstracta, proporcionándoles una interfaz común. Mediante esta interfaz es posible cambiar de método numérico simplemente indicando la instancia de la subclase que lo implementa. El ciclo de vida típico de estos objetos es:

1. Creación del objeto.
2. Establecimiento de parámetros y selección del método.
3. Ejecución.
4. Recuperación de la solución.
5. Destrucción y liberación de memoria.

Esta misma filosofía de diseño se aplica a las estructuras de datos como matrices y vectores, resultando muy interesante para ocultar las distintas posibilidades de implementación.

3.2. Modelo de paralelismo

Las máquinas paralelas más potentes en la actualidad son prácticamente todas de tipo cluster con un gran número de nodos interconectados mediante una red de altas prestaciones. Estos nodos son ordenadores completos construidos a partir de los mismos componentes utilizados para equipos convencionales. Como cada uno de los nodos dispone de su propia memoria principal, el cluster es considerado una máquina paralela de memoria distribuida.

Desde hace bastante tiempo, los microprocesadores de gama alta incorporan facilidades hardware para compartir la memoria entre un número pequeño de procesadores independientes. Recientemente, este número se ha incrementado gracias a la inclusión de varios núcleos de proceso en un único circuito integrado. De esta forma, cada uno de los nodos de un cluster se convierte en una pequeña máquina paralela de memoria compartida.

Las máquinas paralelas de memoria compartida y distribuida se programan tradicionalmente de forma diferente, pero el paso de mensajes es más general y funciona también en las máquinas de memoria compartida. La alternativa más sencilla es utilizar este modelo entre todos los procesadores, compartan o no memoria.

PETSc utiliza el estándar MPI [5] para la comunicación entre procesadores. Este estándar permite desarrollar programas independientes de la arquitectura paralela subyacente. Se define como una librería de funciones que encapsulan las primitivas de paso de mensajes entre procesadores. En el modelo de ejecución de PETSc cada procesador arranca una copia idéntica del programa en un espacio de direcciones privado, lo que se ajusta a un modelo de memoria distribuida. También puede utilizarse en máquinas de memoria compartida. MPI está implementado en prácticamente todas las máquinas paralelas y suele tener excelentes prestaciones porque ha sido optimizado para el hardware de interconexión por el propio fabricante. También existen implementaciones del estándar de dominio público que funcionan sobre hardware de interconexión de propósito general como MPICH y OpenMPI.

3.3. Distribución de datos

En PETSc se intenta que la paralelización sea prácticamente transparente para el usuario. Para ello la gestión de la distribución de datos se realiza de forma automática. Esto no impide que el usuario pueda modificar esta distribución si lo considera necesario.

La distribución de datos por defecto en PETSc para las matrices es por bloques de filas donde a cada procesador se le asigna un conjunto de filas contiguas. Los elementos de los vectores se encuentran repartidos en bloques contiguos entre los procesadores. Con esta distribución, la paralelización de las primitivas típicas que aparecen en los algoritmos para matrices dispersas queda de la forma siguiente:

Producto matriz por vector PETSc proporciona la función `MatMult`, cuyas prestaciones dependen del patrón de dispersión de la matriz. Si se

elige un reparto adecuado de los elementos entre los procesadores solamente son necesarias comunicaciones punto a punto entre procesadores vecinos.

Producto escalar y norma de vectores Las funciones `VecDot` y `VecNorm` calculan el producto escalar y la norma con una multireducción. Esta primitiva de MPI es una comunicación global donde todos los procesadores envían sus datos parciales y todos reciben el resultado global de la operación. La implementación de esta primitiva implica múltiples mensajes y está optimizada para cada arquitectura de red específica. También se dispone de la función `VecMDot` que permite realizar el producto con varios vectores con una única comunicación.

Suma, escalado y copia de vectores Estas operaciones (`VecAXPY`, `VecScale`, `VecSet`, `VecCopy`) pueden realizarse totalmente en paralelo sin comunicaciones gracias a cómo se realiza la distribución de los datos en PETSc. También se dispone de la función `VecMAXPY` que es una versión optimizada de `VecAXPY` para la suma escalada de varios vectores.

Todas estas primitivas tienen en común realizar un número de operaciones aritméticas del orden de la cantidad de datos utilizados. Por ejemplo, el cálculo de la norma de un vector supone aproximadamente dos operaciones aritméticas por cada elemento.

Capítulo 4

Implementación en SLEPc

Una vez descrito el algoritmo del cálculo de la exponencial de un matriz utilizado en EXPOKIT y sus fundamentos teóricos, en este capítulo se considerarán los aspectos más relevantes de su implementación en SLEPc.

4.1. El objeto *Matrix Function*

Siguiendo la filosofía de PETSc y SLEPc se ha creado un nuevo objeto que añade la funcionalidad necesaria para el cálculo de funciones matriciales:

`MFN` El objeto *Matrix Function* especifica el problema $x = f(\alpha A)b$ donde f es una función matricial, A una matriz y b un vector. El escalar $\alpha \in \mathbb{R}$ representa un factor de escala.

Como se vio en el capítulo 1 el problema $f(A)b$ se puede resolver mediante distintas técnicas. Por tanto, el objeto `MFN` se ha diseñado para soportar la implementación de distintos *solvers* y, al igual que el objeto `EPS`, proporcionar una interfaz común.

La figura 4.1 muestra como el objeto `MFN` se agrega en la jerarquía de clases de SLEPc.

4.1.1. Interfaz de `MFN`

El ciclo de vida típico de un objeto `MFN` es similar a cualquier objeto de PETSc. Seguidamente se listan las funciones que proporciona la interfaz para trabajar con un objeto `MFN`:

1. Creación del objeto mediante la función `MFNCreate`

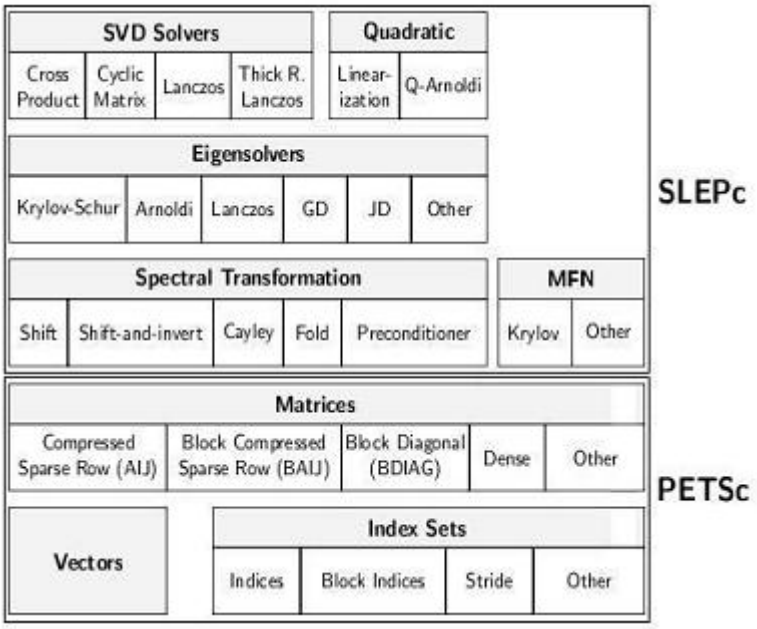

Figura 4.1: El nuevo objeto *Matrix Function* en la jerarquía de SLEPc

2. Selección del método y establecimiento de parámetros con varias funciones:

   - `MFNSetType` selecciona el método de resolución.
   - `MFNSetOperator` especifica la matriz A.
   - `MFNSetFunction` especifica la función f. Por defecto la función exponencial.
   - `MFNSetScaleFactor` especifica el escalar α.
   - `MFNSetDimensions` especifica la dimensión del subespacio utilizado por el *solver*.
   - `MFNSetTolerances` define la tolerancia y el número máximo de iteraciones externas.

   Todos los atributos que no implican matrices o vectores pueden especificarse en tiempo de ejecución mediante la función `MFNSetFromOptions` que procesa los parámetros de la línea de comandos. La tabla 4.1 muestra la correspondencia entre funciones y parámetros de la interfaz `MFN`.

3. Ejecución del *solver* mediante la función `MFNSolve`.

Función	**Parámetro**
`MFNSetType`	`-mfn_type`
`MFNSetFunction`	`-mfn_exp`
`MFNSetScaleFactor`	`-mfn_scale`
`MFNSetDimensions`	`-mfn_ncv`
`MFNSetTolerances`	`-mfn_tol` `-mfn_max_it`

Tabla 4.1: Equivalencia entre funciones y parámetros

4. Recuperación de cierta información:
   - `MFNGetIterationNumber` devuelve el número de iteraciones empleadas por el *solver*.
   - `MFNGetConvergeReason` devuelve la razón por la cual el método fue detenido.
5. Destrucción y liberación de memoria mediante `MFNDestroy`.

A modo de ejemplo, el siguiente fragmento de código crea un objeto `MFN`, establece algunas opciones e invoca al *solver*.

```
70    /*
71       Create MFN context
72    */
73    ierr = MFNCreate(PETSC_COMM_WORLD,&mfn);CHKERRQ(ierr);
74
75    /*
76       Set operators.
77    */
78    ierr = MFNSetOperator(mfn,A);CHKERRQ(ierr);
79    ierr = MFNSetTolerances(mfn,0.15e-07,1000);CHKERRQ(ierr);
80    ierr = MFNSetFunction(mfn,SLEPC_FUNCTION_EXP);CHKERRQ(
         ierr);
81
82    /*
83       Set solver parameters at runtime
84    */
85    ierr = MFNSetFromOptions(mfn);CHKERRQ(ierr);
86
87    /* - - - - - - - - - - - - - - - - - - - - - - - - - - - -
88                          Solve the problem
```

```
89        - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
90
91    ierr = MFNSolve(mfn,b,x);CHKERRQ(ierr);
92
93    /*
94       Optional: Get some information from de solver
95    */
96    ierr = MFNGetIterationNumber(mfn,&its);CHKERRQ(ierr);
97    ierr = PetscPrintf(PETSC_COMM_WORLD, "Number of
         iterations of the method: %D\n", its);CHKERRQ(ierr);
98
99    /*
100      Free work space
101   */
102   ierr = MFNDestroy(&mfn);CHKERRQ(ierr);
```

En tiempo de ejecución se indica el factor de escala y el numero de vectores de la base.

```
$ ejemplo -mfn_scale 10 -mfn_ncv 30
```

4.1.2. La función f

La función f dota de funcionalidad al objeto `MFN`. Aunque el objeto `MFN` está preparado para albergar cualquier función matricial, en esta ocasión se ha implementado la función exponencial de una matriz. Por diseño, según el algoritmo empleado para el cálculo de la función f especificada, el objeto `MFN` utiliza un determinado *solver* para resolver el problema. El algoritmo empleado por EXPOKIT (algoritmo 2.1) se basa fundamentalmente en encontrar el subespacio de Krylov que aproxime $w = \mathrm{e}^{tA}v$. Por tanto, se ha implementado un *solver* basado en el método de Arnoldi.

Cuando la función f se especifica mediante `MFNSetFunction`, el objeto `MFN` cambia su funcionalidad por la del *solver* correspondiente. Es decir, si se especifica la función exponencial como en el ejemplo anterior (ver línea 80), el objeto MFN resolverá el problema mediante el *solver* asociado a la etiqueta `SLEPC_FUNCTION_EXP`. El *solver* asociado a la etiqueta `SLEPC_FUNCTION_EXP` es justamente el que implementa el algoritmo de EXPOKIT.

Desde el punto de vista de la programación orientada a objetos convencional, el objeto MFN se instancia en una de sus subclases dependiendo de la función f. El diagrama de clases UML de la figura 4.2 muestra esta relación.

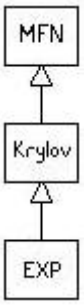

Figura 4.2: Jerarquía de clases MFN

4.2. El objeto *Direct Solver*

El objeto `DS` se utiliza, tradicionalmente, en SLEPc para resolver problemas de dimensión pequeña con métodos directos. Con esta filosofía, se le han añadido nuevas rutinas a su interfaz para dotarle con la funcionalidad necesaria para calcular funciones matriciales.

- `DSSetFunctionMethod` especifica el método para calcular una función matricial.
- `DSComputeFunction` calcula una función matricial.

Opcionalmente, se puede especificar el método en tiempo de ejecución mediante el uso del parámetro `-ds_function_method` en la línea de comandos.

Cuando se invoca la rutina `DSComputeFunction` con el parámetro `SLEPC_FUNCTION_EXP` se calcula la función exponencial utilizando la aproximación racional de Padé, con aproximantes diagonales ($p = q$), combinado con el método de escalado y potenciación mencionaddo en la sección 1.4. La rutina `__PADM` de EXPOKIT utiliza el mismo método. La implementación en SLEPc se realiza mediante llamadas a BLAS y LAPACK.

4.3. Paralelización

La paralelización del algoritmo 2.1 se realiza mediante primitivas de PETSc. La matriz y los vectores del problema se declaran como `Mat` y `Vec` respectivamente, lo que hace la paralelización bastante directa.

El algoritmo 4.1 es una versión esquematizada del algoritmo propuesto por EXPOKIT, donde se puede deducir el uso de las funciones `VecCopy` y `VecNorm` para implementar la primera línea.

Algoritmo 4.1 Calcula w(t) = exp(tA)v

1: $w = v/\|v\|_2$;
2: $t_k = 0$;
3: **while** $t_k < t$ **do**
4: $[V, H, \beta] = \text{arnoldi}(A, w, m)$;
5: **repeat**
6: $\tau = stepsize$;
7: $w = \beta V \exp(\tau H) e_1$
8: $err\_loc = local\ error\ estimate$;
9: **until** $err\_loc \leq \delta tol$;
10: $t_k = t_k + \tau$;
11: **end while**

Un paso importante en el algoritmo es la factorización realizada por el procedimiento de Arnoldi (ver línea 4). Donde los vectores de la base del subespacio ($V_k = [v_1, v_2, \cdots, v_k]$) se declaran como `Vec`, por lo tanto se encuentran repartidos entre los distintos procesadores. La matriz proyectada H_k esta replicada en todos los procesadores, dado que suele ser de pequeño tamaño y no resulta eficiente su paralelización. Además, no está declarada como `Mat` sino como un *array* de elementos consecutivos. De esta forma se pueden aprovechar las funciones de BLAS para operar con ella. La figura 4.3 muestra la distribución de datos realizada.

La implementación del procedimiento de Arnoldi (algoritmo 1.1) se realiza mediante operaciones Matriz-Vector. En el cálculo del vector $w = Av_k$ se emplea la primitiva `MatMult`. Después, en la ortogonalización, se emplean `VecMAXPY` y `VecMDot`. Estas rutinas consideran al conjunto de vectores V_k como una matriz, lo que permite aprovechar mejor la jerarquía de memorias mediante un producto matriz vector. Como estas rutinas obtienen mejores prestaciones que un bucle de operaciones `VecAXPY` o `VecDot`, el método de Gram-Schmidt clásico resulta más competitivo que el Gram-Schmidt modificado debido a que realiza menor número de sincronizaciones en paralelo.

Un vez realizada la factorización, el cálculo de la exponencial de H_k se realiza en todos los procesadores de forma replicada mediante el objeto `DS` interno al *solver*. Seguidamente se emplea la primitiva `VecMAXPY` para actualizar el vector w (ver línea 7) y ,tras comprobar el error cometido, se inicia una nueva iteración.

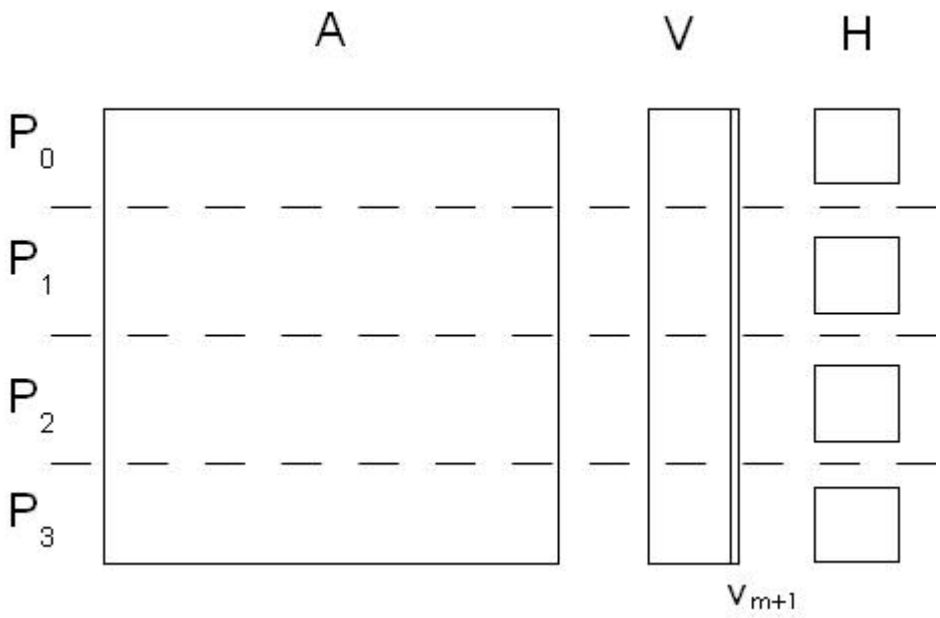

Figura 4.3: Distribución de datos

Capítulo 5

Experimentos y resultados

En este capítulo se describen los experimentos llevados a cabo y sus resultados con el objetivo de comprobar la funcionalidad del algoritmo implementado.

5.1. Máquinas y colecciones de problemas

Las matrices usadas en las pruebas pertenecen a las colecciones de *Matrix Market* [9] y de la *University of Florida Sparse Matrix Collection* [4]. Se han realizado pruebas con matrices reales, generales, simétricas y sus análogas complejas, hermitianas. La tabla 5.1 recoge las matrices empleadas en las pruebas. Todos los problemas proceden de aplicaciones reales.

Matriz	Dimensión	No nulos
AMAZON0302	262.111	1.234.877
BCSTM35	68.902	375.558
CAGE13	445.315	7.479.343
HCIRCUIT	105.676	513.072
MHD1280A	1.280	47.906
ORANI678	2.529	90.158
P2P-GNUTELLA30	36.682	88.328
SOC-EPINIONS1	75.888	508.837

Tabla 5.1: Matrices de las pruebas

Los experimentos se han ejecutado en el clúster Kahan, propiedad de la Universidad Politécnica de Valencia y gestionado conjuntamente por el Departamento de Sistemas Informáticos y Computación y por el Máster

Universitario en Computación Paralela y Distribuida. El clúster consta de 6 nodos biprocesadores AMD Opteron 16 Core 6272 a 2.1GHz conectados mediante una red InfiniBand QDR 4X (40Gbps, tasa efectiva 32Gbps). Formando, en total, una máquina con 192 núcleos de procesamiento y 192GB de memoria.

5.2. Evaluación numérica

Con el fin de comprobar la corrección del resultado obtenido por el *solver* se ha realizado una prueba para medir el error cometido

$$\|x - \exp(tA)b\|_2$$

donde x es la aproximación obtenida por el *solver* y $\exp(tA)b$ es la matriz exponencial de tA, calculada mediante el método de Padé, multiplicada por el vector b.

La figura 5.1 muestra los resultados obtenidos para distintas tolerancias. En el experimento se ha utilizado la matriz ORANI678, que acompaña al paquete EXPOKIT con el mismo propósito, un tamaño de la base de 30 vectores y 1000 iteraciones como máximo. El vector b es el vector unidad $[1, 1, \cdots, 1]^T$ y t igual a 10. La tolerancia se ha ido variando en las distintas ejecuciones mediante el parámetro `-mfn_tol`.

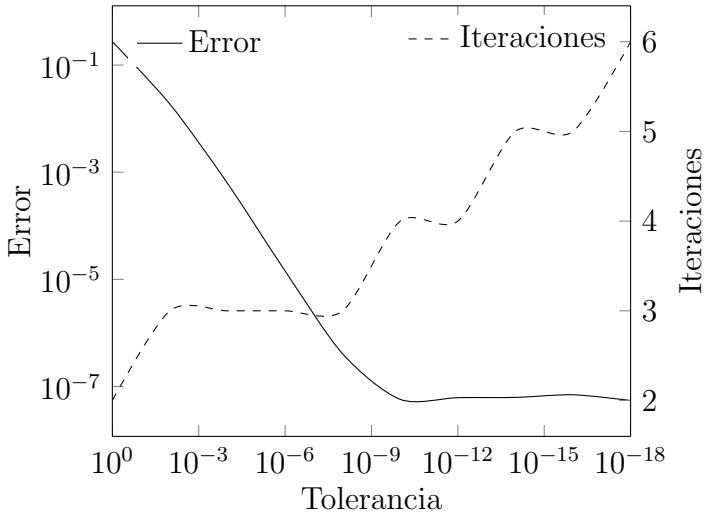

Figura 5.1: Evaluación numérica

Los resultados muestran que el error cometido por el *solver* se estanca a cierta tolerancia y que el número de iteraciones para resolver el problema

incrementa. Dado que no se va a obtener mejor solución y que el coste del algoritmo aumenta con el número de iteraciones, un error aceptable es el cometido con la tolerancia 10^{-8}

$$\|x - \exp(tA)b\|_2 = 4,052 \times 10^{-7}$$

en el que el *solver* emplea 3 iteraciones y 0,0290 segundos, frente a los 290 segundos que emplea el método de Padé.

5.3. Evaluación de las prestaciones en paralelo

Para evaluar las prestaciones paralelas, se han realizado dos experimentos, uno para medir el *speed-up* y otro para comprobar la escalabilidad del método.

5.3.1. *Speed-up*

El principal índice de las prestaciones de los algoritmos paralelos es el *speed-up*, es decir, la aceleración que se obtiene al utilizar varios procesadores con respecto a la versión secuencial. Idealmente, el tiempo de ejecución se debe reducir en un factor p si se utilizan p procesadores. El *speed-up* se calcula como

$$S_p = \frac{T_s}{T_p},$$

donde T_s es el tiempo en secuencial del algoritmo más rápido y T_p es el tiempo en paralelo con p procesadores.

En esta prueba se obtiene el *speed-up* del método configurando el *solver* con la matriz HCIRCUIT, t igual a 1, una tolerancia de 10^{-8} y manteniendo el número de iteraciones máximo a 1000. Esta matriz es bastante más grande que la anterior, con una dimensión de 105.676 y un total de 513.072 elementos no nulos, proveniente de la simulación de circuitos semiconductores.

Se han utilizado opciones de afinidad, incorporadas en OpenMPI, para aprovechar mejor la jerarquía de memorias de los nodos de Kahan. Concretamente, se han lanzado las pruebas especificando los parámetros `-bysocket` y `-cpus-per-proc 8` y se ha comprobado una clara mejoría en el rendimiento. En la figura 5.2 se muestran los resultados obtenidos. La tabla 5.2 recoge los valores obtenidos para el *speed-up* y la eficiencia ejecutando la prueba con las opciones de afinidad.

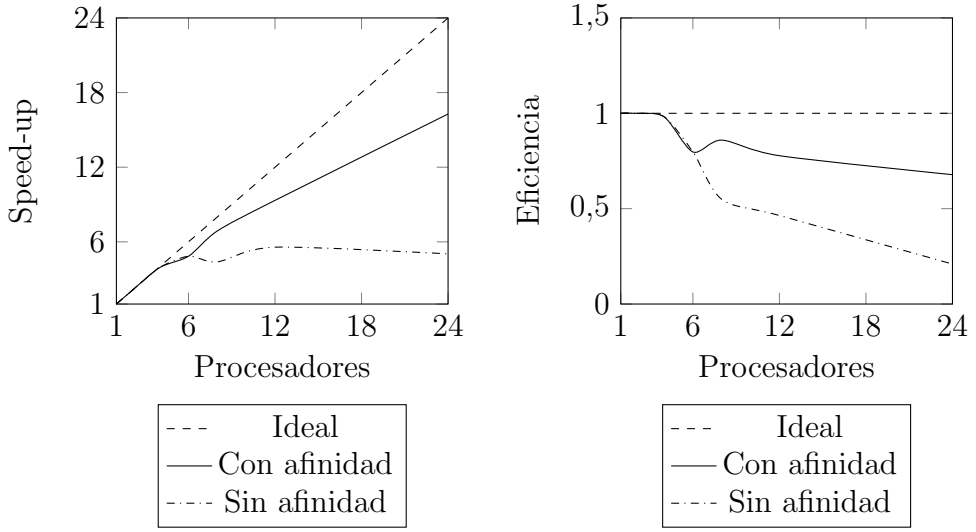

Figura 5.2: *Speed-up* y eficiencia con la matriz HCIRCUIT

Proc.	Tiempo(seg.)	*Speed-up*	Eficiencia
1	2,5882	1	1
2	1,2941	2	1
4	0,6590	3,9274	0,9818
6	0,5346	4,8413	0,7975
8	0,5892	6,8758	0,8594
12	0,4646	9,3335	0,7777
24	0,5143	16,2779	0,6782

Tabla 5.2: *Speed-up* y eficiencia con la matriz HCIRCUIT

5.3.2. *Speed-up* escalado

Existe un límite en las prestaciones paralelas al utilizar problemas de tamaño fijo, debido a que el tiempo empleado en la parte del algoritmo paralelizable disminuye al repartirlo entre un número mayor de procesadores, mientras que el tiempo empleado en la parte secuencial se mantiene constante. Para evitar este efecto se ha recurrido a un problema tridiagonal de dimensión proporcional al número de procesadores, con elementos aleatorios entre 0 y 1. Asumiendo que el coste computacional crece de forma lineal con la dimensión, se define el *speed-up* escalado para p procesadores como

$$S = \frac{T_s \times p}{T_p}$$

En este prueba se ha construido una matriz pseudo-aleatoria de dimensión 1.000.000 $\times p$ con la que se ejecuta el *solver* con diferente número de procesadores. El resto de parámetros se mantienen iguales a la prueba anterior. En la figura 5.3 se puede ver que el método escala perfectamente.

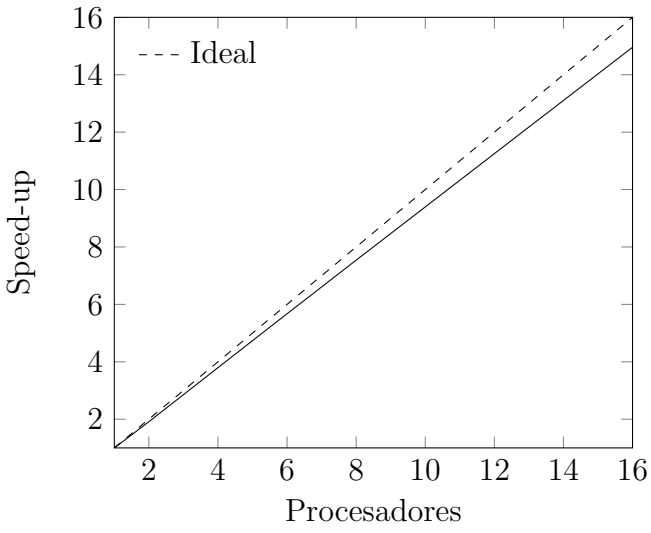

Figura 5.3: *Speed-up* escalado

5.4. Paralelización

Aprovechando las facilidades de introspección que incorpora la librería PETSc, se ha realizado un experimento que mide el tiempo empleado por cada una de las partes más relevantes del algoritmo 4.1. Estas son: la factorización de Arnoldi, el cálculo de la exponencial de la matriz H mediante el método de Padé y la actualización del vector w, como se vio en la sección 4.3.

En este prueba se ha utilizado la matriz CAGE13 de dimensión 445.315 y 7.479.343 elementos no nulos. El resto de parámetros del *solver* se han establecido a los valores por defecto: 30 vectores para la base de Krylov, tolerancia igual a 10^{-8}, el factor de escala $t = 1$ y 1000 iteraciones como máximo. La figura 5.4 muestra los resultados obtenidos.

La mayor parte del tiempo empleado por el *solver* para resolver el problema (`MFNSolve`, arriba a la izquierda) es consumido por el procedimiento de Arnoldi (`MFNBasicArnoldi`, arriba a la derecha). También se puede observar que las partes paralelizadas del algoritmo, tanto el procedimiento de

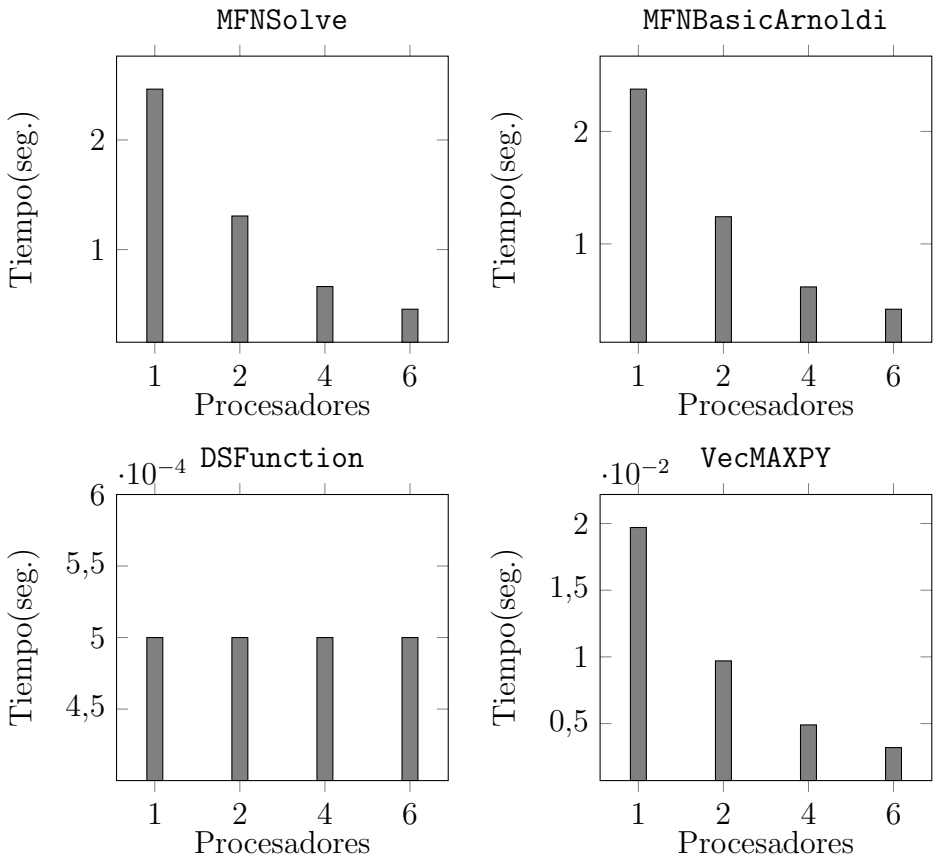

Figura 5.4: Tiempos con la matriz CAGE13

Arnoldi como la actualización de w (`VecMAXPY`, abajo a la derecha) escalan muy bien.

5.5. Caso de uso: Propagador Crank-Nicolson

La ecuación de Schrödinger dependiente del tiempo

$$i\frac{d}{dt}\Psi(t) = H\Psi(t) \tag{5.1}$$

puede ser numéricamente resuelta para cada tiempo t_n como

$$\Psi(t_n) = e^{-iH(t_n)\triangle t}\Psi(t_{n-1}) \tag{5.2}$$

donde $\triangle t = t_n - t_{n-1}$. Calcular la exponencial de la matriz $H(t)$ para cada tiempo es costoso. En su lugar, se puede aproximar la exponencial como:

$$e^{-iH(t)\triangle t} = \left(1 + iH(t)\frac{\triangle t}{2}\right)^{-1}\left(1 - iH(t)\frac{\triangle t}{2}\right)$$

La expresión anterior se obtiene simplemente aproximando las derivadas de primer orden de la ecuación (5.1) como se indica a continuación para el caso particular del átomo de Helio.

El operador Hamiltoniano contiene las componentes independientes del tiempo (H_0) y las dependientes del tiempo (V_t):

$$H = H_0 + V_t$$

donde, H_0 tiene los términos del primer y segundo electrón y

$$V_t = \frac{1}{r_{12}} + \mu E(t)$$

Tal partición es conveniente, ya que los términos del primer electrón son bloques en la diagonal de la matriz y los términos de V_t son diagonales fuera de la diagonal a bloques.

Entonces, la aproximación de las derivadas de la ecuación (5.1) es:

$$i\left[\frac{\Psi(t_{n+1}) - \Psi(t_{n-1})}{2\triangle t}\right] = \frac{H(t)\Psi_{n+1} + H(t)\Psi_{n-1}}{2}$$

Después de reorganizar los términos nos encontramos con la siguiente expresión, que hay que resolver para cada paso de tiempo.

$$(1 + i\triangle tH(t))\Psi(t_{n+1}) = (1 - i\triangle tH(t))\Psi(t_{n-1})$$

Para ello, debe resolverse un sistema de ecuaciones ($Ax = b$) donde el lado derecho es $b = (1 - i\triangle tH(t))\Psi(t_{n-1})$ y x la función de onda obtenida para cada nuevo paso de tiempo $\Psi(t_{n+1})$.

Con el *solver* implementado en el nuevo componente de SLEPc (`MFN`) se podría calcular directamente la expresión (5.2) multiplicando, previamente, la matriz Hamiltoniana por $-i$. Como demostración, se ha realizado una prueba con H independiente del tiempo. La figura 5.5 muestra la estructura de la matriz Hamiltoniana de dimensión 37.636 y 1.247.808 elementos no nulos.

En esta prueba se ha obtenido el *speed-up* configurando el *solver* con la matriz Hamiltoniana, una base de 30 vectores, t igual 0,5, una tolerancia de 10^{-8} y 1000 iteraciones como máximo. La figura 5.6 muestra la gráfica del *speed-up* y la tabla 5.3 recoge el tiempo de ejecución para los diferentes procesadores.

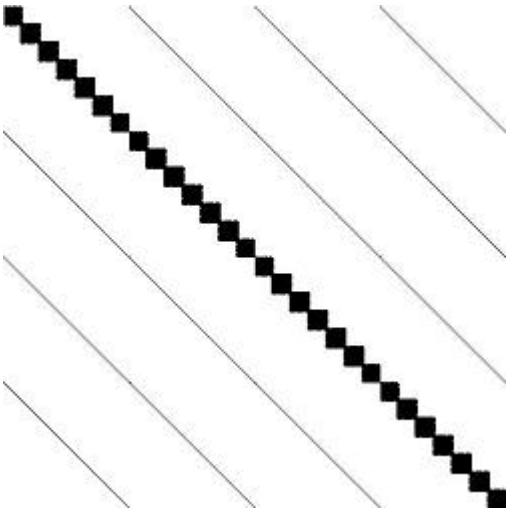

Figura 5.5: Estructura de la matriz Hamiltoniana para un átomo de Helio

Proc.	Tiempo(seg.)	*Speed-up*
1	21,278	1
2	10,733	1,9824
4	5,6010	3,7989
6	3,8624	5,5090
8	2,9358	7,2477
12	2,0493	10,383
24	1,1308	18,816

Tabla 5.3: Tiempos con la matriz Hamiltoniana

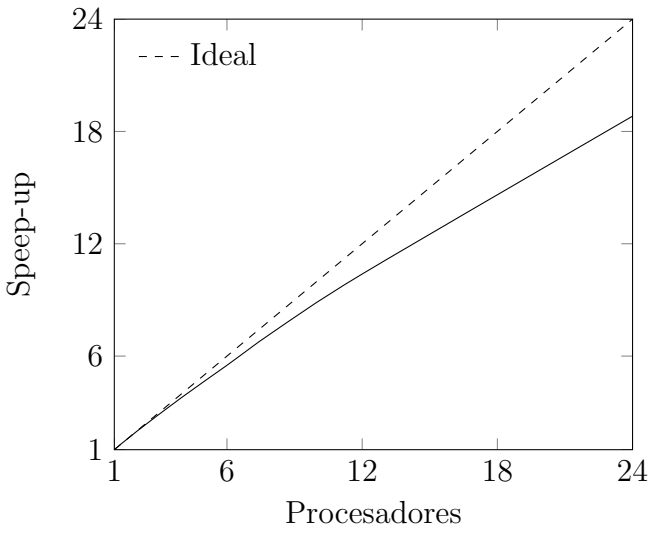

Figura 5.6: *Speed-up* con la matriz Hamiltoniana

Capítulo 6

Conclusiones y trabajo futuro

Tras una introducción al estado del arte de los métodos utilizados para resolver funciones matriciales, en este trabajo se ha diseñado un nuevo componente para dotar de dicha funcionalidad a la librería SLEPc. A modo de ejemplo, se ha implementado la función exponencial de una matriz partiendo del algoritmo propuesto por EXPOKIT. Esto ha supuesto familializarse con las librerías PETSc y SLEPc, conocer su estructura, y realizar un estudio del software EXPOKIT.

La implementación realizada se ha validado mediante una batería de pruebas con matrices procedentes de aplicaciones reales. Las prestaciones paralelas se han medido en una máquina tipo clúster comprobando una buena escalabilidad. El trabajo termina con el estudio de un caso de uso, resolviendo satisfactoriamente la ecuación de Schrödinger para un átomo de Helio. Dado los prometedores resultados y las facilidades que incorpora este nuevo componente a la librería, se incluirá en la próxima versión de SLEPc de modo experimental.

Como trabajo futuro se plantean diferentes posibilidades. Por un lado, se podría implementar el caso no homogéneo ($u \neq 0$) del sistema lineal EDO planteado en el capítulo 2. La rutina `__PHIV` de EXPOKIT soluciona el problema $w = \exp(tA)v + t\varphi(tA)u$. Por otro lado, se podrían implementar otras funciones matriciales como la función logarítmica o la función signo, que junto a la función exponencial son las más usuales.

Agradecimientos

Las matrices utilizadas para los resultados del apartado 5.5 fueron proporciadas por Alicia Palacios, del grupo *Computations in Atomic and Molecular Physics of Unbounded Systems* de la Universidad Autónoma de Madrid (http://campusys.qui.uam.es).

Bibliografía

[1] S Balay, J Brown, K Buschelman, V Eijkhout, W Gropp, D Kaushik, M Knepley, L Curfman McInnes, B Smith, and H Zhang. Petsc users manual revision 3.4. 2013.

[2] Carmen Campos, José E Román, Eloy Romero, and Andrés Tomás. Slepc users manual. Technical report, Tech. Rep. DSIC-II/24/02-Revision 3.3, D. Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, 2012.

[3] Expokit web page. www.maths.uq.edu.au/expokit/.

[4] University of florida sparse matrix collection web page. www.cise.ufl.edu/research/sparse/matrices/.

[5] Message Passing Interface Forum. *MPI: A message-passing interface standard*. 1995.

[6] V Hernandez, JE Roman, and A Tomas. Parallel arnoldi eigensolvers with enhanced scalability via global communications rearrangement. *Parallel Computing*, 33(7):521–540, 2007.

[7] Nicholas J. Higham. *Functions of Matrices: Theory and Computation*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2008.

[8] Nicholas J. Higham and Awad H. Al-Mohy. Computing matrix functions. *Acta Numerica*, 19:159 –208, 2010.

[9] Matrix market web page. math.nist.gov/MatrixMarket/.

[10] Cleve Moler and Charles Van Loan. Nineteen dubious ways to compute the exponential of a matrix. *SIAM Review*, 20:801–836, 1978.

[11] Cleve Moler and Charles Van Loan. Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later. *SIAM review*, 45(1):3–49, 2003.

[12] Petsc web page. www.mcs.anl.gov/petsc/.

[13] Roger B Sidje. Expokit: a software package for computing matrix exponentials. *ACM Transactions on Mathematical Software (TOMS)*, 24(1):130–156, 1998.

[14] Slepc web page. www.grycap.upv.es/slepc/.

[15] A. Tomas y V.Vidal V. Hernandez, J. E. Roman. Arnoldi methods in slepc. Technical report, STR-4, Universidad Politécnica de Valencia, 2006.