Risultati Numerici sulla Risoluzione di PQEP

Alessio Marchetti

1 Introduzione

In questa relazione si presenteranno i codici relativi agli algoritmi descritti nell'articolo [1]. Si andranno a discutere i risultati e a confrontarli con alcune funzioni presenti in Matlab. L'obiettivo è quello di determinare autovalori e autovettori del problema quadratico

$$P(\lambda) = \lambda^2 A^{\mathsf{T}} + \lambda Q + A$$

dove A e Q sono matrici quadrate complesse e presentano una particolare struttura a blocchi e Q è simmetrica.

Tutti i codici presentati sono stati testati con Octave versione 6.1.0. I grafici e i risultati riportati sono stati prodotti con il medesimo software.

2 Costruzione delle matrici

Per costruire le matrici A e Q si segue la struttura indicata in [2]. L'implementazione è presentata nel seguente codice.

```
D = eye(m);
U = diag(ones(m-1,1),  1);
L = diag(ones(m-1,1), -1);
T = zeros(m);
T(1,m)=1;

Kt = kron(D,K0) + kron(U,K1.') + kron(L,K1);
Mt = kron(D,M0) + kron(U,M1.') + kron(L,M1);
Kc = kron(T, K1);
Mc = kron(T, M1);
Dt = c1 * Mt + c2 * Kt;
Dc = c1 * Mc + c2 * Kc;

Q = Kt + i * omega * Dt - omega * omega * Mt;
A = Kc + i * omega * Dc - omega * omega * Mc;
```

Poichè l'esecuzione è molto rapida e va eseguita una singola volta si è preferita la leggibilità rispetto all'ottimizzazione. Affinchè queste istruzioni funzionino è necessario avere nello scope il parametro `k` e le matrici `K0`, `K1`, `M0` e `M1`. Per le verifiche nel seguito della relazione, si sono state usate sia matrici

derivanti dallo studio reale della struttura delle rotaie, e verranno chiamate matrici dei treni, sia delle matrici casuali generate con `rand`. Poichè si richiede che `K0` e `M0` siano simmetriche si aggiungono anche le due seguenti istruzioni che assicurano tale proprietà.

```
K0 = 0.5 * (K0 + K0.');
M0 = 0.5 * (M0 + M0.');
```

3 Doubling Algorithm

In questa sezione si presenta un metodo per risolvere l'equazione

$$X + A^{\mathsf{T}} X A = Q$$

dove A e Q sono delle matrici complesse $n \times n$ date e Q è simmetrica. Il metodo risolutivo, chiamato Doubling Algorithm, si basa sul seguente schema iterativo:

$$\begin{aligned} A_{i+1} &= A_i (X_i - Y_i)^{-1} A_i^{\mathsf{T}} \\ X_{i+1} &= X_i - A_i^{\mathsf{T}} (X_i - Y_i)^{-1} A_i \\ Y_{i+1} &= Y_i + A_i (X_i - Y_i)^{-1} A_i^{\mathsf{T}} \end{aligned}$$

e viene inizializato con

$$A_0 = A; \qquad X_0 = Q; \qquad Y_0 = 0.$$

Il ciclo si ferma quando l'errore relativo

$$\frac{\|X_{i+1} - X_i\|}{\|X_i\|} \leq \varepsilon_T$$

risulta minore di un valore di tolleranza ε_T che verrà preso vicino all'epsilon di macchina.

In seguito si riporta il codice della relativa implementazione.

```
function XX1 =  doubling(A, Q, rtol=1e-16)
    AA1 = A;
    XX1 = Q;
    YY1 = zeros(size(A));
    while 1
        AA = AA1;
        XX = XX1;
        YY = YY1;
        [L,U,P] = lu(XX - YY);
        SOL_AA = U \ (L \ (P*AA));
        AA1 = AA * SOL_AA;
        XX1 = XX - AA.' * SOL_AA;
        YY1 = YY + AA * (U \ (L \ (P*AA.')));
        err = norm(XX1-XX) / norm(XX);
        if  err < rtol
            break;
        end
    end
end
```

In un'ottica di ottimizzazione, non viene mai calcolata la matrice $(X_i - Y_i)^{-1}$, ma vengono risolti dei sistemi lineari della forma $(X_i - Y_i)Z = \bar{A}$, dove $\bar{A}$ può essere A oppure A^{T}. Inoltre per fare ciò la fattorizzazione PLU di $X_i - Y_i$ viene calcolata una singola volta per risolvere entrambe le equazioni.

Passiamo ora all'analisi della convergenza del metodo. Per questi scopi si utilizzeranno delle matrici H_0 e H_1 come descritte sopra, con $k = 303$ e $m = 19$. La velocità di convergenza dipende dal valore

$$\hat{\gamma} = \rho(\hat{\Phi}^{-1} H_1)$$

dove ρ indica il raggio spettrale e $\hat{\Phi}$ è la soluzione trovata. Nel caso considerato i valori, al variare di alcuni valori di ω sono descritti nella seguente tabella.

ω	100	1000	3000	5000
$\hat{\gamma}$	0.996	0.987	0.979	0.973

Il codice utilizzato è il seguente.

```
load new_mtx_K_M.mat;
k = 303;
m = 19;
omegas = [100 1000 3000 5000];
rtol = 1e-16;

c1 = 0.8;
c2 = 0.2;

for ii = 1:4
    omega = omegas(ii);
    H1 = K1 + i * omega * (c1 * M1 + c2 * K1) - omega * omega
        * M1;
    H0 = K0 + i * omega * (c1 * M0 + c2 * K0) - omega * omega
        * M0;

    out = [];

    AA1 = H1;
    XX1 = H0;
    YY1 = zeros(k,k);
    while 1
        AA = AA1;
        XX = XX1;
        YY = YY1;
        [L,U,P] = lu(XX - YY);
        SOL_AA = U \ (L \ (P*AA));
        AA1 = AA * SOL_AA;
        XX1 = XX - AA.' * SOL_AA;
        YY1 = YY + AA * (U \ (L \ P*AA.'));
        err = norm(XX1-XX) / norm(XX);
        if err > 0
            out = [out err];
        end
        if  err < rtol
            break;
        end
    end
    gamma = abs(eigs(inv(XX1)*H1, 1));
```

```
38      disp(strcat(num2str(omega), ': ', num2str(gamma)));
39      semilogy(out, strcat(";", num2str(omega), ";"));
40      hold on;
41  end
42  hold off;
```

Qui `load new_mtx_K_M.mat` carica i dati delle matrici `K0`, `K1`, `M0`, `M1`.

Infine si riporta in 1 il grafico dell'errore in funzione del numero di passi dell'algoritmo di doubling.

Si noti che al crescere di ω il valore $\hat{\gamma}$ decresce e quindi la velocità di convergenza aumenta.

Passiamo quindi agli errori prodotti dall'algoritmo. Si definisce l'errore di una soluzione Φ di $X + A^{\mathsf{T}} X^{-1} A = Q$ come

$$\varepsilon = \frac{\left\| \Phi + A^{\mathsf{T}} \Phi^{-1} A - Q \right\|}{\|\Phi\| + \|A\|^2 \|\Phi^{-1}\| + \|Q\|}$$

In tabella si riportano tali valori di ε.

ω	100	1000	3000	5000
ε	1.93e-17	2.17e-17	2.70e-17	1.66e-17

Essi sono dell'ordine dell'epsilon di macchina. I risultati cambiano drasticamente se al posto di utilizzare le matrici dei treni derivanti dall'effettiva struttura dei binari si prendono matrici casuali con le proprietà richieste. A titolo di esempio si riporta una tabella analoga alla precedente con matrici di uguali dimensioni.

ω	100	1000	3000	5000
ε	3.48e-08	8.50e-05	1.17e-3	4.64e-3

Risulta evidente il calo della qualità delle soluzioni.

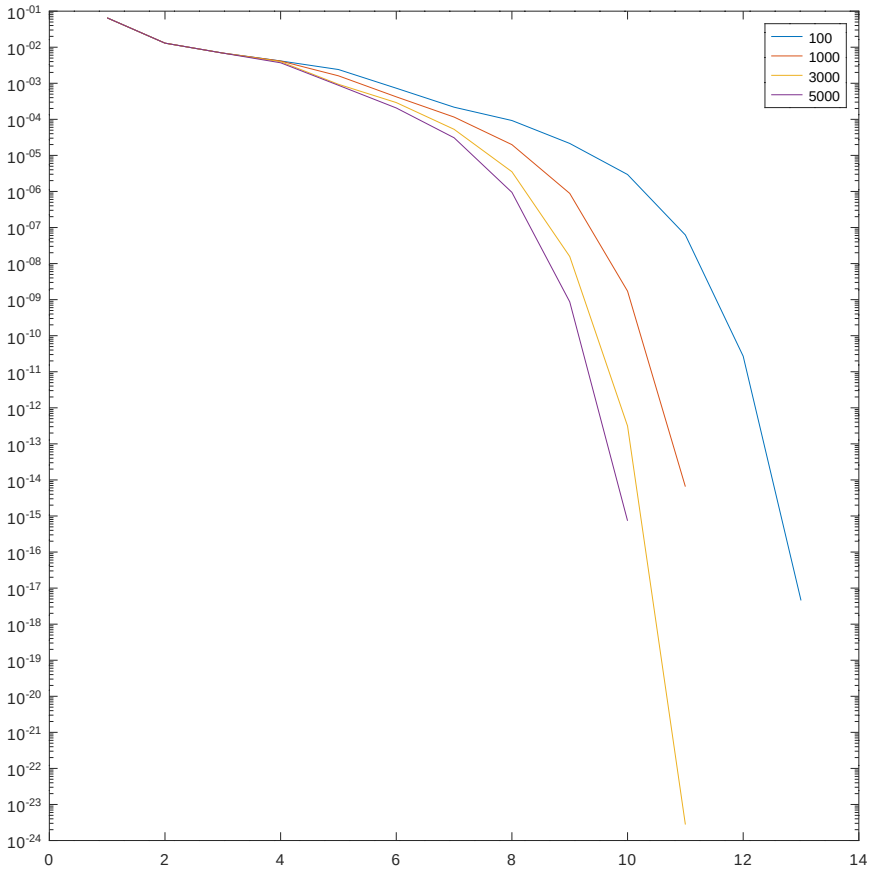

Figura 1: Convergenza dell'algoritmo di doubling.

4 Risoluzione di $\hat{P}$

In questa sezione si vogliono calcolare autovalori e autovettori del polinomio

$$\hat{P}(\lambda) = \lambda^2 H_1^{\mathsf{T}} + \lambda H_0 + H_1$$

dove H_0 e H_1 sono matrici complesse $k \times k$ e H_0 è simmetrica. Per fare ciò si passa per la scomposizione

$$\hat{P}(\lambda) = (\lambda H_1^{\mathsf{T}} + \Phi)\Phi^{-1}(\lambda\Phi + H_1)$$

in cui Φ è una soluzione dell'equazione $X + H_1^{\mathsf{T}} X^{-1} H_1 = H_0$, che può essere risolta con il doubling algorithm. Per la simmetria di Φ, si ha che gli autovalori di $\lambda\Phi + H_1$ sono esattamente i reciproci di quelli di $\lambda H_1^{\mathsf{T}} + \Phi$. Vale che gli autovettori destri di $\lambda\Phi + H_1$ sono anche autovettori destri di $\hat{P}$. Inoltre se il vettore v è un autovettore destro di $\lambda H_1^{\mathsf{T}} + \Phi$, allora un vettore w tale che $v = \Phi^{-1}(\lambda\Phi + H_1)w$ è un autovettore destro di $\hat{P}$. Si noti anche che gli autovettori destri di $\lambda H_1^{\mathsf{T}} + \Phi$ sono gli autovettori sinistri di $\lambda\Phi + H_1$. Un discorso analogo si può fare per gli autovettori sinistri.

Quindi tutti e soli gli autovettori destri di $\hat{P}$ sono gli autovettori destri di $\lambda\Phi + H_1$ e i vettori w tali che $\Phi v = (\lambda\Phi + H_1)w$ dove v è un'autovettore sinistro di $\lambda\Phi + H_1$.

Per calcolare gli autovettori di $\lambda\Phi + H_1$, si passa per l'algoritmo QZ, che trova due matrici unitarie U e V tali che

$$UH_1V = G \qquad \text{e} \qquad U\Phi V = T$$

con G e T triangolari superiore. Allora per calcolare w bisogna risolvere $U^{\mathsf{H}}(\lambda G + T)V^{\mathsf{H}}w = U^{\mathsf{H}}TV^{\mathsf{H}}v$ e quindi $(\lambda G + T)u = TV^{\mathsf{H}}v$ e poi $V^{\mathsf{H}}w = u$. Questo metodo è vantaggioso perché la risoluzione diretta costerebbe $O(k^3)$, mentre nel modo presentato, sfruttando la struttura di T e G, il costo è di $O(k^2)$ per singolo autovettore.

Il seguente codice implementa l'algoritmo sopra descritto. Si noti che si è preferito scomporre $-\Phi$ invece che Φ per ottenere direttamente gli autovettori con il segno corretto. Questo rende necessario un cambio di segno nelle formule che coinvolgono T.

```
function [lam, vecr, vecl] = sda_fast(H0, H1)
    PHI = doubling(H1, H0);
    [G, T, U, V, vecr, vecl] = qz(H1, -PHI);
    lam = diag(G) ./ diag(T);
    vecl = conj(vecl);
    vecl = T * V' * vecl;
    kk = size(vecl)(2);
    for ii = 1:kk
        vecl(:, ii) = (-lam(ii) * G + T) \ vecl(:, ii);
    end
    vecl = V * vecl;
end
```

Qui la funzione ritorna un vettore di autovalori `lam` e i relativi autovettori in `vecr`. Inoltre i reciproci dei valori in `lam` sono gli autovalori relativi agli autovalori in `vecl`.

Si presenta anche un codice simile al precendente che però risolve direttamente il sistema senza passare per la fattorizzazione QZ.

```
function [lam, vecr, vecl] = sda_slow(H0, H1)
    PHI = doubling(H1, H0);
    [vecr, lam, vecl] = eig(H1, -PHI, 'vector');
    vecl = conj(vecl);
    kk = size(vecl)(2);
    for ii = 1:kk
        vecl(:, ii) = (PHI + lam(ii) * H1) \ (PHI * vecl(:,
            ii));
    end
end
```

Per valutare la bontà delle soluzioni trovate, introduciamo il residuo relativo di una coppia di autovalore e autovettore (μ, v) definito come

$$RRes = \frac{\left\|(\mu^2 H_1^{\mathsf{T}} + \mu H_0 + H1)v\right\|}{(|\mu|^2 \|H_1\| + |\mu| \|H_0\| + \|H_1\|) \|v\|}$$

Dove per i vettori si prende la norma 2 e per le matrici si prende la norma di Frobenius. In figura 2 si mostrano i residui relativi al problema con i dati delle rotaie risolti con `sda_fast`. In figura 3 si presentano dati analoghi relativi a `sda_slow`.

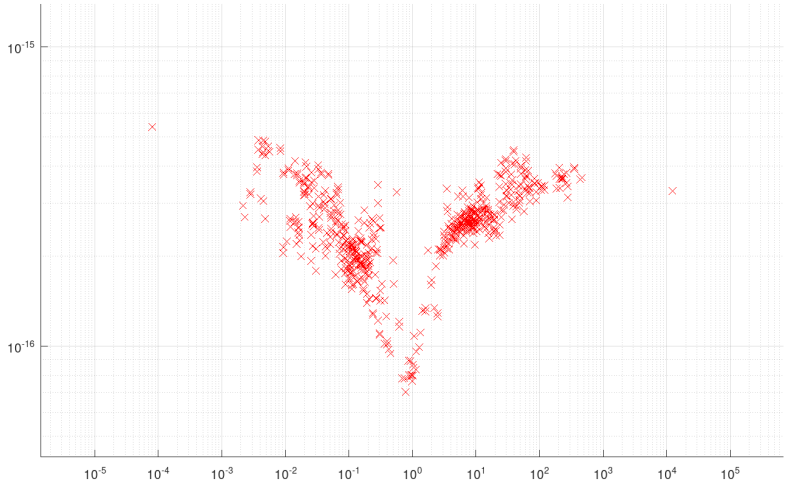

Figura 2: Residui relativi con `sda_fast`.

I dati coincidono sugli autovalori dentro il cerchio unitario, mentre i risultati dell'algoritmo lento sono leggermente migliori al di fuori di esso. In ogni caso il residuo relativo risulta essere dell'ordine dell'epsilon di macchina, e i risultati sono in accordo con quelli trovati nell'articolo.

Il seguente script calcola i residui per lo stesso problema $\hat{P}$ risolto con la funzione `polyeig`.

```
outx2 = [];
outy2 = [];
```

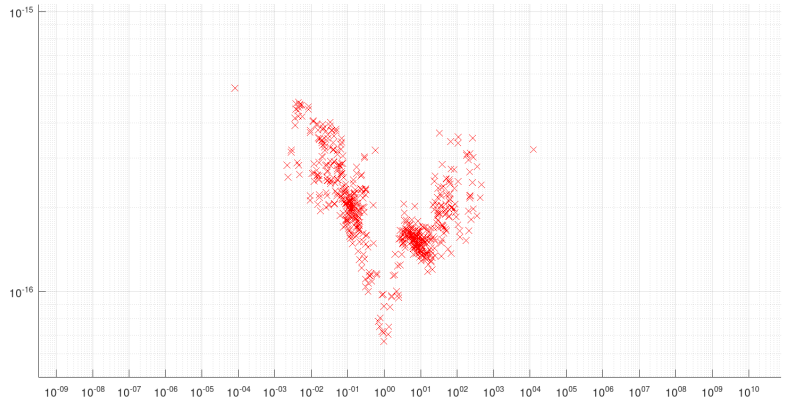

Figura 3: Residui relativi con `sda_slow`.

```
3
4  build_quick;
5  [vec2, lam2] = polyeig(H1, H0, H1.');
6  na = norm(H1, 'fro');
7  nq = norm(H0, 'fro');
8  for ii = 1:size(lam2)(1)
9      mu = lam2(ii);
10     v = vec2(:,ii);
11     num = norm(mu*mu*H1.'*v + mu*H0*v + H1*v);
12     mod = abs(mu);
13     den = mod*mod*na + mod*nq + na;
14     den = den * norm(v);
15     rres = num/den;
16     outx2 = [outx2 mod];
17     outy2 = [outy2 rres];
18 end
19 plt2 = scatter(outx2, outy2, 'r', 'x');
20 set(gca,'Xscale','log', 'Yscale', 'log');
```

La funzione `build_quick` si occupa di costruire le matrici `H0` e `H1` come nei programmi precedenti. I risultati sono mostrati in figura 4.

Essi sono ben peggiori di quelli calcolati con SDA e inoltre non rispettano la palindromia del problema, ossia gli autovalori non compaiono in coppie del tipo λ e $1/\lambda$.

Infine in figura 5 si confrontano `polyeig` e `sda_fast` su matrici generate casualmente. L'algoritmo qui presentato continua a comportarsi meglio di `polyeig` su autovalori piccoli, mentre risulta problematico sugli autovalori di modulo maggiore di 1, calcolati passando per un'ulteriore equazione lineare rispetto agli altri.

Per l'analisi del tempo di calcolo, nella tabella che segue sono riportati i tempi di esecuzione in secondi dei tre algoritmi `sda_fast`, `sda_slow` e `polyeig` con input le matrici dei treni.

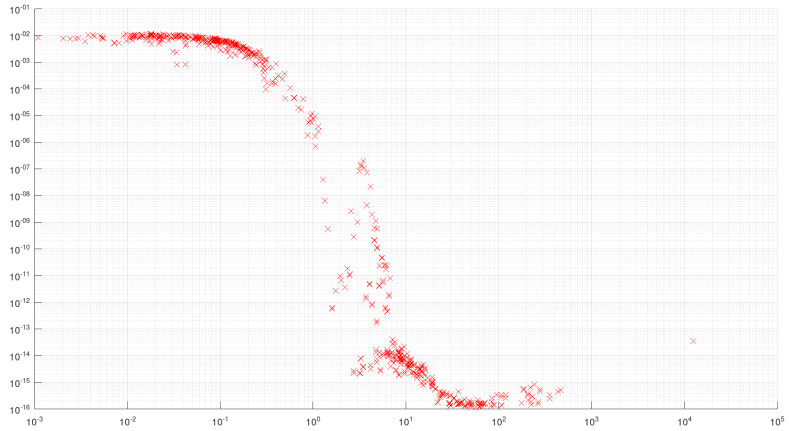

Figura 4: Residui relativi con `polyeig`.

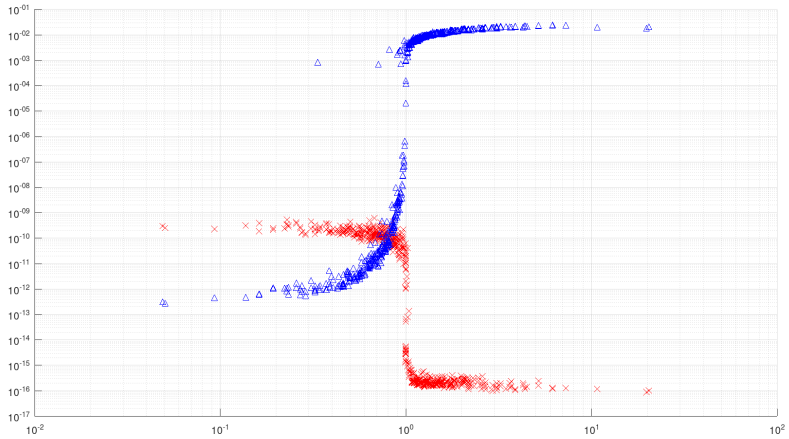

Figura 5: Residui relativi con `polyeig` (in rosso) e con `sda_fast` (in blu).

`sda_fast`	7.85	7.90	7.79	7.90	7.84
`sda_slow`	12.86	12.80	12.82	12.75	12.78
`polyeig`	3.27	3.37	3.31	3.36	3.31

5 Risoluzione di P

Avendo a disposizione tutte le coppie di autovalori e autovettori di $\hat{P}$. Per fare questo supponiamo di avere la coppia (μ, v) per $\hat{P}$, allora

$$\left(\mu^m, \begin{pmatrix} v \\ \mu v \\ \vdots \\ \mu^{m-1} v \end{pmatrix}\right)$$

è una coppia per P.

Tale procedimento è stato implementato nel seguente codice.

```
function [lam, vec] = full_spec(H0, H1, m)
    k = size(H0)(1);
    [lam0, vecr0, vecl0] = sda_3_left(H0, H1);
    lam = zeros(1, 2*k);
    vec = zeros(m*k, 2*k);
    tmp = (0:m-1)';
    for jj = 1:k
        lam(jj) = lam0(jj)^m;
        lam(k+jj) = 1/lam(jj);
        tmp2 = lam0(jj) .^ tmp;
        vec(:, jj) = kron(tmp2, vecr0(:, jj));
        tmp2 = 1./ tmp2;
        vec(:, k+jj) = kron(tmp2, vecl0(:, jj));
    end
end
```

Come prima, per verificare che l'errore sui risultati sia piccolo si fa riferimento al residuo relativo per la coppia (μ, v).

$$RRes = \frac{\left\|(\mu^2 A^{\mathsf{T}} + \mu Q + A)v\right\|}{(|\mu|^2 \|A\| + |\mu| \|Q\| + \|A\|) \|v\|}.$$

Considerata però la struttura particolare di Q e A è più appropriato anche considerare il residuo nella forma

$$RRes_{new} = \frac{\left\|(\mu^2 A^{\mathsf{T}} + \mu Q + A)v\right\|}{|\mu|^2 \|H1\| \|y_1\| + |\mu| \|Q\| \|y\| + \|H1\| \|y_m\|}.$$

dove il vettore y è stato partizionato in $y^{\mathsf{T}} = (y_1^{\mathsf{T}} \ \cdots \ y_m^{\mathsf{T}})$.

In figura 6 si mostrano i risultati dei calcoli dei residui, sia strutturati che non strutturati. Per questo calcolo è stato utile utilizzare la forma sparsa delle matrici A e Q, cosa che ha velocizzato di parecchio l'esecuzione del programma.

Di nuovo, tutti gli errori sono vicini all'epsilon di macchina. Infine si riporta lo stesso grafico relativo alle matrici generate casualmente in figura 7. Si nota il gap tra gli autovalori dentro il cerhio unitario e quelli fuori.

Per un ultimo confronto del metodo con `polyeig` si vorrebbe produrre un tale grafico anche per questo algoritmo. Questo tuttavia non mi è possibile per i lunghi tempi di esecuzione. Anche supponendo di disporre di un

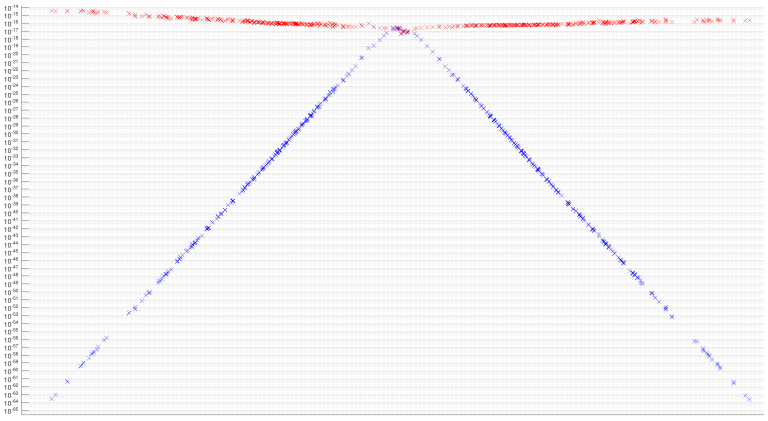

Figura 6: $RRes$ (in blu) e $RRes_{new}$ (in rosso) per matrici del treno.

sufficiente tempo e potenza di calcolo, probabilmente la qualità delle soluzioni risulterebbe comunque molto bassa. A titolo di esempio in figura 8 si riportano i residui di `polyeig` per un valore $m = 3$. Già a queste dimensioni i tempi sono lunghi, e gli errori commessi molto maggiori di quelli di SDA.

Per quanto riguarda le matrici casuali, si riporta in figura 9 il grafico con i residui per i due metodi con matrici di piccole dimensioni, cioè $k = 30$ e $m = 10$. Si evince che in questo caso gli errori compiuti sono paragonabili, con SDA più preciso all'interno del cerchio unitario. Anche qui SDA rispetta la simmetria del problema, al contrario di `polyeig`. Si noti che SDA suppone già che si conoscano $2(n-k)$ autovalori e autovettori, cioè quelli a 0 e infinito (e in tutte queste figure non sono riportati, né sarebbero visibili data la scala logaritmica), mentre `polyeig` li deve calcolare.

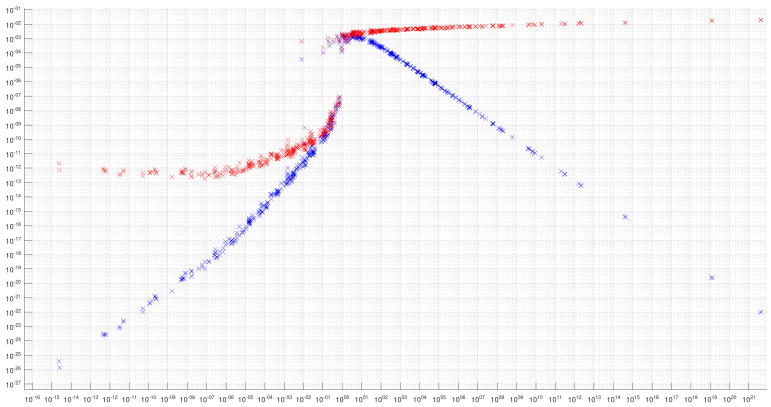

Figura 7: $RRes$ (in blu) e $RRes_{new}$ (in rosso) per matrici casuali.

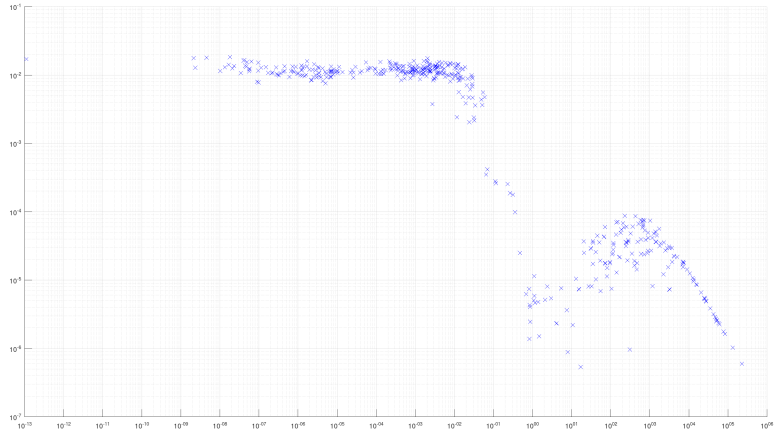

Figura 8: $RRes$ calcolato per `polyeig`

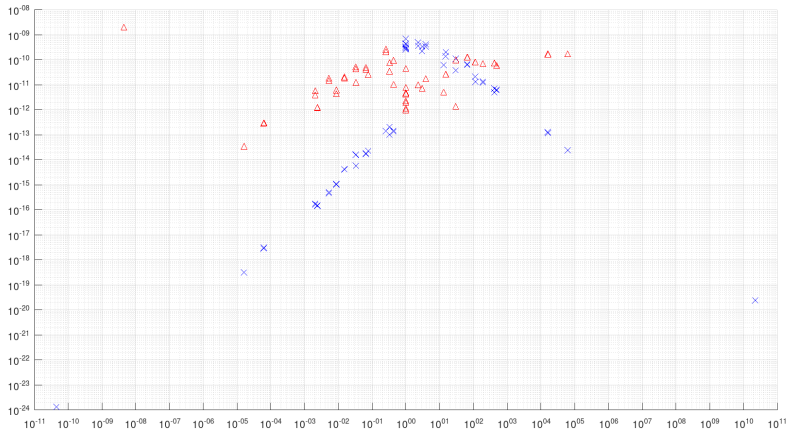

Figura 9: *RRes* calcolato per `polyeig` (in rosso) e con SDA (in blu).

6 Conclusioni

L'algoritmo SDA si comporta meglio di `polyeig` sia in termini di tempo che in termini di errore. Si deve però anche sottolineare che SDA risolve un insieme molto più specifico di problemi rispetto all'altro algoritmo.

Si è inoltre verificato che i risultati ottenuti partendo dalle matrici dei treni sono decisamente migliori rispetto a quelli ottenuti da matrici generiche.

Tutti i risultati ottenuti sono in accordo con quelli presentati in [1].

Riferimenti bibliografici

[1] Linzhang Lu, Teng Wang, Yueh-Cheng Kuo, Ren-Cang Li, Wen-Wei Lin, *A Fast Algorithm for Fast Train Palindromic Quadratic Eigenvalue Problems*. 2016.

[2] Linzhang Lu, Fei Yuan, Ren-Cang Li, *A New Look at the Doubling Algorithm for a Structured Palindromic Quadratic Eigenvalue Problem*. 2015.