國立臺灣師範大學理學院數學系

碩士論文

Department of Mathematics, College of Science

National Taiwan Normal University

Master's Thesis

# An Experimental Performance Study of Polynomial Preconditioner in PCG

張耀文

Yao-Wen Chang

指導教授：黃聰明 教授

Advisor: Professor Tsung-Ming Huang

中華民國 111 年 2 月

February 2022

**Abstract**

Solving laplacian system is common in the field of computer science nowadays. Preconditioner is an essential tool while solving linear system with indirect method. It may bring significant improvements to number of iterations, CPU time, and the errors. In this work we will start with graph laplacian, matrix splitting and approximation theories to get some polynomial preconditioners, and investigate the performance in the changes of different parameters in PCG(Preconditioned Conjugate Gradient) method mainly by experiments. We will show the experimental result as conclusion for the purpose of accelerating the iteration in future works.

**Keywords:** Graph Laplacian, PCG, Polynomial Preconditioner

# Contents

# 1 Introduction

Graph theory is becoming more and more important in modern science. Especially the fields including numerical computation. As the growth of your friend list in facebook, a network to your relationship getting more complicated. It is not suitable to analysis a large-sized graph by traditional methods.

First of all, we can describe a graph by graph laplacian or some equivalent decomposition [13]. Most of the graphs we were interested in may be sparse but large, therefore doing a Gaussian elimination is not a wise choice. In order to solving laplacian systems, mathematicians has developed various of methods. We will talk about this issue in the next section.

"Laplacian systems arise in many areas such as machine learning [16], computer vision, partial differential equations and interior point methods, and solvers are naturally needed." [15]

Solvers are naturally needed, so next, our goal changed into solving laplacian system $Lx = b$, which is a kind of linear system $Ax = b$ in general. To solve a linear equation

$$Ax = b \tag{1}$$

where $A$ is an $n \times n$ symmetric positive matrix. Notice that if $A$ is a graph laplacian, it must be symmetric and semi-positive definite. Thanks to its properties [14], we can easily giving some shift in original problem to satisfy the condition.

There were some classic splittings for iterative methods. Suppose $A = M - N$ is a splitting of $A$, then we have knew some standard methods.

- **Jacobi Method**

  Take $M = D$ and $N = L + U$.

- **Gauss-Seidel Iterative Method**

  Take $M = D - L$ and $N = U$.

- **Succesive-Overrelaxion Scheme**

  Take $M = \frac{1}{\alpha}(D - \alpha L)$ and $N = \frac{1}{\alpha}(\alpha U + (1 - \alpha)D)$

Just to name a few. Time complexity of direct methods has been verified completely [17]. Measuring pros and cons, indirect methods may sometimes been better. Hence we applied preconditioined conjugate gradient method (PCG) to solve laplacian systems mainly.

Take $x_0$ be an initial value which is approximate to $x$. Compute pcg algorithm until $x_{i+1}$ is close enough to $x_i$ (under the given tolerance or reach the limited maximum iteration numbers).

A suitable preconditioner is undoubtedly an effective tool to our iteration [1, 2, 8, 10, 18]. Differ from intuitive splitting above, a polynomial approximation may possibly works better. "Polynomials are widely used in computational models of scientific or engineering problems, because of their finite evaluation schemes" [19]. From the viewpoint of Johnson [1], there is another great choice by applying Neumann series, which can be regarded as the generalization of geometric series or power series. Then we solve (1) by doing

$$M^{-1}Ax = M^{-1}b, \tag{2}$$

where $M$ is decided by the series. We will discuss more properties about preconditioner in section 3.

Someone suggested parameterizing the series [2]. Excepts Neumann series(or Tayler series in general), Chebyshev polynomial is the most famous one in approximation theories. Besides, the most important approximation theorem, Weierstrass approximation theorem, has provided an awesome Bernstein basis polynomial. Thus we tried to examine about these different polynomials and meanwhile given some weights in order to improve the performance in section 4.

In section 5, we will show the result of experiments graphically. In analysis theory, those approximative polynomials works well. In practice, the shift added in linear system, the degree of approximate polynomial we have chosen, a weight setting in splitting, cause different influences to performance. Best of the best choice in theory is not necessary in computer. Thus we focus on iteration steps and cpu time to judge their outcomes. In the end, to make an inductive summary that how to improve the calculating in certain method by concluding our experiments.

# 2 Graph Laplacian

Graph laplacian is a useful tool and applied frequently in various fields. The relationships in facebook, twits on your page, ads popped out when searching something on google, are inseparable with graph laplacian. Graph theory applied some methods to analysis a graph, however, solving a problem about friends' network in facebook with huge size might be a matter of computer

science. Hence we need graph laplacian to simplify these tasks.

## 2.1 Graph Laplacian

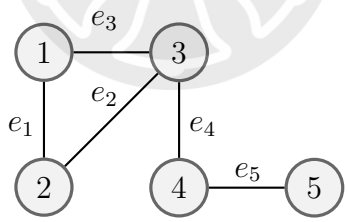
Consider a undirected and unweighted graph $G = (V, E)$, this representation is not useful to our computation. Two important matrices, adjacency matrix $A$ and degree matrix $D$, can be defined by vertices and edges as follow:

$$A(G)_{i,j} = \begin{cases} 1 & if \quad v_i v_j \in E, \\ 0 & otherwise. \end{cases}$$

In other words, we write 1 in $A_{i,j}$ if $v_i$ is adjacent to $v_j$. And

$$D(G)_{i,j} = \begin{cases} d_i & if \quad i = j, \\ 0 & if \quad i \neq j. \end{cases}$$

where $d_i$ is degree of $v_i$, the number of edges connected with vertex $v_i$. That is, $D$ is a diagonal matrix with diagonal elements are the degree of each vertices. Take a graph G as example:




For the first row of $A$, we can see vertex *1* is adjacent to vertices *2*, *3*. By definition, $A_{1,2} = 1$ and $A_{1,3} = 1$. Keep following the rule as above to finish $A$. Then

$$A(G) = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}.$$

Next, to see the adjacency matrix. For $D_{1,1}$, there are two vertices, *2* and *3*, adjacent to *1*. Hence $D_{1,1} = 2$

$$D(G) = \begin{bmatrix} 2 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

Next we define the graph laplacian of $G$ by

$$L(G) := D - A.$$

For last example,

$$L(G) = \begin{bmatrix} 2 & -1 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ -1 & -1 & 3 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 1 \end{bmatrix}.$$

From this definition, we can see the concept of matrix splitting. Why we named this as a laplacian? Recalling the traditional laplacian operator for an function $f$:

$$\Delta f = \nabla^2 f.$$

Similarly, we can define a $f : V \to \mathbb{R}$ act on all vertices of the graph. Then it's 'gradient' can be taken as

$$g(e_k) := f(v_i) - f(v_j), \tag{3}$$

$e_k$ is the edge connecting vertices $v_i$ and $v_j$ and set $i < j$. Therefore, $L$ can be regarded as a laplacian operator. Refer to Bernstein's notations [13], we

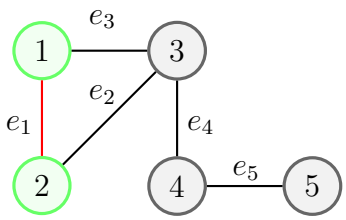
represent function $f$ and $g$ as vectors.

$$f := \begin{bmatrix} f(v_1) & f(v_2) & \cdots & f(v_n) \end{bmatrix}^T$$

and

$$g := \begin{bmatrix} g(e_1) & g(e_2) & \cdots & g(e_m) \end{bmatrix}^T.$$

For the same graph as example.




Then the first element $g(e_1) = f(1) - f(2)$.

Keep doing so, we can easily find a $K_{m\times n}$ satisfying

$$Kf = g. \tag{4}$$

Then $K$ can be solved by

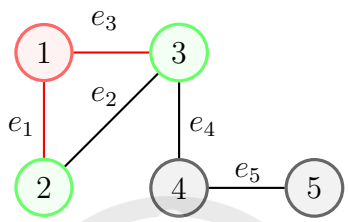
$$K \begin{bmatrix} f(1) \\ f(2) \\ f(3) \\ f(4) \\ f(5) \end{bmatrix} = \begin{bmatrix} f(1) - f(2) \\ f(2) - f(3) \\ f(1) - f(3) \\ f(3) - f(4) \\ f(4) - f(5) \end{bmatrix}.$$

In this case,

$$K = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 1 & -1 \end{bmatrix}.$$

This $K$ is the incidence matrix which describe the edges of a graph in another perspective from adjacency matrix.

Finally we can think about the divergence of a graph. Divergence is an operator measuring the amount of flow in or out a point. So we consider a $d_{n\times 1}$ to show the divergence of each vertices in $G$ by counting edges connected with them. For instance,




$e_1$ and $e_3$ connected with *1*, then $d_{1,1} = g(e_1) + g(e_3)$. (Notice that symbols should follow the rule $i < j$.) In the end, we can figure out that

$$K^T g = d. \tag{5}$$

In our example,

$$\begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 \\ 0 & -1 & -1 & 1 & 0 \\ 0 & 0 & 0 & -1 & 1 \\ 0 & 0 & 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} g(e_1) \\ g(e_2) \\ g(e_3) \\ g(e_4) \\ g(e_5) \end{bmatrix} = \begin{bmatrix} g(e_1) + g(e_3) \\ -g(e_1) + g(e_2) \\ -g(e_2) - g(e_3) + g(e_4) \\ -g(e_4) + g(e_5) \\ -g(e_5) \end{bmatrix}.$$

In conclusion, equation (4) (5) imply that

$$K^T K f = d \tag{6}$$

Furthermore, Bernstein and Das [13, 14] gave us opinion that

$$L = D - A := K^T K. \tag{7}$$

This matrix $K$ here is said to be the incidence matrix of graph $G$. That's the reason why we defined the $L$ as laplacian of a graph. Now check this easily by focusing on diagonal elements and the rest respectively.

$$\begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 \\ 0 & -1 & -1 & 1 & 0 \\ 0 & 0 & 0 & -1 & 1 \\ 0 & 0 & 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} 1 & -1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 1 & -1 \end{bmatrix} = \begin{bmatrix} 2 & -1 & \cdots & 0 \\ -1 & \ddots & & \vdots \\ \vdots & & & \\ 0 & \cdots & & 1 \end{bmatrix}$$

A column in $K$ corresponds to a vertex in $G$, then number of nonzero element on $i-$th column represents the edges connected to $v_i$. Hence the inner product by $i-$th column to itself is the degree of $v_i$.

$$\begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 \\ 0 & -1 & -1 & 1 & 0 \\ 0 & 0 & 0 & -1 & 1 \\ 0 & 0 & 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} 1 & -1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 1 & -1 \end{bmatrix} = \begin{bmatrix} 2 & -1 & \cdots & 0 \\ -1 & \ddots & & \vdots \\ \vdots & & & \\ 0 & \cdots & & 1 \end{bmatrix}$$

See the off-diagonal entries next. The first element in column 2 and column 1 are nonzero, that is, vertex *1* and *2* both connected by edge $e_1$. Therefore we record $A_{2,1} = 1$ implies that $L_{2,1} = -1$.

## 2.2 Laplacian System

In equation (7), graph laplacian $L$ can be written as $K^T K$, which is the form of Cholesky decomposition. When solving a linear system with a case

$A = G$, applying this factorization could been a good choice. However, direct methods becoming difficult with the degree, or the size of graph getting larger.

Fortunately, assays have gave us some properties of graph laplacian [13, 14, 15]:

- Graph laplacian $L$ is symmetric.
- Graph laplacian $L$ is positive semidefinite.
- All eigenvalues of $L$ are nonnegative.
- The smallest eigenvalue is zero.

These properties lead us appling approximate iteration solver when solving laplacian system easily. Conjugate gradient method with preconditioner may be a suitable choice.

# 3 Preconditioners

For symmetric positive problem in (1), we choose preconditioined conjugate gradient(PCG) method to solve it mainly. Never forgot that a graph laplacian matrix $L$ may be only semi-positive definite. A small shift can change our matrix positively. Let $M$ be a chosen $n \times n$ positive definite matrix. Refer to Johnson's algorithm of PCG iteration [1], we solve $d_i$ by inner iteration

$$Md_i = r_i, \tag{8}$$

where $r_i$ is the residual in the $i^{th}$ step of iteration, for $i = 0, 1, 2, \ldots$. Since

$$d_i = M^{-1}r_i = M^{-1}(b - Ax_i),$$

we actually need $M^{-1}$ while calculating. This can be fulfilled by using function handles in matlab.

From another perspective to realize this problem easier, we need $A^{-1}$ to solve (1). However the inverse matrix is so difficult to find that people don't use direct methods in big size coefficient matrix. To compromise these difficulties, pseudo inverse or Drain inverse can be a way [12]. But that is not our point. Now the goal changes into looking for a matrix $M$ such that $M^{-1} \approx A^{-1}$. Then

$$M^{-1}Ax = M^{-1}b \tag{9}$$

implies

$$x \approx M^{-1}b.$$

For any given symmetric and positive $M$, there exists a matrix $E$ such that $M = EE^T$. We have seen this in (7), graph laplacian can be Cholesky factorized naturally. If the spectrum of $M^{-1}A$ has better clustered than $A$'s, we may iterative (9) faster than the origin problem [7] . Since $E^{-1}AE^{-T}$ has the same eigenvalues to $M^{-1}A$, the system can be transformed into

$$(E^{-1}AE^{-T})(E^Tx) = E^{-1}b. \tag{10}$$

To construct a suitable $M$, we have to do some scaling and splitting first, then apply theories of polynomial approximation [1, 2]. Weierstrass theorem has ensured that a continuous function can be uniformly approximated as closely as we want by some polynomial. If ones can split $A$ to the form $I-G$, then

$$A^{-1} = (I-G)^{-1} = I + G + G^2 + \cdots \tag{11}$$

(11) is usually said to be a Neumann series, which is an infinitely series of the form

$$\sum_{i=0}^{\infty} G^i. \tag{12}$$

But here comes a problem: Knowing that the power series, for example, $(1-x)^{-1}$ converges to $1+x+x^2+...$ over the certain radius of convergence. What if the matrix polynomial? Climent indicated that (12) converges if and only if the sequence $\{G^k\}_k$ converges to null matrix, i.e., if and only if the spectral radius, that is the maximum of its eigenvalues, is smaller than 1 [3, 4].

Without loss of generality that $A$ can be scaled to have unit diagonal and denoted as $\tilde{A}$ by

$$\tilde{A} = \sqrt{D}A\sqrt{D}, \tag{13}$$

where $D$ is the diagonal matrix of $A$. Hence $\tilde{A}$ is of the form $I-G$, $G$ is symmetric with zero entries on its diagonal. We have to notice that the right hand side and the solution in (1) should change along with the scaling. $\tilde{x} = \sqrt{D}^{-1}x$, and $\tilde{b} = \sqrt{D}\cdot b$, then $\tilde{A}\tilde{x} = \tilde{b}$. Next, split the matrix by taking $G = I - \tilde{A}$.

$$\tilde{A} = I - (I - \tilde{A}) := I - G. \tag{14}$$

Then

$$\tilde{A}^{-1} = (I-G)^{-1} = I + G + G^2 + \cdots.$$

We choose

$$M_k^{-1} = \sum_{i=0}^{k} G^i. \tag{15}$$

be a Neumann preconditioner of order $k$. Theoretically, as the order $k$ increasing(large enough), the performance of convergence shall be better. Experimentally, it is not necessary that the order took higher, the iteration is better, especially the cpu time in big matrices. Saad has gave us some graphically outcomes between iterative steps and the total times of matrices multiplication [18]. Since the inner iteration kept doing the product between matrices and vectors , which may increase the total CPU time mainly.

# 4 Other Choices

Excepts the splitting and Neumann series we have mentioned before, there are some other popular choices.

## 4.1 Weighted Preconditioners

Recalling the matrix splitting, now we let

$$N = N(\omega) = I - \omega A \tag{16}$$

and $\omega$ is a scaling parameter. Then we can change (14) into

$$\hat{A} := \omega A = I - (I - \omega A) = I - N.$$

Equation (14) is the case when $\omega = 1$. We are wondering that whether the performance changes along with $\omega$ by taking

$$M_k = \sum_{i=0}^{k} N(\omega)^i.$$

We changed $\omega$ to compare their performance, see figures in subsection 5.2.

## 4.2 Splittings

The splitting in (14) can be regarded as a special case of

$$\hat{A} = \omega A = D - (D - \omega A) \tag{17}$$

where $D$ is the diagonal of $A$. After doing the scale in (13), it became to have unit diagonal, thus $D = I$. Then

$$\begin{aligned}(\hat{A})^{-1} &= [D - (D - \hat{A})]^{-1} \\ &= [D(I - (I - D^{-1}\hat{A}))]^{-1} \\ &= [I - (I - D^{-1}\hat{A})]^{-1}D^{-1} \\ &= (I - N)^{-1}D^{-1}.\end{aligned}$$

Result in the approximation

$$M_k^{-1} = (I + N + N^2 + \cdots + N^k)D^{-1} = (\sum_{i=0}^{k} N^i)D^{-1},$$

where

$$N = I - D^{-1}\hat{A}.$$

In general, the diagonal $D$ can be changed into any other matrix. A small experiment showed that iterate with (14) has better performance.

## 4.3 Different Polynomials

Besides using Neumann series, there are some other famous choices of polynomial in, such as Legendre polynomial, Chebyshev polynomial. Coefficients may sometimes being so complicated, an easier structure, for example, the Bernstein polynomial is also a great decision. To see definitions and first few of them in appendix. Experimental result will be showed in

the next section.

Mudde gave us a comparison: Legendre polynomial is the best in $L_2$ norm, and Chebyshev is the best in uniform norm [5]. There is a little conclusion that Chebyshev polynomial might be a better choice in general. From the perspective of mini-max iteration, we can scale Chebyshev polynomials by using the spectrum of coefficient matrix [1, 8].

Then setting some coefficients $c$ in $M_k$ (15), and since $M_k$ is depending on $A$, let

$$\dot{M_k}^{-1} = \sum_{i=0}^{k} c_{i,k} A^i := p_k(A). \tag{18}$$

Now we hope a polynomial $q$

$$\dot{M_k}^{-1} A = p_k(A)A := q_{k+1}(A) \tag{19}$$

to be as close to 1 as possible.

Let $\lambda_1, \lambda_n$ be the smallest and the largest eigenvalues of the coefficient matrix. Considering the domain of Chebyshev polynomial, for any eigenvalues $\lambda \in [\lambda_1, \lambda_n]$, taking a map $\mu$ from $[\lambda_1, \lambda_n]$ into $[-1, +1]$ where

$$\mu(\lambda) = -1 + 2\frac{\lambda - \lambda_1}{\lambda_n - \lambda_1}. \tag{20}$$

Define

$$\hat{T}_{k+1}(\lambda) = 1 - \frac{T_{k+1}\left(\mu(\lambda)\right)}{T_{k+1}\left(\mu(0)\right)}. \tag{21}$$

By Saad's theorem 3 in [8], the above $\hat{T}_{k+1}$ may be a choice for $q_{k+1}$ which has a smallest condition number over all polynomials of degree $\leq k+1$. In

general, (21) can set with a scalar $\gamma$

$$\tilde{T}_{k+1}(\lambda) = 1 - \frac{T_{k+1}\left(\mu(\lambda)\right)}{T_{k+1}\left(\mu(\gamma)\right)}. \tag{22}$$

Moreover, the $\mu$ in (20) can be defined by different intervals

$$\tilde{\mu}(\lambda) = -1 + 2\frac{\lambda - a}{b - a}. \tag{23}$$

Chebyshev polynomials has the smallest condition number. Pyzara and Bylina's results [9] has shown that: "the condition number provides an approximate upper bound on the error in a computed solution". However, it is not necessary the iteration performs better or faster. Youcef Saad gave us an important information: "The usual optimal parameters $a = \lambda_1$ and $b = \lambda_2$ used in Chebyshev iteration are no longer optimal in Chebyshev polynomial preconditioned conjugate gradient method." [18]. Using the largest and the smallest eigenvalue as parameters may sometimes yield poor convergence. Yet we still try this in comparison with other preconditioners.

# 5 Experiments

In this section, we applied the thinkings in previous sections. Showing some results to induct a common trend. (All these computations were calculated and plotted by MATLAB R2019b in MacBook Pro 2015.)

The following table shows the attributes to our benchmarks. 'nnz' is the number of nonzero matrix elements. 'spy' is the sparsity of laplacian matrix. '$d_{max}$' and '$d_{min}$' are the largest and the smallest value on diagonal ,repectively.

| | benchmark | size | nnz | spy | $d_{min}$ | $d_{max}$ |
|---|---|---|---|---|---|---|
| 1 | hamster-household | 1576 | 4032 | .16% | 0 | 147 |
| 2 | friendships-hamster | 1858 | 12534 | .36% | 0 | 272 |
| 3 | hamster | 2426 | 16631 | .28% | 0 | 273 |
| 4 | hamster-friend | 2952 | 12534 | .15% | 0 | 272 |
| 5 | wikipedia-link-co | 8252 | 177420 | .26% | 0 | 1326 |
| 6 | ego-twitter | 23370 | 33101 | .006% | 0 | 239 |
| 7 | ego-gplus | 23628 | 39242 | .007% | 0 | 2771 |
| 8 | brightkite-edges | 58228 | 214078 | .006% | 0 | 1134 |
| 9 | flickrEdges | 105938 | 2316948 | .02% | 0 | 5425 |
| 10 | wordnet-words | 146005 | 656999 | .003% | 0 | 1008 |
| 11 | gowalla-edges | 196591 | 950327 | .002% | 0 | 14730 |
| 12 | com-amazon | 334863 | 925872 | .0008% | 0 | 549 |

Table 1: Attributes of benchmarks

Then we take benchmark 8 as a representative. Being a control group, plotted for the purpose of comparing with sub-experiments in this section. It was iterated by pcg without preconditioner in different tolerance.

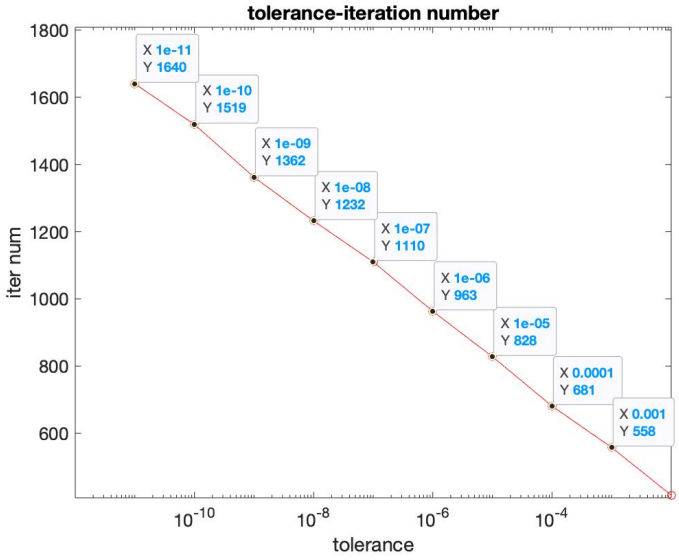



Figure 1: PCG, benchmark 8, without preconditioner, run tolerance

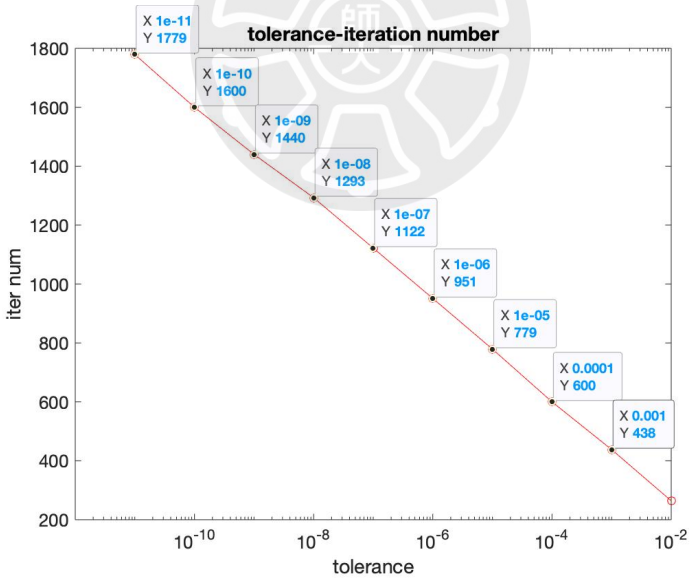
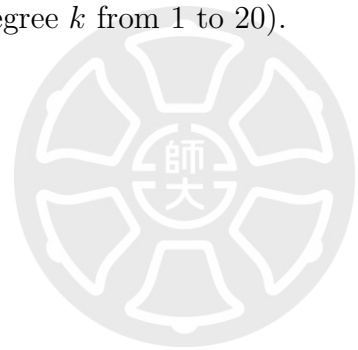


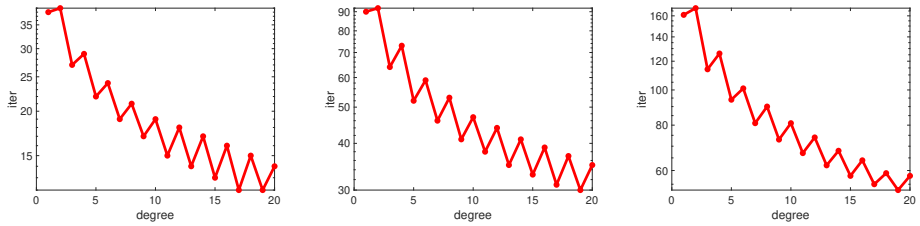

Figure 2: PCG, benchmark 12, without preconditioner, run tolerance

## 5.1 Neumann Series

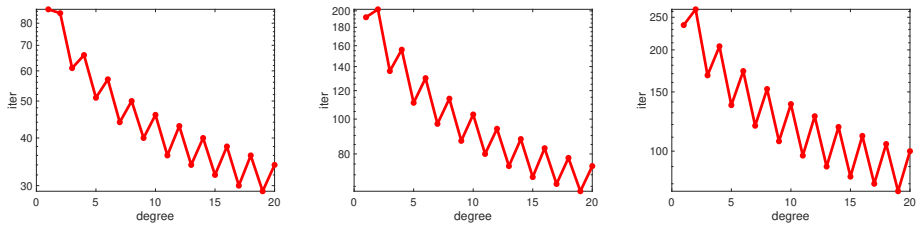
From the perspective of convergence analysis, taking a higher degree polynomial approximation, the error might be smaller. It can be realized from error analysis to Taylor series. But the iteration is not necessary faster. See this result from graphs of cpu time in the following subsections.

We can define a preconditioner easily in matlab by setting the function handle. Take (15) as the above preconditioner $M$. In the following experiments we fixed a maximum iteration number and tolerance ($10^{-5}$). To see the changes of iteration numbers when the degree of Neumann series increase (examine the degree $k$ from 1 to 20).

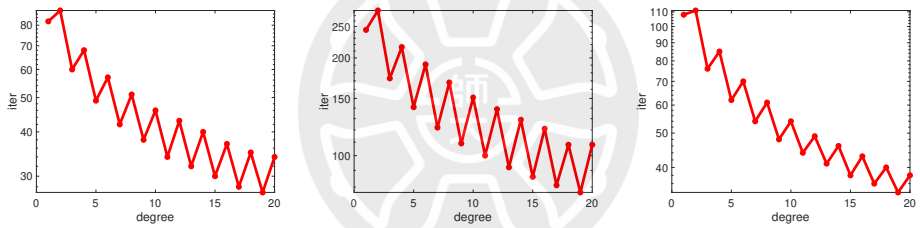




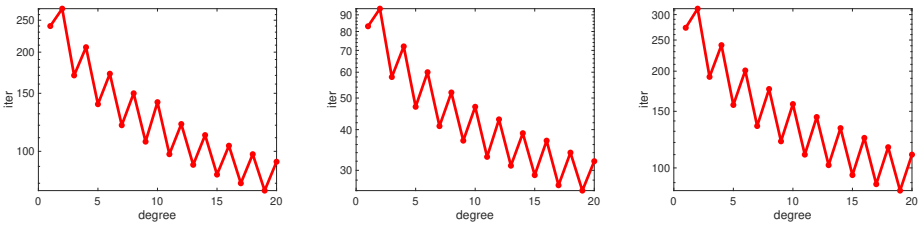

(a) benchmark 1 (b) benchmark 2 (c) benchmark 3




(d) benchmark 4 (e) benchmark 5 (f) benchmark 6




(g) benchmark 7 (h) benchmark 8 (i) benchmark 9




(j) benchmark 10 (k) benchmark 11 (l) benchmark 12

Figure 3: PCG, preconditioner = Neumann series

It is clearly that they have similar trend as the change of degree $k$ in $x-$axis. Let us take benchmark 8 as an example, discuss some more details.

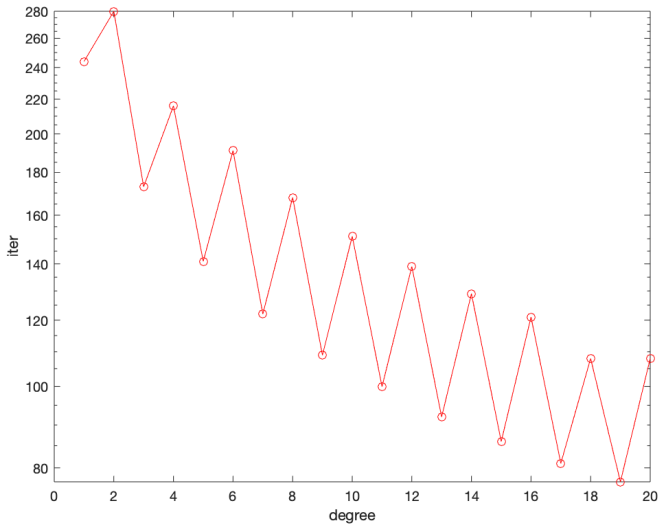



Figure 4: benchmark 8, Neumann, iteration times

It shows that the iteration numbers decreased 'alternatively'. And how about its cpu time?

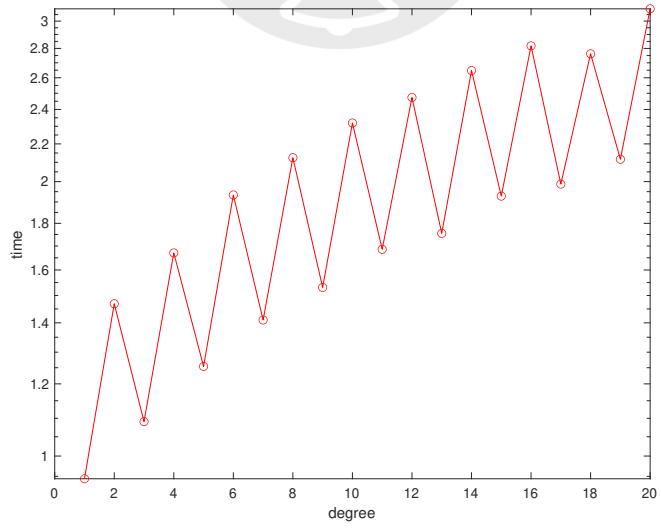
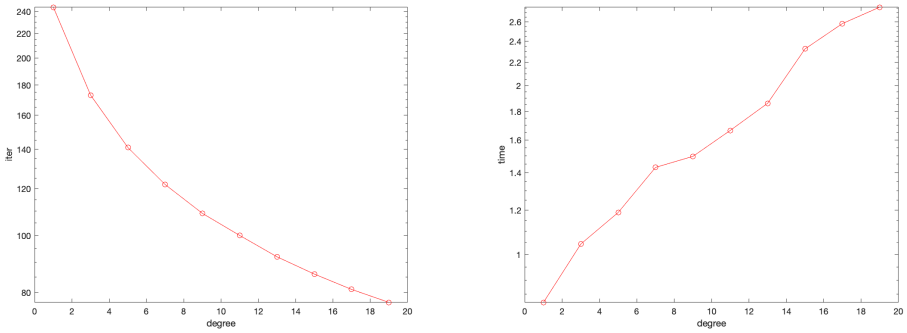


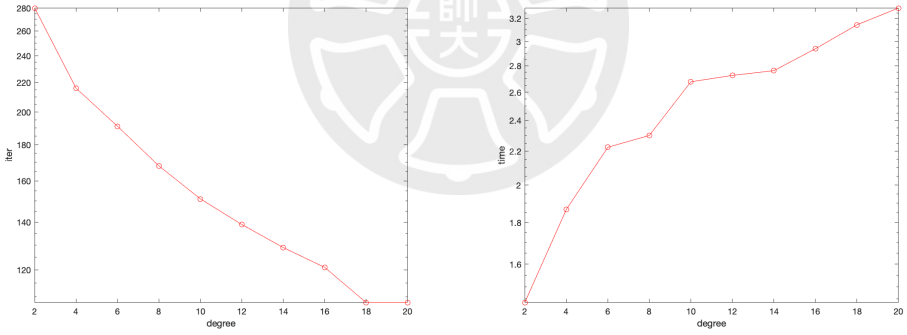

Figure 5: benchmark 8, Neumann, cpu time

Globally, as the degree grew higher, it cost more time during inner iterate of pcg. Next we separate odd and even degree.




(a) odd degree, iteration times (b) odd degree, cpu time

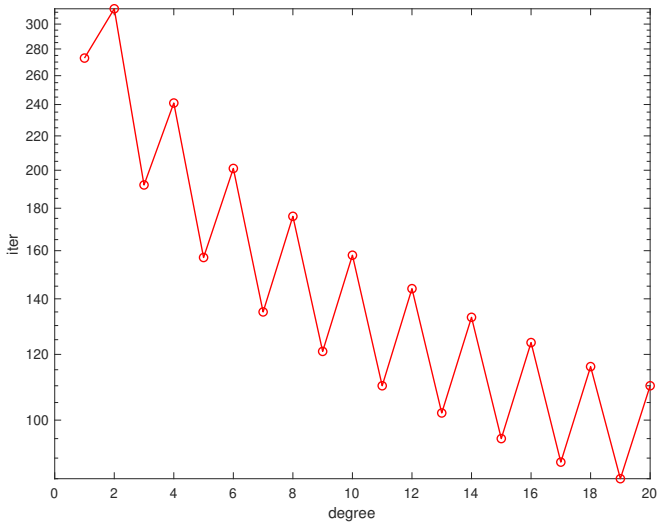
Figure 6: benchmark 8, odd degree




(a) even degree, iteration times (b) even degree, cpu time

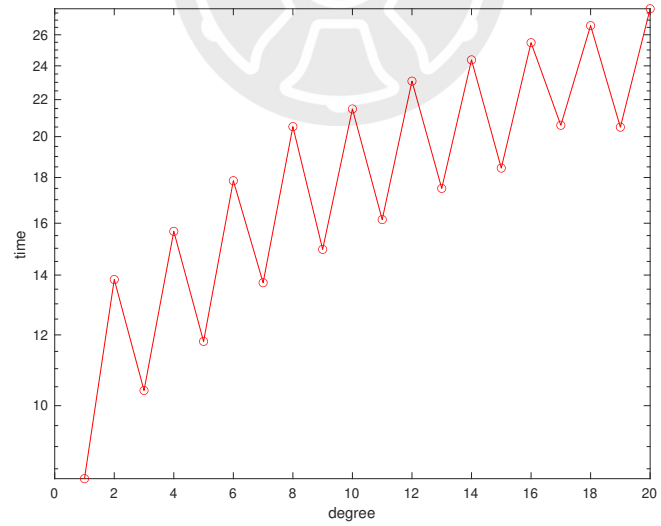
Figure 7: benchmark 8, even degree

Here we found that iteration numbers is decreasing. A bad news is shown at the right of Figure 6, 7. Although the iteration times decreased with the degree grown, cpu time increased. (Notice that these graphs are plotted by 'semilogy'). We can examine another benchmark to see this trend since the cpu time may sometime affected by complicated factors in computer.



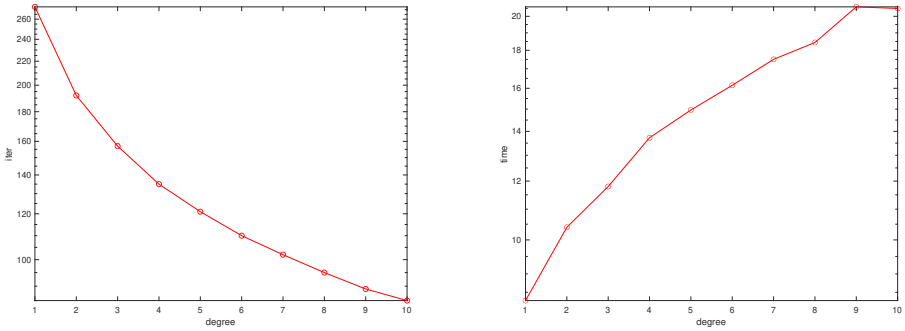

(a) iteration times

Figure 8: benchmark 12, Neumann, iteration times



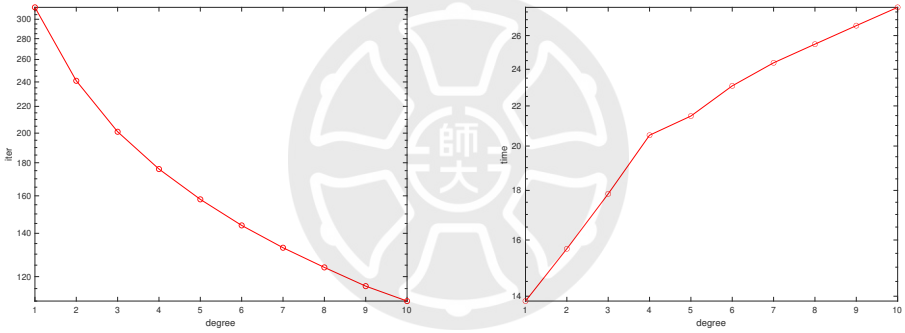

(a) iteration times

Figure 9: benchmark 12, Neumann, cpu time




(a) odd degree, iteration times (b) odd degree, cpu time

Figure 10: PCG, benchmark 12, odd degree




(a) even degree, iteration times (b) even degree, cpu time

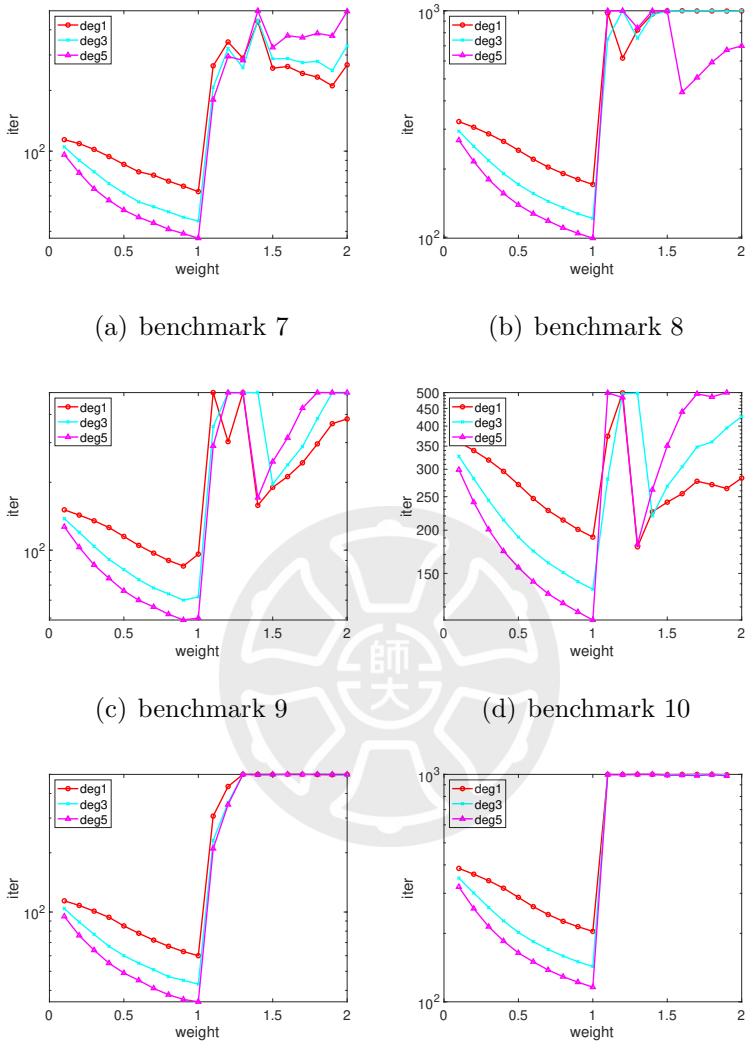
Figure 11: PCG, benchmark 12, even degree

Interestingly, iteration times decreased globally, but faster in odd degree. An experimental conclusion, there is no doubt that the degree is higher, the iteration number is lower. Yet we should consider the growth of cpu time meanwhile. Since the computer may do more matrix-vector multiplication in each steps of iteration if we have chosen a higher degree approximation [18].

## 5.2 Weighted

The following figures show how the performance is with change of weights, the $\omega$, in different degree. Having the experiences from above subsection, we known that the performance might be different between odd and even degree polynomial approximation. Thus we focus on odd and even degree respectively. Here are the higher-degree exapmles, the trend is more obviously in these graphs.

Odd degree:




(a) benchmark 7 (b) benchmark 8

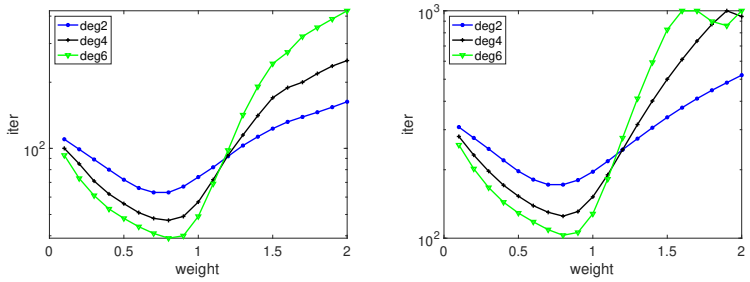
(c) benchmark 9 (d) benchmark 10

(e) benchmark 11 (f) benchmark 12

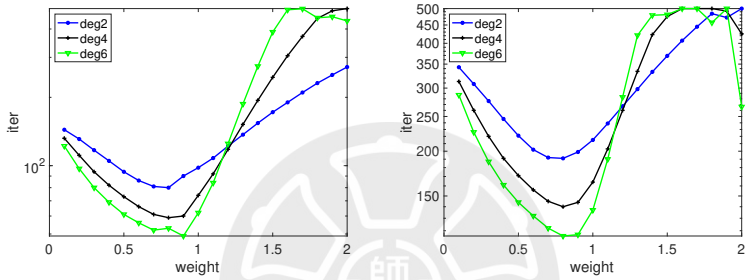
Figure 12: PCG, odd degree, Neumann series, change of $\omega$

In figure 12, we can see that odd degree Neumann series reach the lowest iteration times when setting $\omega = 1$. In other words, it does not need weight in these cases.

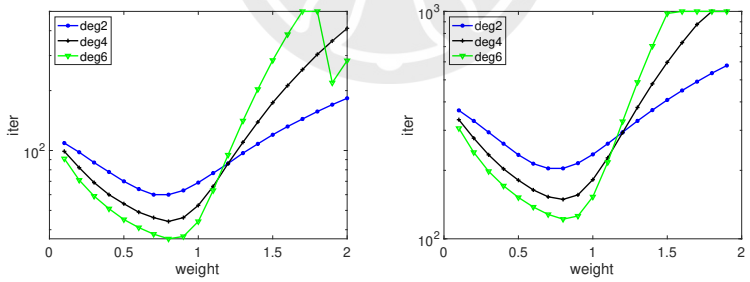
Even degree:




(a) benchmark 7 (b) benchmark 8



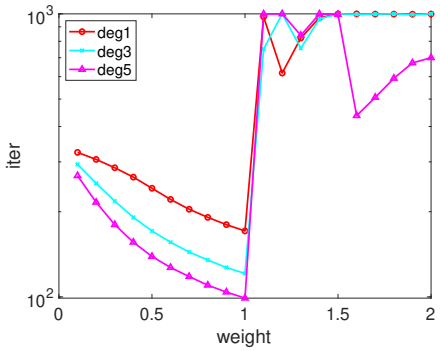

(c) benchmark 9 (d) benchmark 10



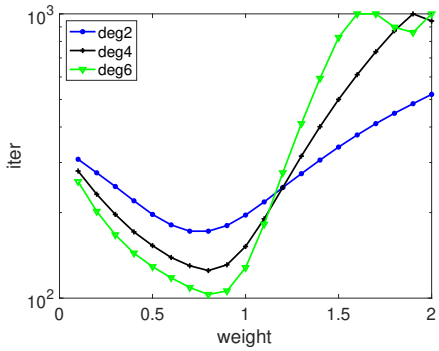

(e) benchmark 11 (f) benchmark 12

Figure 13: PCG, even degree, Neumann series, change of $\omega$

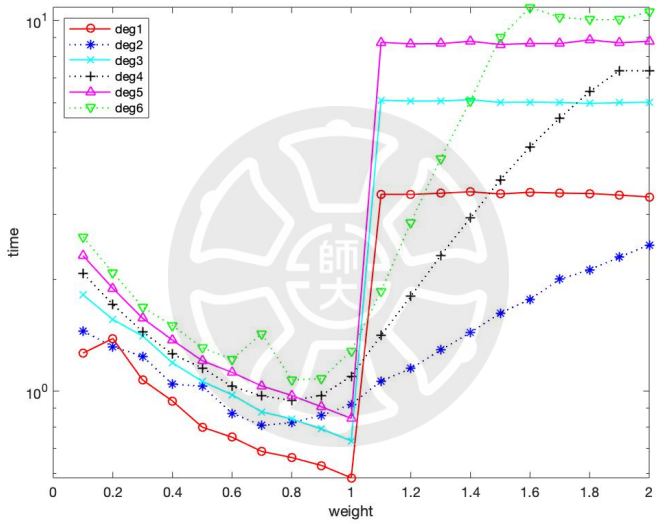
Differ from odd curves in Figure 12, Figure 13 shows that even curves reached the lowest when $\omega < 1$.




(a) odd degree, iteration times



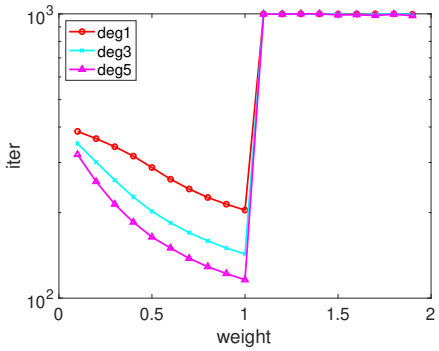

(b) even degree, iteration times



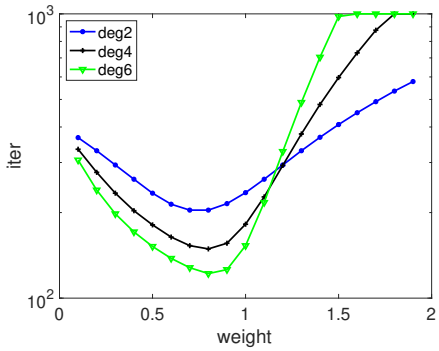

(c) cpu time

Figure 14: PCG, benchmark 8

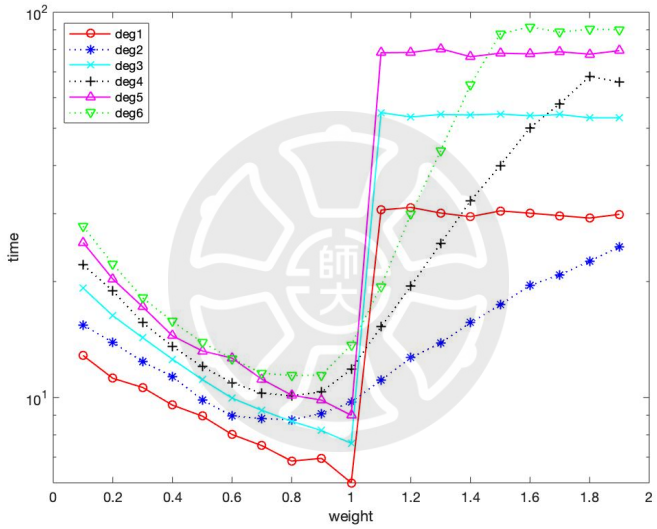
From (a) and (b) in benchmark 8, we can see the difference between odd and even degree cases.




(a) odd degree, iteration times




(b) even degree, iteration times



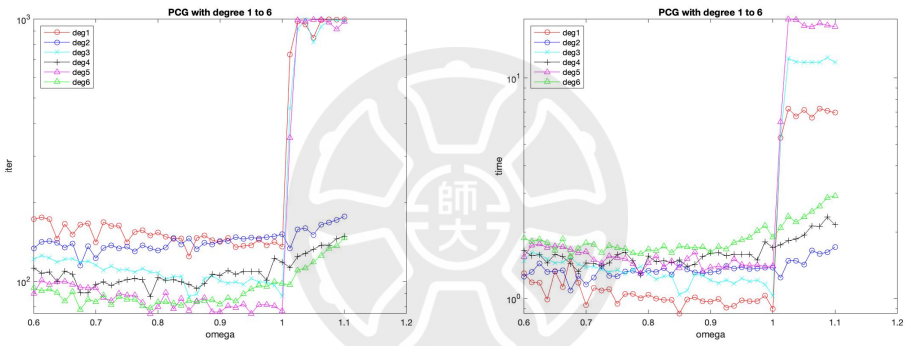

(c) cpu time

Figure 15: PCG, benchmark 12

Take two outcomes as an example. First, we observe that the number of iterations and CPU time decreased when $\omega$ is small. Reached the lowest while $\omega$ near 1 then increased quickly after 1. Notice that the iteration does not converge in big $\omega$, or, when the curves became flat in the right side of these graphs.

It matches the experimental result in subsection 5.1 that the iteration number and cpu time change along with the degree of Neumann approximation. The iteration number is lower when the degree is higher. Since the matrix-vector multiplication waste a lot of time, the performance in CPU time is not necessary better.

Then divide the difference smaller to check the critical point when $\omega$ near 1 and the trend greater than 1. Zoom in to the interval around 1 to see the trend locally.

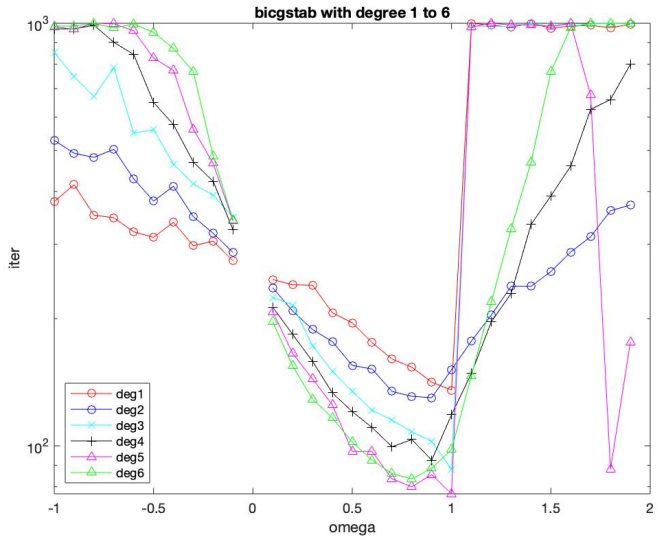



(a) iteration times (b) cpu time.

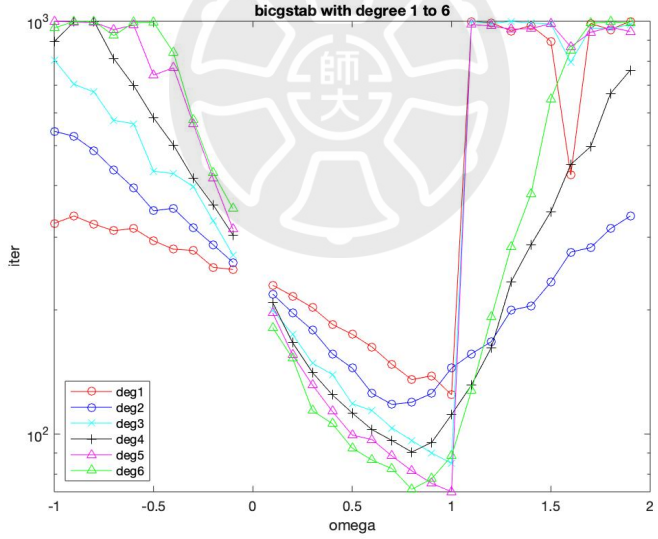
Figure 16: PCG with $\omega \in [0.6,1.1]$

In Figure 7, 8, 9, we can see **even** curves reach the lowest at about $0.7 \sim 0.8$, **odd** curves reach the lowest at 1. When $\omega > 1$, even curves increase smoothly, but odd curves diverge rapidly.

Next, we were interested in negative $\omega$. Considering the limit of PCG, we applied biconjugate gradient stabilized method (bicgstab) to investigate the performance when $\omega < 0$.



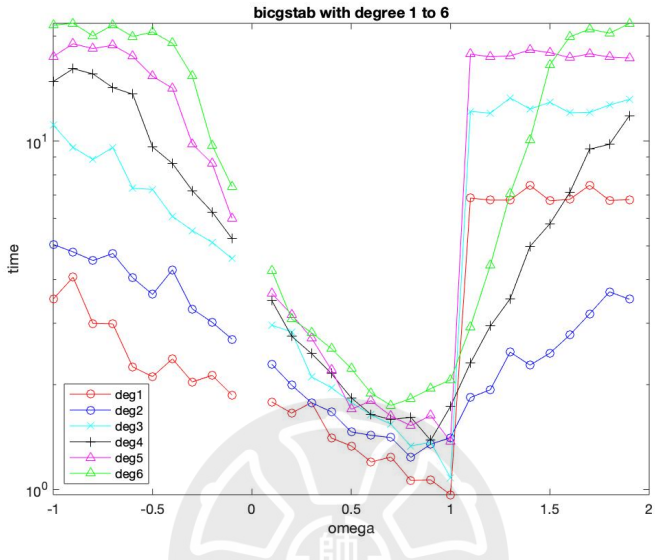

(a) iteration times, benchmark 8



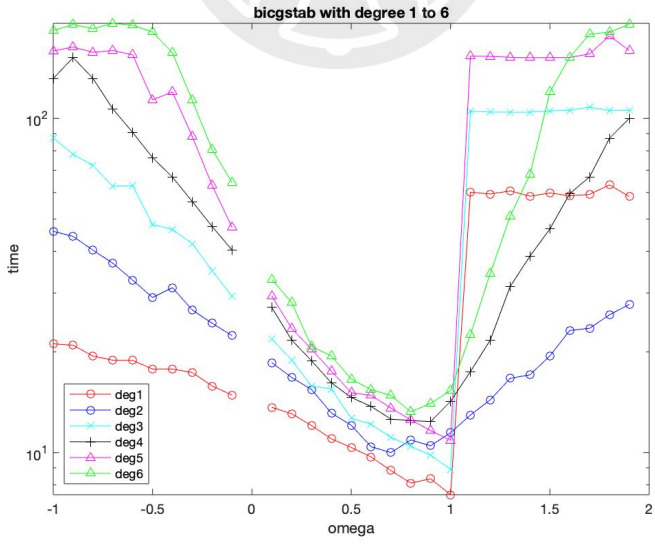

(b) iteration times, benchmark 12

Figure 17: bicgstab, iteration times

See Figure 17, the performance is similar to previous one in positive $\omega$. In negative $\omega$, it is opposite to positive cases. However the iteration times is much more higher. Next, let's check the CPU time of these bicgstab.




(a) cpu time of benchmark 8



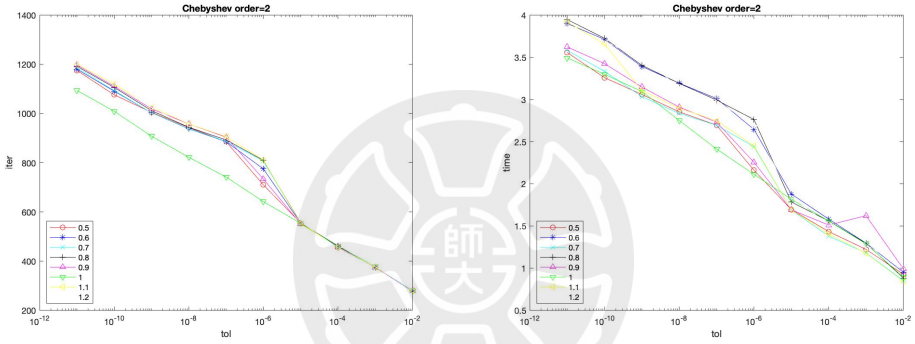

(b) cpu time of benchmark 12

Figure 18: bicgstab, CPU time

In the two benchmarks of Figure 18, it showed that low degree iterated faster in negative $\omega$, but still worse than positive cases.

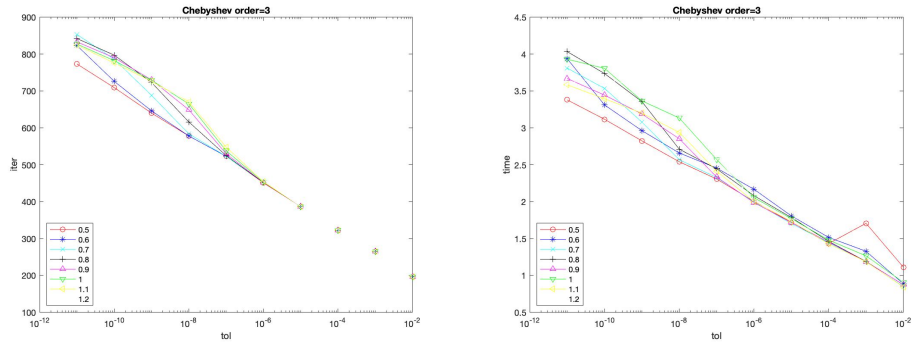
## 5.3 Chebyshev Polynomial

In previous subsection, we found that iterate with different $\omega$ may lead to better results. Evaluating $\hat{T}_{k+1}(y) = p_k(y) \cdot y$ in (21). And also shift the coefficient matrix with different $\omega$. See the outcomes of degree 2 and 3 as representative in following figures.




(a) iteration times (b) cpu time

Figure 19: Chebyshev, benchmark 8, degree=2



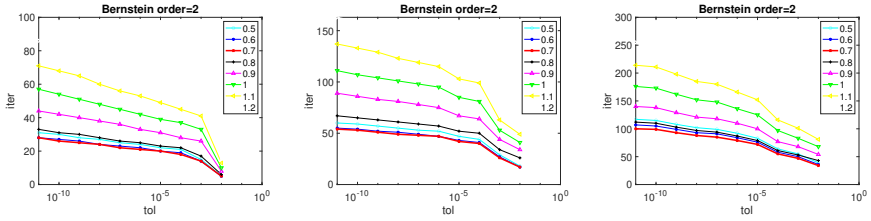

(a) iteration times (b) cpu time

Figure 20: Chebyshev, benchmark 8, degree=3

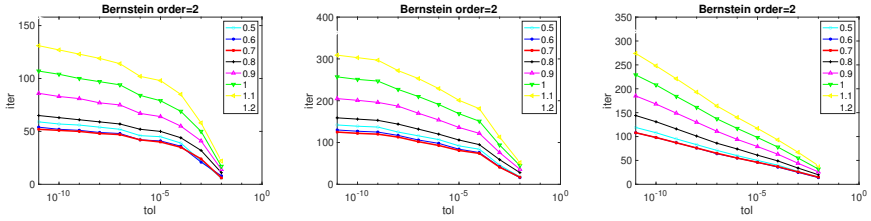
In Figure 19 and 20, Chebyshev method has different outcomes when given $\omega$ in even and odd degree. Although they showed a little differences, iteration times and cpu time still getting worse than Neumann series. Even if we added some weights, it still correspond with Saad's outcome [18], taking $\lambda_1$ and $\lambda_n$ as parameters may yield poor convergence.

## 5.4 Bernstein Polynomial

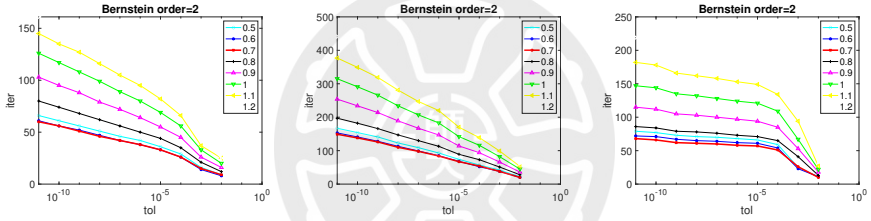
In approximation theories, Bernstein basis polynomial is another useful tool. To see the definition and first few examples in **Appendix C**. Coefficients in Bernstein basis can be easily decided by combination numbers, especially in low degree cases. In this experiment, I ran the tolerance and $\omega$ to see the performance.



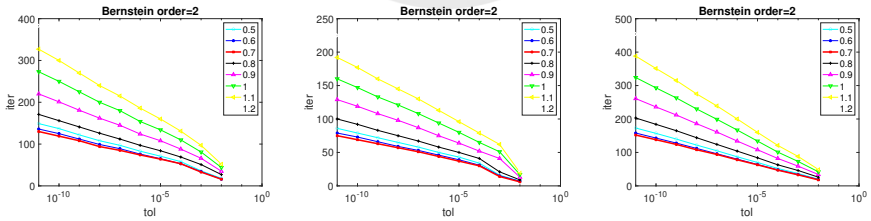

(a) benchmark 1 (b) benchmark 2 (c) benchmark 3




(d) benchmark 4 (e) benchmark 5 (f) benchmark 6




(g) benchmark 7 (h) benchmark 8 (i) benchmark 9




(j) benchmark 10 (k) benchmark 11 (l) benchmark 12

Figure 21: PCG, preconditioner = Bernstein polynomial

Experiments show that different benchmarks have the same trend. Taking one of them to see more detail.

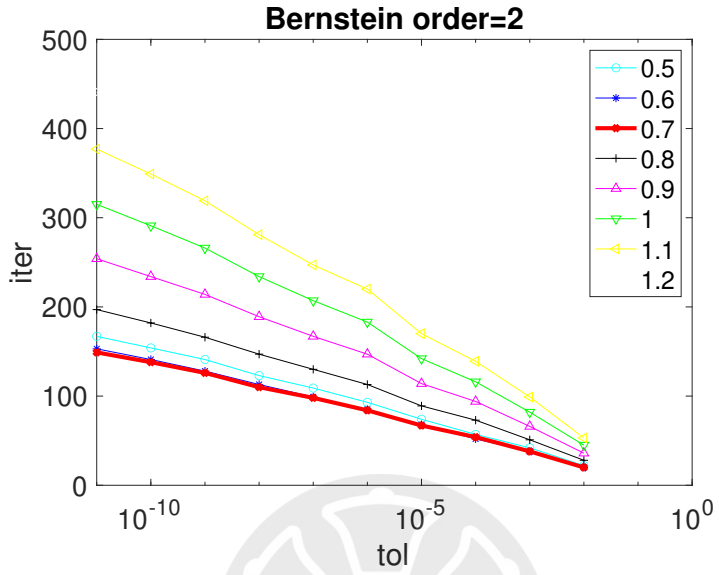



Figure 22: Bernstein, benchmark 8, iteration times

Iteration number increased as the tolerance became smaller. And we can see curves approximate to a straight line(plotted by 'semilogx'). Good news is even if the tolerance is so small, iteration number is still better than most of the cases in previous subsections.

See curves in Figure 22. Taking a weight at about $\omega = 0.7$, it has the lowest iteration number. Experiences were concluded that iteration steps is lower, the total cpu time is not necessary faster. Next graph will tell us the fact.

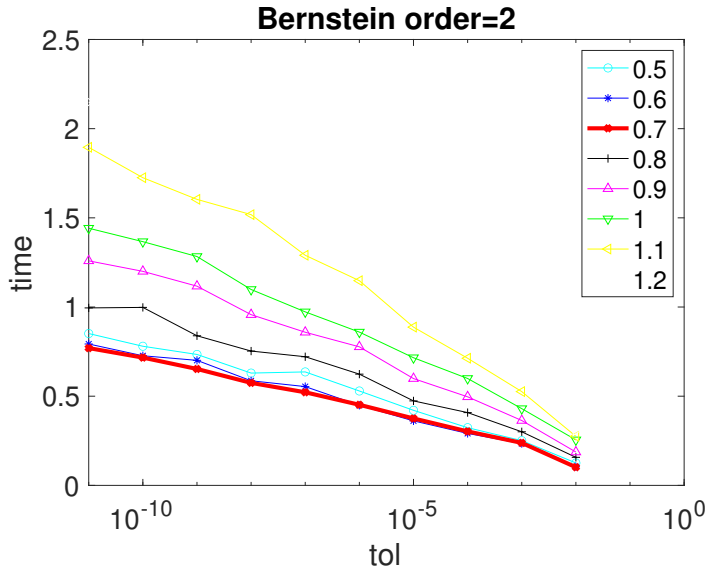



Figure 23: Bernstein, benchmark 8, cpu time

In figure 23, taking a $\omega = 0.7$ lead to the fastest cpu time. Hence in this experiment, set a 0.7 shift may surely helps the iteration performance.

# 6 Summary

In this work, we referred to some theses about polynomial preconditioners, looked for efficiency choices in order to raise the performance of iteration solvers. Studies of approximation theory gave us some common tools, Neumann expansion, Chebyshev polynomial, and Bernstein basis, just to name a few. By taking these different preconditioners, we examined on linear system with large, symmetric, sparse and positive graph laplacian matrix by pcg mainly. To judge the performance by iteration times and cpu time under the help of matlab.

Various of sub-experiments were tested in section 4, here are the concise

conclusion:

- Neumann series.
    1. The degree is higher, iteration number is smaller.
    2. Odd-degree has less iteration steps than even-degree.
- Weighted.
    1. Odd-degree has a lowest iteration number and cpu time at $\omega = 1$.
    2. Even-degree has a lowest iteration number and cpu time at about $\omega = 0.7, 0.8$.
    3. In bicgstab with negative $\omega$, lower degree approximation has not only less steps and also iterate faster.
    4. Negatively $\omega$ does not faster.
- Chebyshev polynomial.
    1. Chebyshev polynomial does not make the iteration better when setting $\lambda_1$ and $\lambda_n$ as parameters. Even if we added weights when splitting, it still corresponds with Saad's outcome.
- Bernstein Polynomial.
    1. Adding weights in Bernstein polynomial has a similar outcome to Neumann expansion, that is, given a $\omega < 1$ may probably iterate faster.

Overall, we can see the differences to these polynomial preconditioners. In this table, we compare the performance of different cases in degree 2.

| | iteration times | cpu time |
|---|---|---|
| Bernstein with $\omega = 0.7$ | 67 | 0.3853 |
| Neumann with $\omega = 0.7$ | 172 | 0.8077 |
| Neumann series | 280 | 1.4692 |
| Chebyshev with $\omega = 0.7$ | 554 | 1.6912 |
| Without preconditioner | 828 | 1.6063 |

Table 2: Comparison

We can see that taking a weighted Bernstein polynomial as preconditioner may lead to the best result in table 2. In conclusion, giving a preconditioner will accelerate the iteration 'sometimes'. It may took too much time on finding complicated coefficients or doing matrix multiplication, it violates the origin intension. We shall have decided the degree and considered the balance between iteration number and cpu time. Furthermore, adding a positive weight near or smaller than 1 in our algorithm raises the performance. Perhaps these decisions are case by case, fortunately, they are sometimes observable by doing the experiments as above.

## Appendix A

Chebyshev polynomial of the first kind, the cosine one, has the recursion

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x). \tag{24}$$

The first few Chebyshev polynomials of the first kind(cosine) are:

$$T_0 = 1$$

$$T_1 = x$$

$$T_2 = 2x^2 - 1$$

$$T_3 = 4x^3 - 3x$$

$$T_4 = 8x^4 - 8x^2 + 1$$

$$T_5 = 16x^5 - 20x^3 + 5x$$

## Appendix B

The $n+1$ **Bernstein basis polynoimals** of degree $n$ is defined as

$$b_{v,n}(x) = \binom{n}{v} x^v (1-x)^{n-v}, v = 0, \ldots, n.$$

So the first few Bernstein basis polynomials for blending 1,2,3 or 4 values together are:

$$b_{0,0}(x) = 1$$

$$b_{0,1}(x) = 1 - x, \qquad b_{1,1}(x) = x$$

$$b_{0,2}(x) = (1-x)^2, \quad b_{1,2}(x) = 2x(1-x), \quad b_{2,2}(x) = x^2$$

$$b_{0,3}(x) = (1-x)^3, \quad b_{1,3}(x) = 3x(1-x)^2, \quad b_{2,3}(x) = 3x^2(1-x), \quad b_{3,3}(x) = x^3.$$

Let $f$ be a continuous function on the interval [0,1] . Consider the Bernstein polynomial

$$B_n(f)(x) = \sum_{\nu=0}^{n} f\left(\frac{\nu}{n}\right) b_{\nu,n}(x)$$

It can be shown that

$$\lim_{n\to\infty} B_n(f) = f$$

uniformly on the interval [0,1] .

# References


[1] O. G. Johnson, C. A. Micchelli and G. Paul, *Polynomial Preconditioners for Conjugate Gradient Calculations*, Society for Industrial and Applied Mathematics, SIAM Journal on Numerical Analysis, Vol. 20, No. 2 (1983), pp. 362-376.

[2] A. K. Nandi, *Iterative Methods for Linear and Multilinear Systems Based on Splittings*, Birla Institute of Technology and Science, Pilani.

[3] H. Wang, S. Xiang, *On the convergence rates of Legendre approximation*, Math. Computation, 81 (2012), pp. 861-877.

[4] J.-J. Climent, N. Thome and Y. Wei, *A geometrical approach on generalized inverses by Neumann-type series*, Linear Algebra and its Applications, 332–334 (2001), pp. 533-540.

[5] M.H. ,Mudde (2017), *Chebyshev approximation*, Master Thesis, University of Groningen, Faculty of Science and Engineering, Groningen.

[6] Weisstein, Eric W., *Legendre Polynomial*, MathWorld, https://mathworld.wolfram.com/LegendrePolynomial.html

[7] J. R Shewchuk (1994), *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain*, Technical Report, Carnegie Mellon University, USA.

[8] Y. Saad (2003), *Iterative Method for Sparse Linear System* $2^{nd}$, Society for Industrial and Applied Mathematics.

[9] A. Pyzara, B. Bylina and J. Bylina, *The influence of a matrix condition number on iterative methods' convergence*, Federated Conference on

Computer Science and Information Systems (FedCSIS) (2011), pp. 459-464.

[10] O. Leary, P. Dianne (1991), *Yet another polynomial preconditioner for the conjugate gradient algorithm*, Linear Algebra and its Applications, pp. 377-388.

[11] G. Mayer, *On the convergence of the Neumann series in interval analysis*, Linear Algebra and its Applications 65 (1985), pp. 63-70.

[12] H. B. Jebreen, Y. Chalco-Cano (2018), *An Improved Computationally Efficient Method for Finding the Drazin Inverse*, Discrete Dynamics in Nature and Society.

[13] M. N. Bernstein (2020), *The Graph Laplacian*, https://mbernste.github.io/posts/laplacian_matrix/

[14] K. C. Das, *The Laplacian spectrum of a graph*, Computers & Mathematics with Applications, Volume 48, Issues 5-6 (2004), pp. 715-724.

[15] N. K. Vishnoi, *Lx = b Laplacian Solvers and Their Algorithmic Applications*, Foundations and Trends in Theoretical Computer Science, Vol. 8, Nos. 12 (2012), pp. 1-141

[16] M. W. Mahoney, L. Orecchia, and N. K. Vishnoi, *A spectral algorithm for improving graph partitions*, Journal of Machine Learning Research, vol. 13 (2012), pp. 2339–2365.

[17] R. Bisseling, J.G. Vors t(1988), *Parallel LU Decomposition on a Transputer Network*, Shell Conference.

[18] Y. Saad (1984), *Practical Use of Polynomial Preconditionings for the Conjugate Gradient Method*, SIAM J. Sci. and Stat. Comput., 6(4), pp. 865-881.

[19] R.T. Farouki (2012), *The Bernstein polynomial basis: A centennial retrospective*, Comput. Aided Geom. Des., 29, pp 379-419.

[20] Y. Saad (1981), *Krylov Subspace Methods for Solving Large Unsymmetric Linear Systems*, Department of Computer Science, University of Illinois at Urbana-Champaign.

[21] Y. Saad and M. Sosonkina (2001), *Enhanced preconditioners for large sparse least squares problems*, Minnesota Supercomputer Institute, University of Minnesota, Minneapolis, MN.

[22] K. Wu, Y. Saad, and A. Stathopoulos (1998), *Inexact Newton Preconditioning Techniques for Eigenvalue Problems*, Minnesota Supercomputer Institute, University of Minnesota, Minneapolis, MN, ETNA, vol. 7, pp. 202-214.

[23] O. Axelsson, G. Lindskog, *On the rate of convergence of the preconditioned conjugate gradient method*, Numerische Mathematik 48.5 (1986), pp. 499-523.

[24] G. H. Golub, Q. Ye, *Inexact preconditioned conjugate gradient method with inner-outer iteration*, SIAM Journal on Scientific Computing 21.4 (1999), pp. 1305-1320.

[25] L. Y. Kolotilina, A. Y. Yeremin, *Factorized Sparse Approximate Inverse Preconditionings I. Theory*, SIAM J. Matrix Anal. Appl., 14(1), pp. 45-58.