Computional Homework14 Report

Student: 纪浩正, jihz2023@mail.sustech.edu.cn

1 Conjugate Gradient Method

1.1 Solving $Ax = b$ with Known Conjugate Directions

We consider the quadratic function

$$f(x) = \frac{1}{2}x^T Ax - b^T x, \tag{1}$$

whose gradient is

$$\nabla_x f = Ax - b. \tag{2}$$

Since A is symmetric positive definite, we can find inverse matrix P s.t.

$$P^T AP = \Lambda \tag{3}$$

where

$$P = \begin{pmatrix} P_0 & P_1 & \dots & P_n \end{pmatrix}, \quad P^T = \begin{pmatrix} P_0^T \\ P_1^T \\ \vdots \\ P_n^T \end{pmatrix}. \tag{4}$$

To solve $Ax = b$, we minimize $f(x)$ so that $\nabla f = 0$. let $x = Py$:

$$f(x) = \frac{1}{2}(Py)^T A(Py) - b^T(Py) \tag{5}$$

$$= \frac{1}{2}y^T P^T APy - b^T Py. \tag{6}$$

then

$$f(x) = \frac{1}{2}y^T \Lambda y - b^T Py. \tag{7}$$

At the minimum:

$$\Lambda y = P^T b, \qquad P^T APy = P^T b, \tag{8}$$

so the solution in spectral form is

$$y = \begin{pmatrix} \dfrac{P_0^T b}{P_0^T A P_0} \\ \vdots \\ \dfrac{P_n^T b}{P_n^T A P_n} \end{pmatrix}, \quad x = Py = \begin{pmatrix} P_0 & P_1 & \cdots & P_n \end{pmatrix} \begin{pmatrix} \dfrac{P_0^T b}{P_0^T A P_0} \\ \vdots \\ \dfrac{P_n^T b}{P_n^T A P_n} \end{pmatrix}. \tag{9}$$

Therefore

$$x = \sum_{i=1}^{n} P_i \frac{P_i^T b}{P_i^T A P_i}. \tag{10}$$

The basis vectors satisfy the A-conjugacy condition

$$P_i^T A P_j = \begin{cases} 0, & i \neq j, \\ \Lambda_{ii}, & i = j. \end{cases} \tag{11}$$

Assuming we have a set of conjugate directions $\{P_0, \ldots, P_n\}$, the iterative update reads

$$x_1 = x_0 - \alpha_0 P_0, \tag{12}$$

$$x_2 = x_1 - \alpha_1 P_1, \tag{13}$$

$$\vdots \tag{14}$$

$$x_n = x_{n-1} - \alpha_{n-1} P_{n-1} = x_0 - \sum_{i=0}^{n-1} \alpha_i P_i = x_0 - Pv, \tag{15}$$

where

$$v = \begin{pmatrix} \alpha_0 \\ \alpha_1 \\ \vdots \\ \alpha_n \end{pmatrix}. \tag{16}$$

We want x_n to be the exact solution. Substituting:

$$f(x_n) = \frac{1}{2} x_n^T A x_n - b^T x_n \tag{17}$$

$$= \frac{1}{2}(x_0 - Pv)^T A (x_0 - Pv) - b^T (x_0 - Pv) \tag{18}$$

$$= \frac{1}{2}\left(x_0^T A x_0 + v^T P^T A P v - 2 x_0^T A P v\right) - b^T A x_0 + b^T P v \tag{19}$$

$$= \frac{1}{2} v^T P^T A P v - (A x_0 - b)^T P v + \text{constant}. \tag{20}$$

Setting the gradient with respect to v to zero yields

$$\alpha_i = \frac{(Ax_0 - b)^T P_i}{P_i^T A P_i}. \tag{21}$$

Thus,

$$x_n = x_0 - \sum_{i=0}^{n-1} \frac{(Ax_0 - b)^T P_i}{P_i^T A P_i} P_i. \tag{22}$$

We have finished discussing how to obtain the solution of $Ax = b$ using a given conjugate vector set P. However, when the original conjugate directions are not available, we need an iterative strategy to generate new conjugate directions.

1.2 Generating Conjugate Directions Iteratively

When $\{P_i\}$ is not known, CG builds them from residuals. Let

$$r_0 = b - Ax_0, \qquad P_0 = r_0. \tag{23}$$

We update

$$x_1 = x_0 - \alpha_0 P_0, \tag{24}$$

$$f(x_0 - \alpha_0 P_0) = \frac{1}{2}(P_0^T A P_0)\alpha_0^2 + r_0^T P_0 \alpha_0 + \text{constant}, \tag{25}$$

and maximize in α_0:

$$\alpha_0 = -\frac{P_0^T r_0}{P_0^T A P_0}. \tag{26}$$

Next

$$r_1 = b - Ax_1, \qquad P_1 = r_1 + \beta_0 P_0, \tag{27}$$

with conjugacy condition $P_1^T A P_0 = 0$, giving

$$\beta_0 = -\frac{r_1^T A P_0}{P_0^T A P_0}. \tag{28}$$

```
def ConjugateGradient(self,b):
## Initialization
# x0 = 0, r0 = b - A*x0 = b, p0 = r0
    x=np.zeros((self.n))
    start_time=time.time()
    r=b
    p=r.copy()
```

```
    dot_r0=np.dot(r,r)
```

In general, if we already have the conjugate directions $P_0, \ldots, P_{k-1}$, then the k-th iteration proceeds as follows. We first update the solution by

$$x_k = x_{k-1} - \alpha_{k-1} P_{k-1}.$$

To determine the optimal step size α_{k-1}, we minimize the quadratic functional $f(x) = \frac{1}{2} x^T A x - b^T x$ along search direction P_{k-1}, i.e.

$$\nabla_{\alpha_{k-1}} f(x_k) = 0, \qquad \nabla_{\alpha_{k-1}} \left\{ \tfrac{1}{2} (P_{k-1}^T A P_{k-1}) \alpha_{k-1}^2 + r_{k-1}^T P_{k-1} \alpha_{k-1} + \text{constant} \right\} = 0.$$

Solving yields the step size

$$\alpha_{k-1} = -\frac{P_{k-1}^T r_{k-1}}{P_{k-1}^T A P_{k-1}} = -\frac{(r_{k-1} + \beta_{k-2} P_{k-2})^T r_{k-1}}{P_{k-1}^T A P_{k-1}} = -\frac{r_{k-1}^T r_{k-1}}{P_{k-1}^T A P_{k-1}}.$$

(From the second equality, we use Lemma 1, which will be proved in the next section.)

```
## CG Iteration
    for ii in range(self.n):
        Ap=self.cal_Ax(p)
        alpha=dot_r0/np.dot(p,Ap)
## Update x_{k}
        x+=alpha*p
```

The residual is then updated as

$$r_k = b - A x_k = b - A(x_{k-1} - \alpha_{k-1} P_{k-1}) = r_{k-1} + \alpha_{k-1} A P_{k-1}.$$

```
## Update r_{k}
        r-=alpha*Ap
```

Finally, we generate a new conjugate direction by

$$P_k = r_k + \beta_{k-1} P_{k-1}.$$

To ensure A-conjugacy of directions, i.e. $P_i^T A P_j = 0$ for $i \neq j$, we impose

$$P_k^T A P_{k-1} = 0, \qquad (r_k + \beta_{k-1} P_{k-1})^T A P_{k-1} = 0,$$

which gives

$$\beta_{k-1} = -\frac{r_k^T A P_{k-1}}{P_{k-1}^T A P_{k-1}} = -\frac{r_k^T (r_k - r_{k-1})/\alpha_{k-1}}{P_{k-1}^T A P_{k-1}} = \frac{r_k^T r_k}{P_{k-1}^T A P_{k-1}} \cdot \frac{P_{k-1}^T A P_{k-1}}{P_{k-1}^T r_{k-1}} = \frac{r_k^T r_k}{r_{k-1}^T r_{k-1}}.$$

(From the second equality, we use Lemma 2, which will be proved in the next section.)

```
## Update p_{k}
        dot_r1=np.dot(r,r)
        beta=dot_r1/dot_r0
        p=r+beta*p
        dot_r0=dot_r1
        if dot_r1<=self.tol**2:
            break
    end_time=time.time()
    spend=end_time-start_time
    return x,spend,np.sqrt(dot_r1)
```

The iteration can be summarized inductively as

$$\begin{aligned}
r_0 &= b - Ax_0, & P_0 &= r_0, \\
x_1 &= x_0 - \alpha_0 P_0, & \alpha_0 &= -\frac{P_0^T r_0}{P_0^T A P_0}, \\
r_1 &= b - Ax_1, & & \\
P_1 &= r_1 + \beta_0 P_0, & \beta_0 &= -\frac{r_1^T A P_0}{P_0^T A P_0}, \\
&\vdots & & \\
x_{k+1} &= x_k - \alpha_k P_k, & \alpha_k &= -\frac{P_k^T r_k}{P_k^T A P_k}, \\
r_{k+1} &= b - Ax_{k+1} = r_k - \alpha_k A P_k, & & \\
P_{k+1} &= r_{k+1} + \beta_k P_k, & \beta_k &= -\frac{r_{k+1}^T A P_k}{P_k^T A P_k}.
\end{aligned}$$

We have completed the main derivation of the iteration scheme, but the most crucial task remains—proving the orthogonality of residuals and the conjugacy of search directions, which

ensures that the conjugacy relationship is preserved throughout the iterations.

1.3 Proof of Residual Orthogonality and A-Conjugacy

Key lemmas (orthogonality and conjugacy)

- $P_i^T r_j = 0,\ i < j$
- $r_i^T r_j = 0,\ i \neq j$
- $P_j^T A P_i = 0,\ i \neq j$

Verification for $i = 1$

$$P_0^T r_1 = P_0^T (r_0 + \alpha_0 A P_0) = P_0^T r_0 + \alpha_0 P_0^T A P_0 = P_0^T r_0 - \frac{P_0^T r_0}{P_0 A P_0} P_0^T A P_0 = 0, \tag{29}$$

$$r_0^T r_1 = P_0^T r_1 = 0, \tag{30}$$

$$P_0^T A P_1 = P_0^T A (r_1 + \beta_0 P_0) = P_0^T A r_1 + \beta_0 P_0^T A P_0 = P_0^T A r_1 - \frac{r_1^T A P_0}{P_0^T A P_0} P_0^T A P_0 = 0. \tag{31}$$

Assume lemmas hold for $i \leq k - 1$.

Then for **lemma 1**:

For i<k-1, by Lemma 1 and Lemma 3 applied to step (k-1),

$$P_i^T r_k = P_i^T \left(r_{k-1} + \alpha_{k-1} A P_{k-1}\right) = P_i^T r_{k-1} + \alpha_{k-1} P_i^T A P_{k-1} = 0. \tag{32}$$

For i = k-1,

$$P_{k-1}^T r_k = P_{k-1}^T \left(r_{k-1} + \alpha_{k-1} A P_{k-1}\right) = P_{k-1}^T r_{k-1} - \frac{P_{k-1}^T r_{k-1}}{P_{k-1}^T A P_{k-1}} P_{k-1}^T A P_{k-1} = 0. \tag{33}$$

Lemma 2. By Lemma 1 applied to step (k), for any $i < k$ we have:

$$0 = P_i^T r_k = (r_i + \beta_{i-1} P_{i-1})^T r_k \tag{34}$$

$$= r_i^T r_k. \tag{35}$$

Lemma 3:

For i<k-1, by Lemma 2 applied to step (k),

$$P_i^T A P_k = P_i^T A(r_k + \beta_{k-1} P_{k-1}) = (AP_i)^T r_k + \beta_{k-1} P_i^T A P_{k-1} = (\frac{r_{i+1} - r_i}{\alpha_i})^T r_k = 0$$

For i = k-1,

$$P_{k-1}^T A P_k = P_{k-1}^T A(r_k + \beta_{k-1} P_{k-1}) = P_{k-1}^T A r_k - \frac{r_k^T A P_{k-1}}{P_{k-1}^T A P_{k-1}} P_{k-1}^T A P_{k-1} = 0$$

then we done the check

Thus all three lemmas hold by induction, confirming that $\{P_0, \ldots, P_n\}$ are mutually A-conjugate.

2 Tridiagonal Method

We consider a linear system $A\mathbf{x} = \mathbf{b}$ where A is an $n \times n$ tridiagonal matrix of the form

$$A = \begin{pmatrix} 2 & -1 & & & 0 \\ -1 & 2 & -1 & & \\ & -1 & 2 & \ddots & \\ & & \ddots & \ddots & -1 \\ 0 & & & -1 & 2 \end{pmatrix}.$$

For convenience, we denote the sub-diagonal, diagonal and super-diagonal by

$$a_i = -1, \qquad B_i = 2, \qquad c_i = -1, \qquad i = 1, \ldots, n.$$

The Tridiagonal Method performs forward elimination followed by back substitution.

Forward Elimination

We rewrite the system in the form

$$B_i x_i + c_i x_{i+1} + a_i x_{i-1} = b_i.$$

In forward elimination, we eliminate x_{i-1} from equation i. Define modified coefficients

$$\gamma_0 = B_0, \qquad \delta_0 = \frac{c_0}{\gamma_0}, \qquad y_0 = \frac{b_0}{\gamma_0}.$$

```
def TridiagonalMethod(self,b):
## Initialization
# gamma0 = B0, delta0 = c0/gamma0, y0=b0/gamma0
    a,B,c=-np.ones(self.n),2*np.ones(self.n),-np.ones(self.n)
    start_time=time.time()
    gamma,delta,y,x=np.zeros(self.n),np.zeros(self.n),np.zeros(self.n),np.zeros((self.n))
    beta=a
    gamma[0]=B[0]
    delta[0]=c[0]/gamma[0]
    y[0]=b[0]/gamma[0]
```

For $i = 1, 2, \ldots, n-1$, we compute

$$\gamma_i = B_i - a_i \delta_{i-1},$$

$$\delta_i = \frac{c_i}{\gamma_i},$$

$$y_i = \frac{b_i - a_i y_{i-1}}{\gamma_i}.$$

```
    for ii in range(1,self.n):
## Update y_{k}
# gamma_{k}=B_{k}-a_{k}*delta_{k-1}, delta_{k}=c_{k}/gamma_{k},
     y_{k}=(b_{k}-a_{k}y_{k-1})/gamma_{k}
        gamma[ii]=B[ii]-a[ii]*delta[ii-1]
        delta[ii]=c[ii]/gamma[ii]
        y[ii]=(b[ii]-beta[ii]*y[ii-1])/gamma[ii]
```

Back Substitution

Starting from the last unknown,

$$x_{n-1} = y_{n-1},$$

we compute each remaining component using

$$x_i = y_i - \delta_i x_{i+1}, \qquad i = n-2, \ldots, 0.$$

```
## Update x_{k}=y_{k}-delta_{k}x_{k+1}
    x[self.n-1]=y[self.n-1]
    for ii in range(self.n-2,-1,-1):
        x[ii]=y[ii]-delta[ii]*x[ii+1]
    end_time=time.time()
```

```
        res=np.linalg.norm(self.cal_Ax(x)-b)
        spend=end_time-start_time
        return x,spend,res
```

3 Results

3.1 Verification of the Numerical Solutions

To verify the correctness of both the tridiagonal solver and the conjugate gradient (CG) solver, we test the systems of size $n = 3, 4, 5$. The following code snippet calls the two solvers respectively and prints their solutions.

```
solver = SPD()
dim = np.array([3, 4, 5])

for ii in range(3):
    x1, _, _ = solver.solve(dim[ii], mode=0) # tridiagonal solver
    x2, _, _ = solver.solve(dim[ii], mode=1) # conjugate gradient solver
    print(f'for dim={dim[ii]}, \nuse tridiagonal method we can get :{x1}\n'
          f'use conjugate gradient method we can get :{x2}\n\n')
```

Figure 1 shows that for all tested dimensions, the solutions obtained by the tridiagonal method and the conjugate gradient method are identical, confirming the correctness of both algorithms.

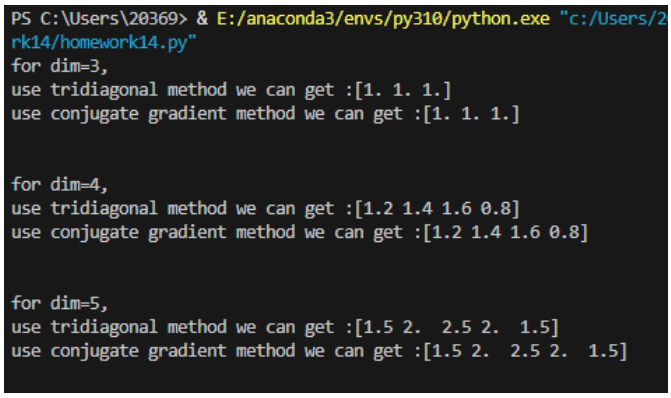

Figure 1 Comparison of numerical solutions from the tridiagonal solver and the conjugate gradient solver. Both methods produce identical results for $n = 3, 4, 5$.

3.2 Comparison Between Tridiagonal Solver and Conjugate Gradient Method

We now compare the computational performance of the tridiagonal solver and the conjugate gradient (CG) method. Because the underlying matrix is symmetric positive definite and tridi-

agonal, the tridiagonal algorithm yields the exact solution in $\mathcal{O}(n)$ time. In contrast, the CG method also obtains the exact solution but requires n iterations.

Time Comparison The computation time for dimensions from $n = 3$ to $n = 10000$ is shown in Figure 2. The tridiagonal method remains significantly faster due to its optimal $\mathcal{O}(n)$ complexity. The CG method is slower for this special case, although it still exhibits good scaling.

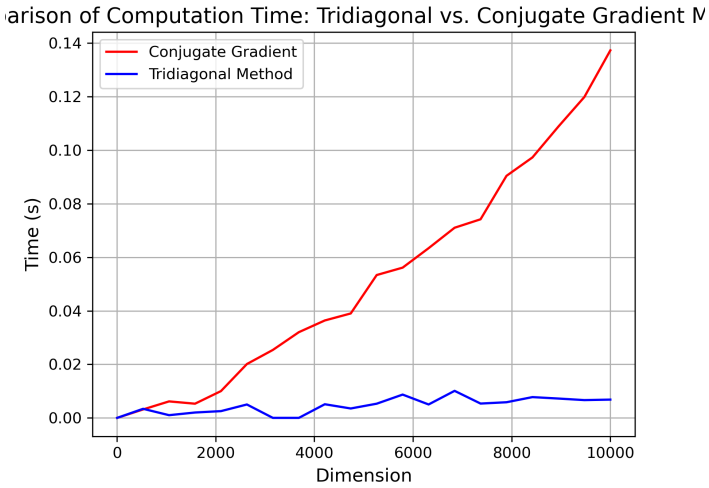

Figure 2 Computation time comparison between the tridiagonal solver and the conjugate gradient solver. The tridiagonal method is optimal for tridiagonal matrices.

Residual Analysis for Conjugate Gradient Figure 3 shows the numerical residual of the conjugate gradient method. After n iterations, CG recovers the exact solution (up to machine precision), as expected for SPD matrices.

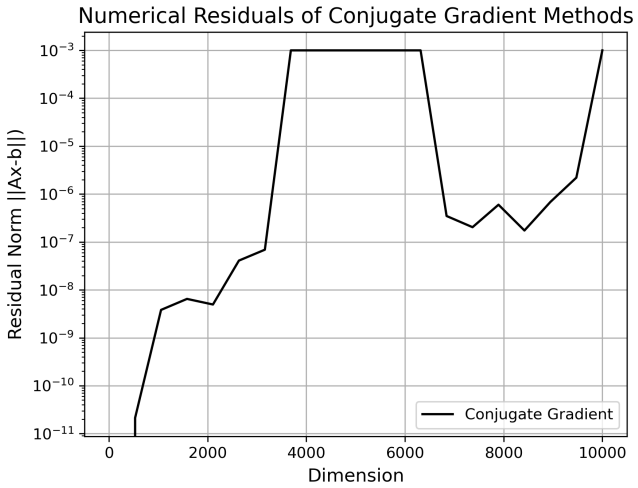

Figure 3 Numerical residual of the conjugate gradient method. Residual decreases and reaches machine precision after n iterations.

```
dim_list = np.linspace(3, 10000, 20).astype(int)
t = np.zeros((len(dim_list), 2))
x = t.copy()
res = np.zeros((len(dim_list)))

for ii in range(len(dim_list)):
    x1, t[ii, 0], _ = solver.solve(dim_list[ii], mode=0)
    x2, t[ii, 1], res[ii] = solver.solve(dim_list[ii], mode=1)
    x[ii, 0], x[ii, 1] = np.linalg.norm(x1), np.linalg.norm(x2)
```