The Method of Conjugate Gradients

Pratik Aghor

December 2015

1 Introduction

1.1 The Problem Statement

Without wasting reader's time, I would directly like to get to the point. Conjugate Gradient (henceforth referred to as CG) is one of the most popular iterative methods to solve a system described by

$$Ax = b$$

Where A = Symmetric Positive Definite Matrix (known)
b = a vector (known)
x = the solution (unknown)

Such systems generally arise in many numerical methods such as Finite Difference and Finite Element Methods, used to solve PDE's. In the following document, we would not bother about **how we got the above mentioned form**, for there may exist various ways to get there, but we will try to actually get our hands dirty and solve the damn linear system! So most of the focus will be on to **what to do if we come across such a linear system that is described above.**

Whatever I am going to write here, is more or less taken from "An introduction to the conjugate gradient method without the agonizing pain."[?]. One should definitely read this paper in order to get the geometrical picture. All I have tried here to do is to **add** to [?], for example, I have tried to make the algebraic steps clearer and written what I understood from the paper.

1.2 The Quadratic Form

The quadratic form is a scalar function of 'x', the unknown vector.

The quadratic form is given by

$$f(x) = \frac{1}{2}x^T Ax - b^T x + c \tag{1.1}$$

where A is a matrix, b and x are vectors and c is a constant.

Throughout this document, we will assume $c = 0$.

Now a natural question might arise in a curious mind and that might be, **"Why this specific form and not something else and what it has to do with CG?"**

We will prove in this section, that minimizing the quadratic form leads to the desired solution of $Ax = b$, and that, my dear curious reader, is the starting point of the method of conjugate gradients.

1.2.1 Proof: Minimizing the quadratic form is equivalent to solving the system $Ax = b$

Let A be a symmetric matrix.
$\Rightarrow A = A^T$
Let x_0 be a solution of $Ax = b$
$\Rightarrow Ax_0 = b$
Now, letting e to be the error term,

$$\begin{aligned}
f(x_0+e) &= \frac{1}{2}(x_0+e)^T A(x_0+e) - b^T(x_0+e) + c \\
&= \frac{1}{2}x_0^T Ax_0 + e^T Ax_0 + \frac{1}{2}e^T Ae - b^T x_0 - b^T e + c...(asA^T = A) \\
&= \frac{1}{2}x_0^T Ax_0 - b^T x_0 + c + e^T b + \frac{1}{2}e^T Ae - b^T e \\
&= f(x_0) + \frac{1}{2}e^T Ae
\end{aligned}$$

But as we have assumed A to be a symmetric **positive definite** matrix, $\frac{1}{2}e^T Ae > 0 \;\forall\; e \neq 0$
Therefore

$$f(x_0+e) > f(x_0)$$

$\forall\; e \neq 0$
Hence $f(x_0)$ is a minima of $f(x)$ if $Ax_0 = b$ is proved.
NOTE: Geometrically, A being positive definite matrix percolates into the fact that $f(x)$ is a paraboloid, that is concave up, as shown in the following figure.

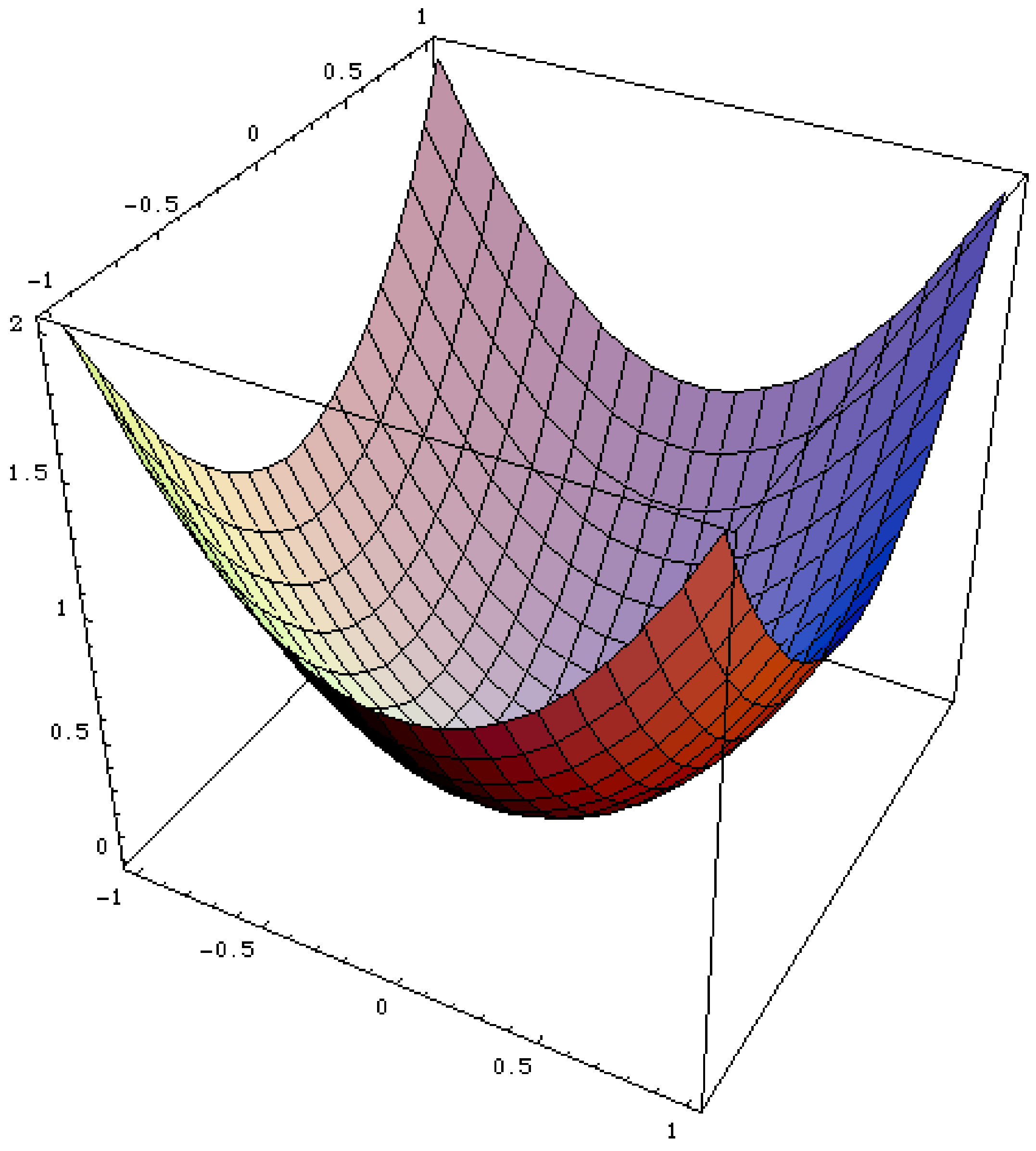

Figure 1.1: form of f(x)

1.2.2 Another important result

We will state another important result here without proof.

That is,

$$f^{'}(x) = \frac{1}{2}A^T x + \frac{1}{2}Ax - b \tag{1.2}$$

if A is symmetric, we get,

$$f^{'}(x) = Ax - b \tag{1.3}$$

2 The Method of Steepest Descent

2.1 Introduction

In the method of steepest descent, we generally begin at an arbitrary point x_0 which almost always is not a solution and then we slide down to the bottom of the paraboloid.

The logical way to go about this is as follows. We start with a point x_0. We determine the direction in which the slope changes the fastest. We take a step in precisely that direction. We keep doing the above procedure until we are convinced that we are "close enough" to the solution. (What do I mean by "close enough"? I will shortly make it clear.)

The direction in which $f(x)$ decreases at the fastest rate is found by taking the gradient of $f(x)$ at that point. $f(x)$ "decreases" quickly in the direction opposite to that.

Hence, the required direction is

$$-f^{'}(x_{(i)}) = b - Ax_{(i)} \tag{2.1}$$

2.1.1 A few useful definitions

- error term is given by
$$e_{(i)} = x_{(i)} - x$$
- residual is
$$r_{(i)} = b - Ax_{(i)}$$
- as $e_{(i)} = x_{(i)} - x$,
$$Ae_{(i)} = Ax_{(i)} - Ax$$
$\Rightarrow Ae_{(i)} = Ax_{(i)} - b \Rightarrow -Ae_{(i)} = r_{(i)}$ Therefore, one can think of residual as the error transformed by A into the space where b belongs.
- We can also see from above that $r_{(i)} = -f^{'}(x_{(i)})$
- Therefore, we move in the direction $r_{(i)}$

Therefore

$$x_{(i+1)} = x_{(i)} + \alpha_{(i)} r_{(i)}$$

Now, having determined **where** to move, we need to determine **how much** to move. There is a procedure so popular for this, it has a name! The procedure is named as **the line search**

2.1.2 Line Search Procedure

This procedure determines the value of $\alpha_{(i)}$ that minimizes $f(x)$ along a line.

Geometrically, a plane will cut a paraboloid in a parabolic curve. We want $\alpha_{(i)}$ that finds the minima of this parabola.

From calculus, we know f is minimized by α when the directional derivative is equal to zero. Therefore,

$$\frac{d}{d\alpha_{(i)}}(f(x_{(i+1)}) = 0$$

$$f^{'}(x_{(i+1)})^T \frac{d}{d\alpha_{(i)}}(x_{(i)} = 0$$

$$f^{'}(x_{(i+1)})^T r_{(i)} = 0$$

...(using chain rule)

Therefore, we see that $\alpha_{(i)}$ should be chosen in such a manner that $r_{(i+1)}$ is orthogonal to $f^{'}(x_{(i)})$, that is, $r_{(i)}$.

Therefore

$$\begin{aligned}
r_{(i+1)}r_{(i)} &= 0 \\
(b - Ax_{(i+1)})r_{(i)} &= 0 \\
(b - A(x_{(i)} + \alpha_{(i)}r_{(i)}))r_{(i)} &= 0 \\
(b - A(x_{(i)})^T r_{(i)} - \alpha_{(i)}(Ar_{(i)})^T r_{(i)} &= 0 \\
(b - A(x_{(i)})^T r_{(i)} &= \alpha_{(i)}(Ar_{(i)})^T r_{(i)} \\
\alpha_{(i)} &= \frac{r_{(i)}^T r_{(i)}}{r_{(i)}^T A r_{(i)}}
\end{aligned}$$

Therefore,

$$\alpha_{(i)} = \frac{r_{(i)}^T r_{(i)}}{r_{(i)}^T A r_{(i)}} \tag{2.2}$$

2.2 Putting it all together

Therefore, in a concise manner, the method of steepest descent can be formulated as follows.

$$r_{(i)} = b - Ax_{(i)} \tag{2.3}$$

$$\alpha_{(i)} = \frac{r_{(i)}^T r_{(i)}}{r_{(i)}^T A r_{(i)}} \tag{2.4}$$

$$x_{(i+1)} = x_{(i)} + \alpha_{(i)}r_{(i)} \tag{2.5}$$

3 The Method of Conjugate Directions

The method of steepest descent works fine, but is there a way to make it better? If there exists a way of choosing the directions smartly, why not do it? What is the cost that we have to pay?

3.1 The Notion of Conjugacy

Lets pick up any n orthogonal search directions. Lets name them as $d_0, d_1, ..., d_{n-1}$.
In each direction, we take precisely one step and try to line up ourselves with the final solution x. After, n steps we will be where we need to be, at the minima of the paraboloid!

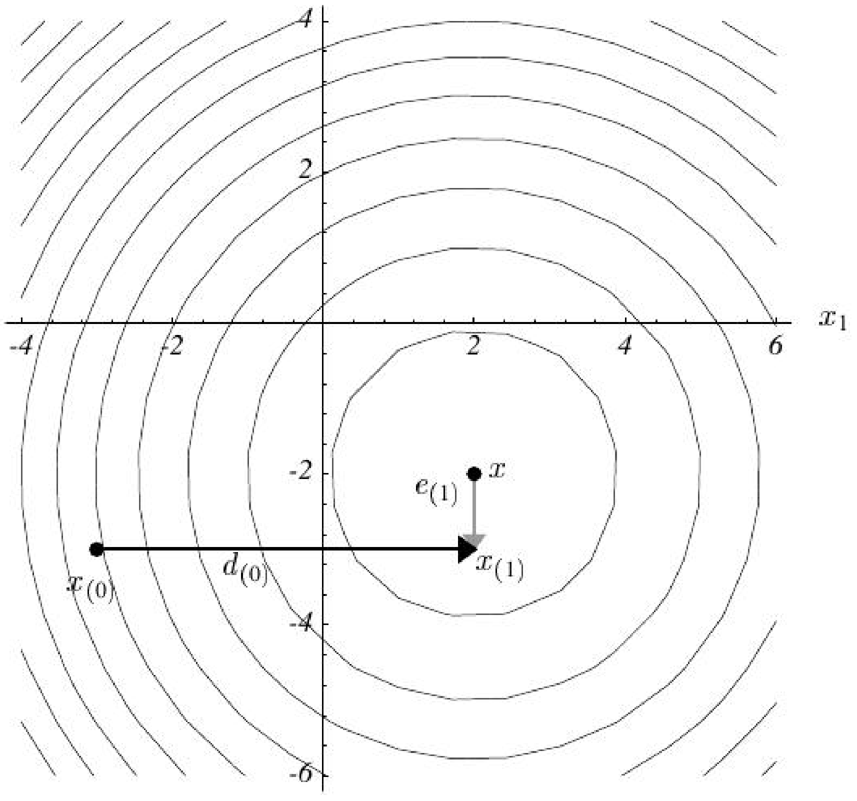

Figure 3.1: Orthogonal Search directions

In general, we choose next point as follows:

$$x_{(i+1)} = x_{(i)} + \alpha_{(i)} d_{(i)}$$

To find the value of $\alpha_{(i)}$, we wisely use the fact that $e_{(i+1)}$ and $d_{(i)}$ are orthogonal to each other.

$$\begin{aligned} d_{(i)}^T e_{(i+1)} &= 0 \\ d_{(i)}^T (e_{(i)} + \alpha_{(i)} d_{(i)}) &= 0 \end{aligned}$$

$\Rightarrow$

$$\alpha_{(i)} = -\frac{d_{(i)}^T e_{(i)}}{d_{(i)}^T d_{(i)}} \tag{3.1}$$

But (3.1) is a situation like Catch-22! We need to realize that we are in a paradoxical situation here. The dilemma is as follows.

We cannot proceed to $x_{(i+1)}$ until and unless we know the value of $\alpha_{(i)}$. But to know the value of $\alpha_{(i)}$, equation (3.1) demands for us to know the value of $e_{(i)}$. Had we known the value of $e_{(i)}$ a priori, we wouldn't be doing all this. We would simply find the solution by adding $e_{(i)}$ to the value of $x_{(i)}$!

So unfortunately, our accomplishment here is useless.

There is one way to break this paradox. And that way is to make the search directions, so called **A-orthogonal** instead of just orthogonal.

Two vectors $d_{(i)}$ and $d_{(j)}$ are said to be **conjugate** or **A-orthogonal** if

$$d_{(i)}^T A d_{(j)} = 0 ... \forall i \neq j \tag{3.2}$$

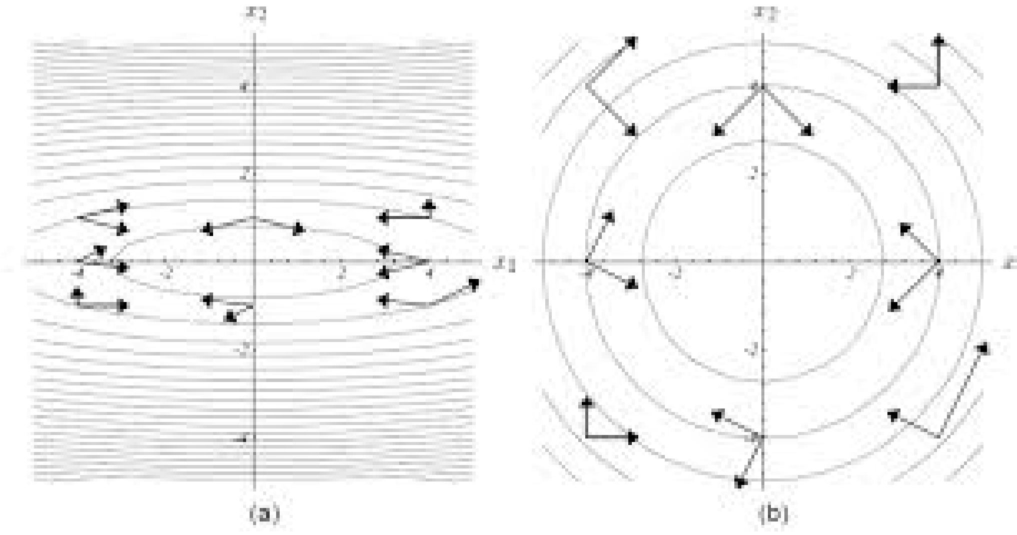

Figure 3.2: vectors in (a) are conjugate or A-orthodonal because if I stretch the figure(i.e., multiply by A) I get figure (b), where they become orthogonal

Again, to find the minima of the paraboloid of $f(x)$, we set the directional derivative to be zero.

$$\frac{d}{d\alpha_{(i)}}(f(x_{(i+1)}) = 0$$

$$f^{'}(x_{(i+1)})^T \frac{d}{d\alpha_{(i)}}(x_{(i)}) = 0$$

$$f^{'}(x_{(i+1)})^T d_{(i)} = 0$$

...(using chain rule)

$$-r_{(i+1)}^T d_{(i)} = 0$$

$$d_{(i)}^T A e_{(i)} = 0$$

As we did earlier, we can find the expression for $\alpha_{(i)}$ by assuming $e_{(i+1)}$ and $d_{(i)}$ are conjugate and not orthogonal. We can follow the same steps as in (3.1), now the difference being, A is inserted in between. $\Rightarrow$

$$\alpha_{(i)} = -\frac{d_{(i)}^T A e_{(i)}}{d_{(i)}^T A d_{(i)}} \tag{3.3}$$

$\Rightarrow$

$$\alpha_{(i)} = \frac{d_{(i)}^T r_{(i)}}{d_{(i)}^T A d_{(i)}} \tag{3.4}$$

3.2 Proof that the method actually finds the solution in n steps

Writing the error term as a linear combination of search directions,

$$e_{(0)} = \sum_{j=0}^{j=n-1} \delta_{(j)} d_{(j)} \tag{3.5}$$

To find the values of $\delta_{(j)}$, we again use the fact that the search directions are conjugate or A-orthogonal. Pre-multiplying (3.5) by $d_k^T A$, we get

$$\begin{aligned} d_k^T A e_{(0)} &= \sum_{j=0}^{j=n-1} \delta_{(j)} d_k^T A d_{(j)} \\ &= \delta_{(k)} d_k^T A d_{(k)} ...(by conjugacy of search directions) \end{aligned}$$

Therefore,

$$\begin{aligned} \delta_{(k)} &= \frac{d_k^T A e_{(0)}}{d_k^T A d_{(k)}} \\ &= \frac{d_k^T A (e_{(0)} + \sum_{i=0}^{i=k-1} \alpha_{(i)} d_{(i)})}{d_k^T A d_{(k)}} ...(by conjugacy of search directions) \end{aligned}$$

$\Rightarrow$

$$\delta_{(k)} = \frac{d_k^T A e_{(k)}}{d_k^T A d_{(k)}} \tag{3.6}$$

We can hence see, from (3.3) and (3.6),

$$\alpha_{(i)} = -\delta_{(i)} \tag{3.7}$$

The above equation (3.7) provides us a new interpretation. The process of constructing x term by term can also be seen as a process of cutting down the error term by term.

$$\begin{aligned} e_{(i)} &= e_{(0)} + \sum_{j=0}^{j=i-1} \alpha_{(j)} d_{(j)} \\ &= \sum_{j=0}^{j=n-1} \delta_{(j)} d_{(j)} - \sum_{j=0}^{j=i-1} \delta_{(j)} d_{(j)} ...using (3.7) \\ &= \sum_{j=i}^{j=n-1} \delta_{(j)} d_{(j)} \end{aligned}$$

$\Rightarrow$

$$e_{(i)} = \sum_{j=i}^{j=n-1} \delta_{(j)} d_{(j)} \tag{3.8}$$

Hence, after n iterations, we have, $e_{(n)} = 0$. Therefore, we see that this method does give us the solution in n steps. Hence proved!

3.3 Gram-Schmidt Conjugation

What are the d-vectors? How to find them? Once we know them, our method is set-up. To find these "search vectors", we use a modified version of the popular Gram-Schmidt orthogonalization. We use the method known as the **Gram-Schmidt Conjugation.**

Remember, conjugate vectors are orthogonal to each other in a space that is stretched or scaled by pre-multiplying the A matrix.

Lets say that we have a set of n linearly independent vectors $u_{(0)}, u_{(1)}, ..., u_{(n-1)}$. Our task for this section is to extract the A-orthogonal or conjugate d-vectors out of these u-vectors.

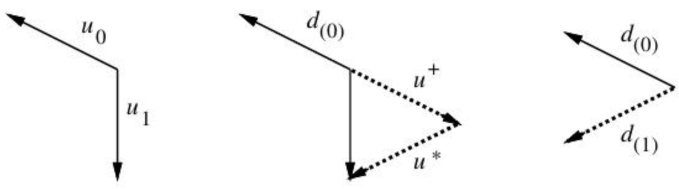

Figure 3.3: Gram-Schmidt Conjugation

set $d_{(0)} = u_{(0)}$

and $\forall 0 < i \leq n-1$ set $d_{(i)} = u_{(i)} + \sum_{k=0}^{i-1} \beta_{ik} d_k$
β_{ik}'s are defined for $i > k$
To find them, we again use the same old property of the d-vectors: conjugacy. As we have done earlier to find the values of δ_j,

$$d_{(i)}^T A d_{(j)} = u_{(i)}^T A d_{(j)} + \sum_{k=0}^{i-1} \beta_{ik} d_k^T A d_j$$
$$0 = u_{(i)}^T A d_{(j)} + \beta_{ij} d_j^T A d_j$$

$\Rightarrow$

$$\beta_{ij} = -\frac{u_{(i)}^T A d_{(j)}}{d_j^T A d_j} \tag{3.9}$$

3.4 Reducing the matrix-vector multiplications in the method of steepest descent

Most of the computational power is consumed in matrix-vector multiplications. We can reduce one matrix-vector multiplication per iteration using the following recurrence relation.

$$r_{(i+1)} = -Ae_{(i+1)} \qquad = -A(e_{(i)} + \alpha_{(i)} d_{(i)})$$

$\Rightarrow$

$$r_{(i+1)} = r_{(i)} - \alpha_{(i)} A d_{(i)} \tag{3.10}$$

3.5 Proof of residual $r_{(i)}$ being perpendicular to the previous search directions

As the d-vectors are constructed using u-vectors, the subspace
$D_i = span\{u_0, u_1, ...u_{i-1}\}$

$$d_{(i)} = u_{(i)} + \sum_{k=0}^{i-1} \beta_{ik} d_k$$

$\Rightarrow$

$$d_{(i)}^T r_{(j)} = u_{(i)}^T r_{(j)} + \sum_{k=0}^{i-1} \beta_{ik} d_k^T r_{(j)} \tag{3.11}$$

$$0 = u_{(i)}^T r_{(j)} ... i < j \tag{3.12}$$

4 The Method of Conjugate Gradients

CG is simply the method of conjugate directions where the search vectors are constructed using residuals.

4.1 Krylov Subspace

As the search vectors are built from residuals,
$span(r_{(0)}, r_{(1)}, ..., r_{(n-1)}) = D_i$

As each $r_{(i+1)}$ is orthogonal to previous search directions, and these directions are built using residuals themselves, each residual is also orthogonal to previous residuals.

$$r^T_{(i+1)} r_{(i)} = 0 ... \forall i \neq j \tag{4.1}$$

Equation (3.10) $\Rightarrow$ Each $r_{(i)}$ is a linear combination of $r_{(i-1)}$ and $Ad_{(i)}$.

Now as $d_{(i-1)} \in D_i$,

This immediately implies that the new subspace D_{i+1} is formed from the union of $D_{(i)}$ and $AD_{(i)}$.

Therefore

$$\begin{aligned} D_i &= span\{d_{(0)}, Ad_{(0)}, A^2 d_{(0)}, ... A^{(i-1)} d_{(0)}\} \\ &= span\{r_{(0)}, Ar_{(0)}, A^2 r_{(0)}, ... A^{(i-1)} r_{(0)}\} \end{aligned}$$

This space is known as **Krylov Space**, formed by using a vector and repeatedly applying a matrix to that vector.

4.2 Deriving the algorithm for CG

From equation (3.9)

$$\beta_{ij} = -\frac{u^T_{(i)} A d_{(j)}}{d^T_j A d_j}$$

We want to simplify this expression.

from equation (3.10)

$$r^T_{(i)} r_{(j+1)} = r^T_{(i)} r_{(j)} - \alpha_{(j)} r^T_{(i)} A d_{(j)}$$

$\Rightarrow$

$$\alpha_{(j)} r^T_{(i)} A d_{(j)} = r^T_{(i)} r_{(j)} - r^T_{(i)} r_{(j+1)}$$

$$r^T_{(i)} A d_{(j)} = \begin{cases} \frac{1}{\alpha_{(i)}} r^T_{(i)} r_{(i)} & \text{if } i = j \\ -\frac{1}{\alpha_{(i-1)}} r^T_{(i)} r_{(i)} & \text{if } i = j+1 \\ 0 & \text{otherwise} \end{cases}$$

therefore

$$\beta_{ij} = \begin{cases} \frac{1}{\alpha_{(i-1)}} \frac{r^T_{(i)} r_{(i)}}{d_{(i-1)^T} A d_{(i-1)}} & \text{if } i = j+1 \\ 0 & \text{otherwise} \end{cases}$$

Lets denote $\beta_{i,i-1} = \beta_i$ We can further simplify above equation

$$\beta_i = \frac{r_{(i)}^T r_{(i)}}{d_{(i-1)^T} r_{(i-1)}} \qquad = \frac{r_{(i)}^T r_{(i)}}{r_{(i-1)^T} r_{(i-1)}}$$

4.3 Putting it all together

$$d_{(0)} = r_{(0)} = b - Ax_{(0)} \tag{4.2}$$

$$\alpha_{(i)} = \frac{r_{(i)}^T r_{(i)}}{d_{(i)}^T A d_{(i)}} \tag{4.3}$$

$$x_{(i+1)} = x_{(i)} + \alpha_{(i)} d_{(i)} \tag{4.4}$$

$$r_{(i+1)} = r_{(i)} - \alpha_{(i)} A d_{(i)} \tag{4.5}$$

$$\beta_{i+1} = \frac{r_{(i+1)}^T r_{(i+1)}}{r_{(i)^T} r_{(i)}} \tag{4.6}$$

$$d_{(i+1)} = r_{(i+1)} + \beta_{i+1} d_{(i)} \tag{4.7}$$

Appendix

Matrix and vector class libraries

Matrix class

```
#ifndef MATRIX_H //Include guard
#define MATRIX_H
#include "vector.h"
template<typename T, int row, int column>
struct matrix
{

T data[row][column];
matrix();

T& operator()(int m,int n) {return data[m][n];}

};

template<typename T, int row, int column>
matrix<T,row,column>::matrix()
{
int i,j;
for(i=0;i<row;i++)
{
for(j=0;j<column;j++)
{
data[i][j]=0;
}
}
}

template<typename T, int row, int column>
std::ostream& operator<<  (std::ostream& os, matrix<T,row,column> &myMatrix)
{
int i,j;
for(i=0;i<row;i++)
{
for(j=0;j<column;j++)
{
std::cout<<'\t';
os<<myMatrix(i,j);
}
std::cout<<'\n';
}

return os;
}

template<typename T, int row, int column>
```

```
std::istream& operator>>  (std::istream& is, matrix<T,row,column> &myMatrix)
{
int i,j;
for(i=0;i<row;i++)
{

for(j=0;j<column;j++)
{
is>>myMatrix(i,j);
}
std::cout<<'\n';
}
return is;
}



template<typename T, int row, int column>
matrix<T,row,column> operator+  ( matrix<T,row,column> &a, matrix<T,row,column> &b)
{
int i,j;
matrix<T,row,column> answer;
for(i=0;i<row;i++)
{
for(j=0;j<column;j++)
{
answer(i,j)=a(i,j)+b(i,j);
}

}
return answer;
}



template<typename T, int row, int column>
matrix<T,row,column> operator-  ( matrix<T,row,column> &a, matrix<T,row,column> &b)
{
int i,j;
matrix<T,row,column> answer;
for(i=0;i<row;i++)
{
for(j=0;j<column;j++)
{
answer(i,j)=a(i,j)-b(i,j);
}

}
return answer;
}




template<typename T, int row1, int column1, int row2, int column2>
matrix<T,row1,column2> operator*  ( matrix<T,row1,column1> &a, matrix<T,row2,column2> &b)
{
int i,j,k,c1,c2,r1,r2;
```

```
matrix<T,row1,column2> answer;

if(column1!=row2)
         {std::cout<<"Matrix multiplication invalid. number of columns of A must be equal to numb

else
{
for(i=0;i<row1;i++)
{
for(k=0;k<column2;k++)
{
answer(i,k)=0;
for(j=0;j<column1;j++)
{
answer(i,k)=answer(i,k)+a(i,j)*b(j,k);
}
}

}
}
return answer;
}

//MATRIX-VECTOR MULTIPLICATION

template<typename T, int row1, int column1, int row2>
vect<T,row2> operator*  ( matrix<T,row1,column1> &a, vect<T,row2> &b)
{
int i,j,k,c1,c2,r1,r2;
vect<T,row2> answer;

if(column1!=row2)
         {std::cout<<"Matrix-vector multiplication invalid. number of columns of A must be equal

else
{
for(i=0;i<row1;i++)
{

answer[i]=0;
for(j=0;j<column1;j++)
{
answer[i]=answer[i]+a(i,j)*b[j];
}


}
}
return answer;
}


//SYMMETRIC MATRIX FLAG
template<typename T, int row, int column>
bool symFlag  ( matrix<T,row,column> &a)
```

```
{
bool flag=1;
int i,j;
for(i=0;i<row;i++)
{
for(j=0;j<column;j++)
{
if(a(i,j)==a(j,i))
{
flag=flag*1;
}
else
{
flag=flag*0;
}
}
}
return flag;
}


//TRANSPOSE OF A MATRIX
template<typename T, int row, int column>
matrix<T,row,column> Transpose (matrix<T,row,column> &a)
{
int i,j;
matrix<T,row,column> answer;
for(i=0;i<row;i++)
{
for(j=0;j<column;j++)
{
answer(i,j)=a(j,i);
}

}
return answer;
}
#endif
```

Vector class

```
#ifndef VECTOR_H //Include guard
#define VECTOR_H
template<typename T, int N>
struct vect
{


T data[100];

vect()
{

int i=0;
// std::cout<<"Enter "<<N<<" numbers: "<<std::endl;
/* for(i=0;i<N;i++)
```

```
{

std::cin>>data[i];
}
*/
}

//double operator[](int j) {return data[j];};
double operator[]( int i) const {return data[i];}
double & operator[]( int i) {return data[i];}
};



template<typename T, int N>
std::ostream& operator<<  (std::ostream& os, vect<T,N> &myVect)
{
for(int i=0; i<N; i++)
{
os<<myVect.data[i];
}
std::cout<<'\n';
return os;

}


template<typename T, int N>
std::istream& operator>>  (std::istream& is, vect<T,N> &myVect)
{
for(int i=0; i<N; i++)
{
is>>myVect.data[i];
}
return is;

}

//*****************************************************************************//
template<int m>
struct metaDot
{
template<typename T, int N>
static T f(vect<T,N>& a, vect<T,N>& b)
{
return a[m]*b[m] + metaDot<m-1>::f(a,b);
}
};

template<> // the end of the recursion
struct metaDot<0>
{
template<typename T, int N>
static T f(vect<T,N>& a, vect<T,N>& b)
{
```

```
return a[0]*b[0];
}
};


// The dot() function invokes metaDot
template<typename T, int N>
inline T dot(vect<T,N>& a, vect<T,N>& b)
{
return metaDot<N-1>::f(a,b);
}

//******************************************************************************//

template<typename T, int N>
vect<T,N> operator+(vect<T,N> &a, vect<T,N> &b)
{
vect<T,N> answer;
int i;
for(i=0;i<N;i++)
{
answer[i]=a[i]+b[i];
}
return answer;
}

template<typename T, int N>
vect<T,N> operator-(vect<T,N> &a, vect<T,N> &b)
{
vect<T,N> answer;
int i;
for(i=0;i<N;i++)
{
answer[i]=a[i]-b[i];
}
return answer;
}



#endif
```

Code for the method of steepest descent

Pseudo-code

Code

```
#include<iostream>
#include "matrix.h"
#include "vector.h"

int main()
{
int r1,c1;
int i,j,k,l,iter,niter;
```

```
const int n=2;

matrix<double, n,n> A, ATranspose;
vect<double,n> b, bTranspose, x, xold, xnew,xGuess;

std::cout<<"Enter elements of matrix A (lhs of Ax=b)(row-wise)"<<n<<"x"<<n<<std::endl;
std::cin>>A;

std::cout<<"Enter elements of vector b (rhs of Ax=b) (row-wise)"<<n<<"x"<<1<<std::endl;
std::cin>>b;

std::cout<<"Enter elements of the guess vector (row-wise)"<<n<<"x"<<1<<std::endl;
std::cin>>xGuess;
x=xGuess;

/************************************************************************************************

double alpha, beta;

vect<double,n> residue, d, error;

double errorMax;

vect<double,n> temp1,temp2,temp6, temp7, temp9;

matrix<double,n,n> temp5, temp8;
double deltaOld, deltaNew, delta0,temp3, temp4 ;

double epsilon= 1e-4;
double eps = epsilon*epsilon;
niter=300; //max no. of iterations
iter=0;


//********************calculate residue with guess vector*******************************//
temp1=A*x;
residue=b-temp1;
//***initialize the search direction d with the residue d0=r0=b-A*x0****//
d = residue;

//calculate alpha -> alpha_(i)= r_(i)^T* r(i)/ d_(i)^T*A*d_(i)
deltaNew=dot(residue,residue);  // this is r_(i)^T* r_(i)-> numerator

delta0=deltaNew;


while(iter<niter && deltaNew>eps*delta0)
{
temp2= A*residue;


temp4=dot(residue,temp2);
// std::cout<<"temp4="<<temp4<<std::endl;
alpha =deltaNew/temp4;
// std::cout<<"alpha="<<alpha<<std::endl;

//Alpha[iter]=alpha;

for(i=0;i<n;i++)
{
temp5(i,i)=alpha;
}
// std::cout<<"temp5="<<temp5<<std::endl;
temp6=temp5*residue;
// std::cout<<"temp6="<<temp6<<std::endl;

xnew= xold+temp6;
/*********************************************************/

//if(iter%50==0)
{
// residue= b-temp2;
}

//else
{
temp7=temp5*temp2; // temp7=(alpha*I)*(A*d)
// std::cout<<"temp7="<<temp7<<std::endl;
residue=residue-temp7;
}

// std::cout<<"residue="<<residue<<std::endl;
/**********************************************************/
deltaNew=dot(residue,residue);
iter =iter+1;
}
std::cout<<"no. of iterations="<<iter<<std::endl;
std::cout<<"ans ="<<x<<std::endl;

return 0;
}
```

Code for CG

Pseudo-code

Code

```
#include<iostream>
#include "matrix.h"
#include "vector.h"

int main()
{
int r1,c1;
int i,j,k,l,iter,niter;

const int n=2;
```

```
matrix<double, n,n> A, ATranspose;
vect<double,n> b, bTranspose, x, xold, xnew,xGuess;

std::cout<<"Enter elements of matrix A (lhs of Ax=b)(row-wise)"<<n<<"x"<<n<<std::endl;
std::cin>>A;

std::cout<<"Enter elements of vector b (rhs of Ax=b) (row-wise)"<<n<<"x"<<1<<std::endl;
std::cin>>b;

std::cout<<"Enter elements of the guess vector (row-wise)"<<n<<"x"<<1<<std::endl;
std::cin>>xGuess;
x=xGuess;

/**************************************************************************************************

double alpha, beta;

vect<double,n> residue, d, error;

double errorMax;

vect<double,n> temp1,temp2,temp6, temp7, temp9;

matrix<double,n,n> temp5, temp8;
double deltaOld, deltaNew, delta0,temp3, temp4 ;

double epsilon= 1e-4;
double eps = epsilon*epsilon;
niter=300; //max no. of iterations
iter=0;


//********************calculate residue with guess vector******************************//
temp1=A*x;
residue=b-temp1;
//***initialize the search direction d with the residue d0=r0=b-A*x0****//
d = residue;

//calculate alpha -> alpha_(i)= r_(i)^T* r(i)/ d_(i)^T*A*d_(i)
deltaNew=dot(residue,residue);  // this is r_(i)^T* r_(i)-> numerator

delta0=deltaNew;

while(iter<niter && deltaNew>eps*delta0)
{
/*******************calculating alpha***********************/
temp2= A*d; //temp2 = q in the given algorithm
//std::cout<<"temp2="<<temp2<<std::endl;

temp4=dot(d,temp2);  //this calculates the denominator of expression for alpha
//std::cout<<"temp4="<<temp4<<std::endl;

alpha =deltaNew/temp4;
//std::cout<<"alpha="<<alpha<<std::endl;
```

```
/***********************************************************/

for(i=0;i<n;i++)
{
temp5(i,i)=alpha;
}

temp6=temp5*d;
// std::cout<<"temp6="<<temp6<<std::endl;
//update x-> xnew =xold+alpha*d//
x = x + temp6;
// std::cout<<"x="<<x<<std::endl;
//**********************************************************/

//if(iter%50==0)
{
// residue= b-temp2;
}

//else
{
temp7=temp5*temp2; // temp7=(alpha*I)*(A*d)
// std::cout<<"temp7="<<temp7<<std::endl;
residue=residue-temp7;
}

// std::cout<<"residue="<<residue<<std::endl;
//*******************calculating beta***********************/

deltaOld=deltaNew;
deltaNew=dot(residue,residue);
beta= deltaNew/deltaOld;
// std::cout<<"beta="<<beta<<std::endl;
//**********************************************************/

for(i=0;i<n;i++)
{
temp8(i,i)=beta;
}

temp9= temp8*d;  //temp9=(beta*I)*d;
//std::cout<<"temp9="<<temp9<<std::endl;
//**************update d (search direction)****************/
d= residue+temp9;
// std::cout<<"d="<<d<<std::endl;
//**********************************************************/
// std::cout<<"iter="<<iter<<std::endl;

// if(iter<niter && deltaNew>epsilon*epsilon*delta0)
{
iter =iter+1;
}
// else
// {break;}
std::cout<<"deltaNew="<<deltaNew<<std::endl;
```

```
std::cout<<"delta0="<<delta0<<std::endl;
double ratio = deltaNew/delta0;
std::cout<<"deltaNew/delta0"<<ratio<<std::endl;
}
std::cout<<"no. of iterations="<<iter<<std::endl;
std::cout<<"ans ="<<x<<std::endl;
return 0;
}
```

Code for the Preconditioned CG: Using Jacobi Preconditioning

Pseudo-code

Code

```
#include<iostream>
#include "matrix.h"
#include "vector.h"

int main()
{
int r1,c1;
int i,j,k,l,iter,niter;

const int n=2;

matrix<double, n,n> A, ATranspose,M,MInverse;
vect<double,n> b, bTranspose, x, xold, xnew,xGuess,s;

std::cout<<"Enter elements of matrix A (lhs of Ax=b)(row-wise)"<<n<<"x"<<n<<std::endl;
std::cin>>A;

std::cout<<"Enter elements of vector b (rhs of Ax=b) (row-wise)"<<n<<"x"<<1<<std::endl;
std::cin>>b;

std::cout<<"Enter elements of the guess vector (row-wise)"<<n<<"x"<<1<<std::endl;
std::cin>>xGuess;
x=xGuess;

/***********************************************************************************************
/**************************Jacobi Preconditioner, extracting the diagonal elements of A**********
for(i=0;i<n;i++)
{
for(j=0;j<n;j++)
{
if(i==j)
{
M(i,j)=A(i,j);
}

}

}

for(i=0;i<n;i++)
```

```
{
for(j=0;j<n;j++)
{
if(i==j)
{
MInverse(i,j)=1/A(i,j);
}

}

}
/*********************************************************************************************

double alpha, beta;

vect<double,n> residue, d, error;

double errorMax;

vect<double,n> temp1,temp2,temp6, temp7, temp9;

matrix<double,n,n> temp5, temp8;
double deltaOld, deltaNew, delta0,temp3, temp4 ;

double epsilon= 1e-4;
double eps = epsilon*epsilon;
niter=300; //max no. of iterations
iter=0;


//********************calculate residue with guess vector******************************//
temp1=A*x;
residue=b-temp1;
//***initialize the search direction d with the residue d0=r0=b-A*x0****//
d = MInverse*residue;

//calculate alpha -> alpha_(i)= r_(i)^T* r(i)/ d_(i)^T*A*d_(i)
deltaNew=dot(residue,d);  // this is r_(i)^T* r_(i)-> numerator

delta0=deltaNew;

while(iter<niter && deltaNew>eps*delta0)
{
/*******************calculating alpha***********************/
temp2= A*d; //temp2 = q in the given algorithm
//std::cout<<"temp2="<<temp2<<std::endl;

temp4=dot(d,temp2);  //this calculates the denominator of expression for alpha
//std::cout<<"temp4="<<temp4<<std::endl;

alpha =deltaNew/temp4;
//std::cout<<"alpha="<<alpha<<std::endl;
/*******************************************************/

for(i=0;i<n;i++)
```

```
{
temp5(i,i)=alpha;
}

temp6=temp5*d;
// std::cout<<"temp6="<<temp6<<std::endl;
//update x-> xnew =xold+alpha*d//
x = x + temp6;
// std::cout<<"x="<<x<<std::endl;
//**************************************************************/

//if(iter%50==0)
{
// residue= b-temp2;
}

//else
{
temp7=temp5*temp2; // temp7=(alpha*I)*(A*d)
// std::cout<<"temp7="<<temp7<<std::endl;
residue=residue-temp7;
}

// std::cout<<"residue="<<residue<<std::endl;
//********************calculating beta************************/
s=MInverse*residue;
deltaOld=deltaNew;
deltaNew=dot(residue,s);
beta= deltaNew/deltaOld;
// std::cout<<"beta="<<beta<<std::endl;
//**************************************************************/

for(i=0;i<n;i++)
{
temp8(i,i)=beta;
}

temp9= temp8*d;  //temp9=(beta*I)*d;
//std::cout<<"temp9="<<temp9<<std::endl;
//***************update d (search direction)*****************/
d= s+temp9;
// std::cout<<"d="<<d<<std::endl;
//**************************************************************/
// std::cout<<"iter="<<iter<<std::endl;

// if(iter<niter && deltaNew>epsilon*epsilon*delta0)
{
iter =iter+1;
}
// else
// {break;}
// std::cout<<"deltaNew="<<deltaNew<<std::endl;
// std::cout<<"delta0="<<delta0<<std::endl;
double ratio = deltaNew/delta0;
std::cout<<"deltaNew/delta0"<<ratio<<std::endl;
```

```
}
std::cout<<"no. of iterations="<<iter<<std::endl;
std::cout<<"ans ="<<x<<std::endl;
return 0;
}
```

```
$i \Leftarrow 0$
$r \Leftarrow b - Ax$
$\delta \Leftarrow r^T r$
$\delta_0 \Leftarrow \delta$
While $i < i_{max}$ and $\delta > \varepsilon^2 \delta_0$ do
    $q \Leftarrow Ar$
    $\alpha \Leftarrow \frac{\delta}{r^T q}$
    $x \Leftarrow x + \alpha r$
    If $i$ is divisible by 50
        $r \Leftarrow b - Ax$
    else
        $r \Leftarrow r - \alpha q$
    $\delta \Leftarrow r^T r$
    $i \Leftarrow i + 1$
```

Figure 4.1: Pseudo-code for the method of steepest descent

```
$i \Leftarrow 0$
$r \Leftarrow b - Ax$
$d \Leftarrow r$
$\delta_{new} \Leftarrow r^T r$
$\delta_0 \Leftarrow \delta_{new}$
While $i < i_{max}$ and $\delta_{new} > \varepsilon^2 \delta_0$ do
    $q \Leftarrow Ad$
    $\alpha \Leftarrow \frac{\delta_{new}}{d^T q}$
    $x \Leftarrow x + \alpha d$
    If $i$ is divisible by 50
        $r \Leftarrow b - Ax$
    else
        $r \Leftarrow r - \alpha q$
    $\delta_{old} \Leftarrow \delta_{new}$
    $\delta_{new} \Leftarrow r^T r$
    $\beta \Leftarrow \frac{\delta_{new}}{\delta_{old}}$
    $d \Leftarrow r + \beta d$
    $i \Leftarrow i + 1$
```

Figure 4.2: Pseudo-code for CG

```
$i \Leftarrow 0$
$r \Leftarrow b - Ax$
$d \Leftarrow M^{-1}r$
$\delta_{new} \Leftarrow r^T d$
$\delta_0 \Leftarrow \delta_{new}$
While $i < i_{max}$ and $\delta_{new} > \varepsilon^2 \delta_0$ do
    $q \Leftarrow Ad$
    $\alpha \Leftarrow \frac{\delta_{new}}{d^T q}$
    $x \Leftarrow x + \alpha d$
    If $i$ is divisible by 50
        $r \Leftarrow b - Ax$
    else
        $r \Leftarrow r - \alpha q$
    $s \Leftarrow M^{-1}r$
    $\delta_{old} \Leftarrow \delta_{new}$
    $\delta_{new} \Leftarrow r^T s$
    $\beta \Leftarrow \frac{\delta_{new}}{\delta_{old}}$
    $d \Leftarrow s + \beta d$
    $i \Leftarrow i + 1$
```

Figure 4.3: Pseudo-code for Preconditioned CG