分类号 ______________ 密级 ______________

UDC ______________ 编号 ______________

中国科学院数学与系统科学研究院
博士后研究工作报告

无约束优化的子空间信赖域方法

王 周 宏

工作完成日期： 2004 年 12 月

报告提交日期： 2004 年 12 月

中国科学院数学与系统科学研究院

2004 年 12 月

北京

中文题目 无约束优化的子空间信赖域方法

英文题目 The Subspace Trust Region Methods for Unconstrained Optimization

博　士　后　姓　名： 王 周 宏

流动站（一级学科）名称： 数　　学

专　业（二级学科）名称： 计算数学

合　作　导　师： 袁 亚 湘 研究员

研究工作起始时间： 2002 年 12 月 30 日

研究工作期满时间： 2004 年 12 月 30 日

提交工作报告日期： 2004 年 12 月 30 日

摘　要

随着社会的进步和技术手段的发展，在优化实践中需要考虑的因素越来越多，所形成的优化模型往往是大规模的，如反问题的求解和蛋白质折叠问题的计算。由于计算机硬件的限制，这类问题的求解通常很困难，一般需要采用高性能计算，同时对优化算法的结构和收敛速度也提出了更高的要求。

本文针对大规模无约束优化问题系统地研究了子空间信赖域方法。首先在第一章介绍了用于测试非线性优化算法的 CUTEr 环境。然后在第二章针对大规模无约束优化问题，通过分析传统拟牛顿信赖域方法的子空间性质，提出了一个子空间信赖域方法。该方法在求解一个大规模的 n 维无约束优化问题时，在第 $k(k < n)$ 次迭代只需计算一个 k 维的二次规划信赖域子问题以获得一个信赖域步。当 $k \ll n$ 时，便可大大地减少计算量，同时仍保持传统信赖域方法的超线性收敛性。通过对 CUTEr 测试环境中大量大规模无约束优化问题的计算表明，与传统信赖域方法相比，子空间信赖域方法极大地减少了求解问题所需的 CPU 时间。

本文第三章在分析有限内存 BFGS 方法的子空间性质的基础上，提出了两个使用子空间技术的有限内存信赖域方法，并针对一些典型的大规模无约束优化问题与有限内存 BFGS 方法进行了比较。目前的计算结果表明，从函数及其梯度计算次数的角度上看，与采用线搜索的有限内存 BFGS 方法相比，有限内存子空间信赖域方法具有相当的竞争力。

本文最后在第四章总结了子空间信赖域方法的特点及其局限性，提出了一些值得进一步研究的问题。

关键词:　大规模优化，无约束优化，拟牛顿方法，信赖域方法，子空间方法.

Abstract

Many large scale nonlinear optimization problems arise in scientific and engineering computations, for example, the computations of inverse problems and the protein folding problems. Usually this class of problems is very difficult to solve, and we have to compute them on high performance computers.

There are four parts in this report. Chapter 1 concerns with the CUTEr environment which is designed for testing nonlinear optimization algorithms. In Chapter 2 we studies subspace properties of trust region methods for unconstrained optimization, assuming the approximate Hessian is updated by quasi-Newton formulae. The trial step can a be defined by minimizing the quasi-Newton quadratic model in the subspace which is spanned by all the gradient vectors computed. When solving a problem whose number of variables is n, we only need to solve a quadratic subproblem whose dimension is $k(k < n)$ at k-th iteration. If $k \ll n$, the amount of computation are reduced significantly, and the suplinear convergence property for the conventional trust region method is still kept. Numerical results for a lot of large scale problems in CUTEr are reported, where we can see that the subspace trust region methods take a lot of advantages over the traditional trust region methods.

In Chapter 3 two limited memory trust region algorithms for large scale unconstrained optimization are constructed, which is based on the analysis of the subspace properties of the limited memory BFGS method. Numerical results on some typical large scale test problems and their comparison with the limited memory BFGS

method are reported. When the number of function and gradient evaluations is used as the performance metric, we can see that the new algorithms are promising compared with the limited memory BFGS method.

Finally, conclusions are drawn in the last Chapter. The properties and shortcomings of the subspace trust region methods are summarized, and some problems worth further researching are proposed.

Keywords: Large scale optimization, Unconstrained optimization, Quasi-Newton methods, Trust region methods, Subspace technique.

目 录

第一章 The Testing Environment CUTEr

CUTEr is an testing environment for constrained and unconstrained nonlinear optimization algorithms. It includes small and very large problems and is carefully managed. This chapter is mainly concerned about the designing principle of CUTEr in some depth, the SIF format and its decoder, and how to program and test optimization algorithms by using C and FORTRAN language together in CUTEr. The main references for this chapter are [22], [1], [19] and [9].

§1.1 Introduction

First let's see some acronyms,

CUTE—Constrained and Unconstrained Testing Environment for optimization software.

CUTEr—CUTE revised

SIF—Standard Input Format

MPS—Mathematical Programming System, a standard input format for LP.

LANCELOT—Large and Nonlinear Extended Lagrangian Optimization Technique

The test problems collected in CUTEr span a wide spectrum of difficulty, ranging from small-scale differentiable unconstrained minimization to large-scale equality and inequality-constrained dense and sparse problems, systems of nonlinear equations, network problems etc. The set covers the Argonne Test Set, the Hock and Schittkowski (1981) collection, the Dembo network problems, the Gould QPs, etc. However, all the test problems collected are written in SIF (a format to input partially grouped problems, which was motivated by the success of MPS, and preserves MPS compatibility), which is not familiar with us.

The usual method to define problems, for example in Fortran language,

```
subroutine objfcn(n, x, f, nprob)
dimension x(n)
```

```
      real yy(20)
c     Rosenbrock Function
      if(nprob .eq. 1) then
        f1=10*(x(2)-x(1)*x(1))
        f2=1-x(1)
        f=f1*f1+f2*f2
     ......
    subroutine grdfcn(n, x, g, nprob)
    dimension x(n),  g(n)
    .....
    subroutine HessFcn(n, x, H, nprob)
    dimension x(n),  H(n, n)
    ....
```

or in C language:

```
void fnt(int n,  double *x, double *f, double *g)
{
   *f=9-8*x[0]-6*x[1]-4*x[2]+2*(x[0]*x[0]+x[1]*x[1])+...;
   g[0]=x[0]+x[1]+x[2]*2-3;
   g[1]=-x[0];g[2]=-x[1];g[3]=-x[2];
   ...
}

void grdfnt(int n,  double *x,  double **gfg)
{ ......}
```

We will call these subroutines (or functions in C language) in our own optimization algorithm software to evaluate the value of the object function, and the constraints, and the gradients and even the Hessians of them.

The difficulties for preliminary users in CUTEr is that the test problems are all written in SIF format. You have to use the SIF decoder "SifDec" to decode SIF files properly in order to generate the necessary Fortran 77 subroutines, which will be used by other packages. But the subroutines are not written in the usual way as above, and the data structure is very complex in accounting for the structure called *group partial separability.*

§1.2 Group Partial Separability and the SIF Format

Structure is an integral and significant aspect of large-scale problems. Structure is often equated with sparsity; indeed the two are closely linked when the problem is linear. However, sparsity is not the most important phenomenon associated with a nonlinear function; that role is played by invariant subspaces. The *invariant subspace* of a function $f(x)$ is the set of all vectors w for which $f(x+w) = f(x)$ for all possible vectors x. For example the function :

$$f(x_1, x_2, \cdots, x_{1000}) = (x_1 + \cdots + x_{1000})^2$$

has a completely dense Hessian matrix but still has an invariant subspace of dimension 999, the set of all vectors orthogonal to a vector of ones, we also could see it by a linear transform

$$y_1 = x_1 + \cdots + x_{1000},\ y_i = x_i \text{ for } 2 \leq i \leq 1000$$

and the function becomes

$$f(x_1, x_2, \cdots, x_{1000}) = y_1^2$$

The importance of invariant subspaces is that nonlinear information is not required for a function in this subspace. We are particularly interested in functions which have large (as a percentage of the overall number of variables) invariant subspaces. This allows for efficient storage and calculation of derivative information. The penalty is, of course, the need to provide information about the subspace to an optimization procedure.

A particular objective function $F(x)$ is unlikely to have a large invariant subspace itself. However, many reasonably behaved functions may be expressed as a sum of *element* functions, each of which does have a large invariant subspace. Such a function is said to be *group partially separable*, which can be represented in the following formulae.

Considering the problem of minimizing or maximizing an objective function of the form

$$F(x) = \sum_{i \in I_O} g_i \left(\sum_{j \in J_i} w_{i,j} f_j(\bar{x}_j) + a_i^T x - b_i \right) + \frac{1}{2} \sum_{j=1}^{n} \sum_{k=1}^{n} h_{j,k} x_j x_k \tag{1.2.1}$$

where $x = (x_1, x_2, \cdots, x_n)$ within the "box" region

$$l_i \le x_i \le u_i, \quad l \le i \le n \tag{1.2.2}$$

(where either bound on each variable may be infinite), and where the variables are required to satisfy the extra conditions

$$g_i \left(\sum_{j \in J_i} w_{i,j} f_j(\bar{x}_j) + a_i^T x - b_i \right) = 0 \ (i \in I_E) \tag{1.2.3}$$

$$0 \left\{ \begin{array}{c} \le \\ \ge \end{array} \right\} g_i \left(\sum_{j \in J_i} w_{i,j} f_j(\bar{x}_j) + a_i^T x - b_i \right) \left\{ \begin{array}{c} \le \\ \ge \end{array} \right\} r_i, \ (i \in I_I) \tag{1.2.4}$$

for some index sets I_0, I_E and I_I and (possibly infinite) values r_i.

The univariate functions g_i are known as ***group functions***. The argument

$$\sum_{j \in J_i} w_{i,j} f_j(\bar{x}_j) + a_i^T x - b_i$$

is known as the ***i-th group***. The functions $f_j, j = 1, \cdots, n_e$, are called ***non-linear element functions***. They are functions of the problem variables $\bar{x}_j$ are either small subsets of $\bar{x}$ or such that f_j has a large invariant subspace for some other reason. The constants $w_{i,j}$ are known as *weights*, while the function $a_i^T x - b_i$ is known as the ***linear element*** for the i-th group. The additional term $\frac{1}{2} \sum_{j=1}^{n} \sum_{k=1}^{n} h_{j,k} x_j x_k$ in the objective function is the ***quadratic objective group***; the leading $\frac{1}{2}$ is there by convention.

It is more common to call the group functions in (1.2.3) equality constraint functions, those in (1.2.4) inequality constraint functions and the sum of those in (1.2.1) the objective function.

When stating a structured nonlinear optimization problem of the form (1.2.1)–(1.2.4), we need to specify the group functions, linear and nonlinear elements, and corresponding gradients and Hessians, and the way that they all fit together. In this way we can explore sparsity of the structured problems to save memory storage as much as possible and to input the gradients and Hessians as easy as possible. So A. R. Conn, N. I. M. Gould, Ph. Toint, etc. developed the SIF (Standard Input Language) input language in accounting for the problem structure.

§1.2.1 The Elemental and Internal Variables.

A nonlinear element function f_j is assumed to be a function of the problem variables $\bar{x}_j$, a subset of the overall variables $\bar{x}$, which has a small invariant subspace. Suppose that $\bar{x}_j$ has n_j components ($n_j \ll n$ in general). Then one can consider the nonlinear element function to be of the structural form $f_j(v_1, \cdots, v_{n_j})$, where we assign $v_1 = \bar{x}_{j_1}, \cdots, v_{n_j} = \bar{x}_{j_{n_j}}$. The *elemental variables* for the element function f_j are the variables v. It is the elemental variables which are important in defining the character of the nonlinear element functions.

As an example, the first nonlinear element function for a particular problem might be

$$(x_{29} + x_3 - 2x_{17})e^{x_{29}-x_{17}} \tag{1.2.5}$$

which has the structural form

$$f_1(v_1, v_2, v_3) = (v_1 + v_2 - 2v_3)e^{v_1-v_3}, \tag{1.2.6}$$

where we need to assign $v_1 = x_{29}, v_2 = x_3, v_3 = x_{17}$. For this example, there are three elemental variables.

Although f_1 is a function of three variables, the function itself is really only composed of *two* independent parts; the product of $v_1 + v_2 - 2v_3$ with $e^{v_1-v_3}$ or, if we write $u_1 = v_1 + v_2 - 2v_3$ and $u_2 = v_1 - v_3$, the product of u_1 with e^{u_2}. The variables u_1 and u_2 are called *internal variables* for the element function. They are obtained as *linear combinations* of the elemental variables. So the dimension of the invariant subspace for the element is $n - 2$, but 3 variables. *The important* *feature as far as an optimization procedure is concerned is that each nonlinear element function involves as few internal variables as possible, as this allows for compact storage and more efficient derivative approximation.*

In general, there will be a linear transformation from the elemental variables to the internal ones. For example in (1.2.6), we have

$$\begin{pmatrix} u_1 \\ u_2 \end{pmatrix} = \begin{pmatrix} 1 & 1 & -2 \\ 1 & 0 & -1 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} \tag{1.2.7}$$

In general the transformation will be of the form

$$u = Wv \tag{1.2.8}$$

and this transformation is *useful* if the matrix W has fewer rows than columns.

§1.2.2 Element and Group Types.

It is quite common for large nonlinear programming problems to be defined in terms of many nonlinear elements. It is also common that these elements, although using different problem variables, are structurally the same as each other. For example, the function

$$\sum_{i=1}^{n-1} (x_i x_{i+1} + x_1/x_i)^i \tag{1.2.9}$$

naturally decomposes into the sum of $n-1$ group functions. Each group is the sum of two nonlinear elements $v_1 v_2$ (where $v_1 = x_i$ and $v_2 = x_{i+1}$) and v_1/v_2 (where $v_1 = x_1$ and $v_2 = x_i$).

The same principal may be applied to group functions. For example, the group functions for

$$\sum_{i=1}^{n-1} c_i (x_i x_{i+1})^2 \tag{1.2.10}$$

have different arguments and depend upon different values of a parameter but are still structurally all the same, each being of the form

$$g(\alpha) = p_1 \alpha^2.$$

Any two group functions which are structurally the same are said to be of the same type; the structural function is known as the group type and its argument is the group-type variable. Once again, using group types makes the task of specifying the characteristics of individual group functions more straightforward. The group type

$$g(\alpha) = \alpha$$

is known as the trivial type. Trivial groups occur very frequently and are considered to be the default type. It is then only necessary to specify non-trivial group types.

After specifying the group functions, linear and nonlinear elements, then we should define the corresponding gradients and Hessians, and the way that they all fit together. These can be done by the SIF format. Let's see an example, which is the famous Rosenbrock function,

$$\begin{array}{cccc} \min & 100(x_2 - x_1^2)^2 & + & (x_1 - 1)^2 \\ & \uparrow & & \uparrow \\ & \text{group 1} & & \text{group 2} \end{array}$$

The function has two groups, of the same group type $g(\alpha) = \alpha^2$. The weight for the first group is 100, and it has a nonlinear element x_1^2, whose type is v_1^2, with only one elemental variable v_1, but no internal variables. The corresponding SIF file (see following) contains 3 parts—SDIF, SEIF, SGIF. SDIF defines the variables, objects and constraints' names, groups' names used in each object and constraint, linear part in the groups, elements used in each group and the corresponding weights. SEIF and SGIF define each element type and group type.

```
***************************
* SET UP THE INITIAL DATA *
***************************
NAME          ROSENBR
*   Problem :
*   *********
*   The ever famous 2 variables Rosenbrock "banana valley" problem
*   Source:  problem 1 in
```

```
*   J.J. More',  B.S. Garbow and K.E. Hillstrom,
*   "Testing Unconstrained Optimization Software",
*   ACM Transactions on Mathematical Software,  vol. 7(1),  pp. 17-41,  1981.

*   SIF input: Ph. Toint,  Dec 1989.

*   classification SUR2-AN-2-0

VARIABLES   $$ defines variables

    X1
    X2

GROUPS

 N  G1        X2        1.0
 N  G1        'SCALE'   0.01
 N  G2        X1        1.0

CONSTANTS

    ROSENBR   G2        1.0

BOUNDS

 FR ROSENBR   'DEFAULT'

START POINT

    ROSENBR   X1        -1.2
    ROSENBR   X2         1.0

ELEMENT TYPE
```

```
 EV SQ        V1

ELEMENT USES

 T  E1        SQ
 V  E1        V1                       X1

GROUP TYPE

 GV L2        GVAR

GROUP USES

 T  'DEFAULT' L2
 XE G1        E1          -1.0

OBJECT BOUND

 LO ROSENBR               0.0

*   Solution

*LO SOLTN                 0.0

ENDATA

***********************
* SET UP THE FUNCTION *
* AND RANGE ROUTINES  *
***********************

ELEMENTS      ROSENBR

INDIVIDUALS
```

```
 T  SQ
 F                    V1 * V1
 G  V1                V1 + V1
 H  V1        V1      2.0

ENDATA

*********************
* SET UP THE GROUPS *
* ROUTINE           *
*********************

GROUPS        ROSENBR

INDIVIDUALS

 T  L2
 F                    GVAR * GVAR
 G                    GVAR + GVAR
 H                    2.0

ENDATA
```

Lines begin with $*$ are treated as comments and the character $ identifies the remainder of the line as a comment. The comments is ignored when interpreting the input file. The file must always start with a **NAME** card, on which a name (in this case ROSENBR) for the example may be given, and must end with an **ENDATA** card. Next name the problem variables. The variables are named X1 and X2. Then name the groups used. The character N in field 1 specifies that this is an objective function group, and the objective function uses two groups, named as G1 and G2, and the group G1 should be divided by 0.01, and contains a linear element $1.0 * X2$, the group G2 contains a linear element $1.0 * X1$. The **CONSTANTS**, **BOUNDS** and **START POINT** sections tell us that the group G2 contains a constant 1.0, so the linear part in group G2 is $1.0 * X1 - 1.0$, and all variables are unbounded (the default is being changed as FR—free), and the start point is (-1.2, 1.0). Next define nonlinear information. The **ELEMENT TYPE** section defines only one nonlinear element type, named as SQ, and has only one elemental variable named as V1. The **ELEMENT USES** section tells us that a nonlinear element E1's type is SQ with $V1 = X1$, so we know that $E1 = SQ(X1)$. In **GROUP TYPE**, we define a group function whose name is L2 and its variable is GVAR (group function is univariate). The **GROUP USES** section tell us that all groups' types in the objective function (indicated by the character T in field 1) are L2, and the group G1 contains a nonlinear element whose name is E1, and its coefficient is -1.0. By now we know that

$$G1 = L2(1.0 * X2 - 1.0 * E1) = L2(X2 - SQ(X1)).$$

The **ENDATA** card tells us that the SDIF part completed. In the following two parts, SEIF and SGIF define the element type SQ and group type L2, and the corresponding derivatives. After fitting them together, we could get the entire function's value and its derivatives.

The line begins with "* classification" in the SIF file above defines the problem's classification. A problem is classified by the string

$$\begin{array}{cccccccccc} X & X & X & r & - & X & X & - & n & - & m \\ \uparrow & \uparrow & \uparrow & \uparrow & & \uparrow & \uparrow & & \uparrow & & \uparrow \\ obj & cons. & smooth & deri. & & origin & internal. & & var.num. & & cons.num. \end{array}$$

The first character defines the type of the objective function,

N—no object

C—the object is constant

L—the object is linear

Q—quadratic

S—sum of squares

O—others

The 2nd character defines the type of the constraints,

U—unconstrained

X—fixed
B—bounded
N—network (the adjacency matrix)
L—linear
Q—quadratic
O—others

The 3rd character indicates the smoothness of the problem,

R—regular, its 1st and 2nd derivatives exist and continuous
I—irregular

The 4th character is an integer, can be 0, 1 or 2, indicates the degree of the highest derivatives provided analytically.

The 5th character immediately following the first hyphen indicates the primary origin of the problem,

A—Academic, constructed by reseachers
M—Modelling excises
R—Real application

The next character indicates whether the problem contains explicit internal variables,

Y—yes
N—no

The last two characters indicate the number of variables and constraints in the problem.

V—the number can be chosen by user.
n or m—a nonnegative integer

For example, the Rosenbrock function's classification is: SUR2-AN-2-0. SifDec also provides a script **select** to help you choose what kind of problems you want to solve.

§1.2.3 The SIF Decoder and the Evaluation Tools

The SIF file is complex and it is diffcult to compute the values of object and gradient directly from a SIF file. The SIF decoder reads the statements found in the SIF file, and if the SIF file is correct, interprets it to produce a few Fortran subroutine files and a data file:

GROUP.f contains a Fortran subroutine that evaluates the numerical functions corresponding to the nontrivial group functions, *as well as their gradients and Hessians*

ELFUN.f contains a Fortran subroutine that evaluates the numerical functions corresponding to the nonlinear element functions, *as well as their gradients and Hessians*

EXTER.f (if present) contains all the user supplied Fortran functions found at the end of the problem SIF file.

RANGE.f contains a Fortran subroutine that computes the linear transformation from elemental to internal variables for all elemental types.

OUTSDIF.d contains data on the problem structure, such as variable and constraint names, number of variables, elements, groups, the starting point and the like.

In CUTEr, the SIF decoder will also produce a file named ***AUTOMAT.d***, which record the type of automatic differentiation used in the decoding.

After decoding the SIF file, we have to fit the subroutines produced by the SIF decoder together to evaluate the numerical function values for the entire functions (objects and constraints) and their derivatives. But this is difficult task and the data structure in the subroutines produced by SIF decoder are very complicated. So CUTEr provides some additional tools to do the work. These tools are in the following two tables. These tools are defined in files located in the folder common/src/tools, and we can call them in our codes to evaluate the desired function values and their derivatives.

§1.3 Installation, Update, Uninstalling and Resizing

As we have seen, CUTEr includes three parts: the SIF problems, the SIF decoder and the evaluation tools. We can download them from http://hsl.rl.ac.uk/cuter-www/index.html. At the present, you have to download them separately,

表 1.1: Tools for Unconstrained Problems

Tool name	Brief description
ubandh	extract a banded matrix out of the Hessian matrix
udh	evaluate the Hessian matrix
udimen	get the number of variables involved
udimse	determine the number of nonzeros required to store the sparse Hessian matrix in finite element format
udimsh	same as udimse, in coordinate format
ueh	evaluate the sparse Hessian matrix in finite element format
ufn	evaluate function value
ugr	evaluate gradient
ugrdh	evaluate the gradient and Hessian matrix
ugreh	evaluate the gradient and Hessian matrix in finite element format
ugrsh	evaluate the gradient and Hessian matrix in coordinate format
unames	obtain the names of the problem and its variables
uofg	evaluate function value and possibly gradient
uprod	form the matrix-vector product of a vector with the Hessian matrix
usetup	set up the data structures for unconstrained minimization
ush	evaluate the sparse Hessian matrix
uvarty	determine the type of each variable.
report	obtain statistics concerning function evaluation and CPU time used

which are: *cuter.tar.gz, sifdec.tar.gz, mastsif_small.tar.gz, mastsif_large.tar.gz.*

There are two ways to install CUTEr and sifdec—script-based and the portable Makefiles based, which are generated from architecture-independent Imakefiles. If you use the second method, you have to download msub.c and issue the command *"gcc -o msub msub.c"* to produce an executable program *msub* which will be used to convert imakefiles to makefiles, and put it in a directory which the environment variable *PATH* contains. The web site to download it is

http://www.dubois.ws/software/imake-stuff.

The installation procedure is described as follows, including three parts—the installation of CUTEr, the installation of test problems (SIF files) and the installation of SifDec (the SIF file decoder).

1. Move the file cuter.tar.gz to a new directory of your choice—we shall refer to this directory as *$CUTERDIR*—and issue the commands

表 1.2: Tools For Constrained Problems

Tool name	Brief description
ccfg	evaluate constraint functions values and possibly gradients
ccfsg	same as ccfg, in sparse format
ccifg	evaluate a single constraint function value and possibly gradient
ccifsg	same as ccifg, in sparse format
cdh	evaluate the Hessian of the Lagrangian
cdimen	get the number of variables and constraints involved
cdimse	determine number of nonzeros to store the Lagrangian Hessian in finite element format
cdimsh	determine number of nonzeros to store the Lagrangian Hessian in coordinate format
cdimsj	determine number of nonzeros to store the matrix of gradients of the objective function and constraints, in sparse format
ceh	evaluate the sparse Lagrangian Hessian in finite element format
cfn	evaluate function and constraints values
cgr	evaluate constraints gradients and objective/Lagrangian gradient
cgrdh	same as cgr, plus Lagrangian Hessian
cidh	evaluate the Hessian of a problem function
cish	same as cidh, in sparse format
cnames	obtain the names of the problem and its variables
cofg	evaluate function value and possibly gradient
cprod	form the matrix-vector product of a vector with the Lagrangian Hessian
cscfg	evaluate constraint functions values and possibly gradients in sparse format
cscifg	same as cscfg, for a single constraint
csetup	set up the data structures for constrained minimization
csgr	evaluate constraints and objective/Lagrangian function gradients
csgreh	evaluate both the constraint gradients, the Lagrangian Hessian in finite element format and the gradient of the objective/Lagrangian in sparse format
csgrsh	same as csgreh, in sparse format instead of finite element format
csh	evaluate the Hessian of the Lagrangian, in sparse format
cvarty	determine the type of each variable
report	obtain statistics concerning function evaluation and CPU time used

prompt% gunzip cuter.tar.gz

prompt% tar xvf cuter.tar

On GNU-based LINUX systems, this is also done by the single command *"tar zxvf cuter.tar.gz"*

2. After unpacking, in the current directory *$CUTERDIR* we could find two

links to *install_cuter, install_script_cuter*, which are located in *$CUTERDIR/build/scripts*, and the directories *build, common, config and log.* The script *install_cuter* will install CUTEr based on imakefiles, and *install_script_cuter* will install CUTEr based on scripts. Issue command

prompt% install_cuter (*install_script_cuter*)

to install the evaluation tools in CUTEr according the prompt. Both single and double precision instances of CUTEr may be installed for the same machine-operating system-compiler-size combination. If you wish, you may redefine the directory name and give it whatever name you like.

3. Define the environment variable *CUTER* as *$CUTERDIR*, *MYCUTER* as *$MYCUTERDIR*, and *MASTSIF* to point to your main SIF repository, and in addition, update the environment variable *MANPATH* to include *$CUTER/common/man*, *PATH* to include *$MYCUTER/bin*, the environment variable *MATLABPATH* to include *$MYCUTER/$PREC/bin*, *$CUTER/common/src /matlab* and the environment variable *LIBPATH* (or LD_LIBRARY_PATH) to include *$MYCUTER/$PREC/lib* (see *$CUTER /common/doc/readme.cshrc* and *$CUTER/common/doc/readme.bashrc* for examples on how to do this).
4. If the installation is based on imakefiles, issue the command

   prompt% install_mycuter

   to complete the installation for CUTEr.
5. Unpack the SIF files to the directory *$MASTSIF*. First move *mastsif_small.tar.gz* or *mastsif_large.tar.gz* or *netlib.tar.gz* to the directory *$MASTSIF*, then issue the command

   prompt% gunzip ∗ ∗ ∗.tar.gz

   prompt% tar xvf ∗ ∗ ∗.tar

   or a single command *tar zxvf ∗ ∗ ∗.tar.gz* on GNU-based LINUX systems.
6. Install the SIF decoder. The procedure is the same as the installation of CUTEr. First move the file *sifdec.tar.gz* to a directory different from *$CUTERDIR*, which will be referred as *$SIFDECDIR*, and unpack it. Issue the command

   prompt% install_sifdec

   to install the SIF decoder based on imakefiles, or prompt% install_script_sifdec

to install the SIF decoder based on scripts. The installation procedure will produce an directory named according to the *architecture*, for example, SifDec.medium.sun.lnx.g77, which will be referred as *$MYSIFDECDIR*. Then Define the environment variable *SIFDEC* as *$SIFDECDIR*, MYSIFDEC as *$MYSIFDECDIR*, and in addition, update the environment variable *MANPATH* to include *$SIFDEC/common/man*, *PATH* to include *$MYSIFDEC/bin* (see *$SIFDEC/common/doc/ readme.cshrc* and *$SIFDEC/common/doc/readme .bashrc* for examples on how to do this). If the installation is based on imakefiles, Then issue the command

prompt% install_mysifdec

to complete the installation for the SIF decoder.

7. Both single and double precision instances of CUTEr and SifDec may be installed for the same machine-operating system-compiler-size combination. You can first install the double precision by default, then install the single precision in the same procedure, only provided that the machine-operating system-compiler-size combination is the same.
8. Issue the command

   prompt% sdgen ROSENBR (sdgen -s ROSENBR)

   to see whether your installation is successful.

The installer will record all installations in *$CUTER/log/install.log* and *$SIFDEC/log/install.log*. If you want to reinstall, just remove *$MYCUTER* (or *$MYSIFDEC*) and *$CUTER/log/install.log* (or *$SIFDEC/log/install.log*) and repeat the installation procedure.

In the process of installing CUTEr, the installer makes the directories *$MYCUTER/bin*, *$MYCUTER/$PREC/config*, *$MYCUTER/$PREC/specs*, *$MYCUTER/$PREC/lib* and *$MYCUTER/$PREC/bin*, casting files from *$CUTER/ common/src* to corresponding directory, build the evaluation tools into the lib file *$MYCUTER/$PREC/lib/libcuter.a*. When you use the evaluation functions such as *ufn* in your program, just link your program with the lib—libcuter.a. The installer for SIFDEC also make directories *$MYSIFDEC/bin, $MYSIFDEC/$PREC/*

config and *$MYSIFDEC/$PREC/bin* build the decoder for the corresponding precision (single and double) and put it in the directory *$MYSIFDEC/$PREC/bin*.

As illustrated above, we can reinstall CUTEr by deleting the directory *$MYCUTER/$PREC* or *$MYSIFDEC$PREC* and the corresponding item in *$CUTER/log/install.log* or *$SIFDEC/log/install.log*. The work can also be done by the script *un_install* provided by the soft. Goto the directory $CUTER or $SIFDEC and issue the command

prompt% un_install $CUTERDIR/CUTEr.medium.sgi.lnx.g77

or for SifDec, issue

prompt% un_install $SIFDEC/SifDec.medium.sgi.lnx.g77

If this directory contains both the single and double precision instances, you will be prompted for which should be removed.

The *update_cuter* (or *update_sifdec*) script is used to upgrade over time (or fix if necessary) the tools supplied in the CUTEr (or SifDec) package, and to install newer instances of the tools. There are two forms of the command. In its first form, update cuter takes two command-line options, as follows

prompt% update_cuter filename

where filename is the name of the file to upgrade, possibly specified with a path. Suppose, for example, that the file *ufn.f* has been improved so as to perform its task faster, upgrading your current instance of CUTEr is achieved by typing

prompt% update_cuter ufn.f

at the command prompt. This command first copies the new source file to proper location, which is in this case *$CUTERDIR/common/src/tools*. If there are currently both single and double precision instances, you will be asked to choose which you would like to update; if only there is only one instance under *$MYCUTER*, the precision will be chosen accordingly. The script then casts and compiles the incoming file, and finally updates the CUTEr library (i.e. *$MYCUTER/double/lib/libcuter.a*). Of course, corresponding actions are performed depending on the type of filename: if it is a script, it is only cast, and stored in its proper place, and if it is a documentation file, it is simply moved to *$CUTERDIR/common/doc*.

In its second form, *update_cuter* takes three command-line options, described as follows

prompt% update_cuter -a filename

where filename is the name of a file describing a list of CUTEr files to be upgraded. The file filename should contain

1. on its first line, the directory where the new (upgraded) files can be found, and
2. on subsequent lines, the names of those upgraded files, possibly preceded by their destination directory (which is not necessary).

A single file per line should be given. An example input file might be

```
/home/upgrade
ufn.f
install_cuter
compiler.cry.unc.f90
sdknit.
```

As above, CUTEr copies these files from /home/upgrade to their proper location, prompts for the precision required (if necessary), casts and, where necessary, compiles the incoming files, and updates the specified instance stored under *$MYCUTER*. The additional command-line option -m forces update cuter to simply move the files to their proper location and to skip compilation. Help may be obtained from update cuter through either of the *-h, -help* or *–help* flags.

The CUTEr (include SifDec) package is distributed with three default "sizes": large, medium, and small. These sizes refer to the size of the memory available for problem decoding and solution, and hence are directly related to the size (the amount of data) of the problems that CUTEr can tackle. Typically, when running on a problem with too many variables, CUTEr and SifDec will complain that one of the size parameters is too small and stop. You then have to increase this parameter (if this is possible on your machine) in order to handle the problem. In CUTEr, the size of tools is defined by five parameters *LIWK*, *LWK*, *LLOGIC*, *LCHARA*, *LFUVAL*. In SifDec, the size of tools is defined by the following parameters, *NMAX*, *NGMAX*, *NGRMAX*, *NGPVMX*,

NELMAX, *NLMAX*, *NEVMAX*, *NINMAX*, *NSETVC*, *NEPVMX*, *LA*, *NINDEX*, *NRLNDX*, *NBMAX*, *NSMAX*, and *NOBMAX*. In fact, the installer will produce a file named *tools.siz* (*sifdec.siz* for SifDec) which defines the values of these parameters, copy it to *$MYCUTER/$PREC/config*, then copy the source files from *$CUTER/common/src* to a temporary directory, find lines begin with "C#{sizing}" and and substitute it with the file, concurrently substitute CBIG (specifies the large size), CMED (the medium size), CTOY (the small size) or CCUS (the custom size) with spaces, and define some other parameters in *local.f* related with the *architecture*, which is the combination of

machine-operating system-compiler-size-precision.

This process is called as *casting files*. If the size is not enough to tackle your problem, you have to modify the corresponding parameters in *tools.siz* (*sifdec.siz* for SifDec), and rebuild your CUTEr (or SifDec) instance, namely *casting* the source files again and rebuild them. The process can be seen as a partial re-installation. The package provide such a script to do this work. First modify corresponding parameters in *$MYCUTER/$PREC/config/tools.siz* (*$MYSIFDEC/$PREC/config/sifdec.siz* for SifDec)

prompt% rebuild *$MYCUTER* (or *$MYSIFDEC*)

to rebuild the current instance.

If using the imakefile-based CUTEr and SifDec, simply change to directory *$MYCUTER* (or *$MYSIFDEC*) and issue a command "***make -s all***" to make sure that everything that needs to be rebuilt is rebuilt.

§1.4 Drivers and Interfaces for Optimization Package

This section describes existing CUTEr interfaces with optimization and linear algebra packages and how to create a new interface.

As we know, if we want to test our algorithm package with a problem, we have to define subroutines which evaluate the numerical function values and the corresponding derivatives of the problem, which will be called in our algorithm package. When we test our algorithm in CUTEr, what we have to do is the same. First we should write a driver program—the main program, which call the evaluation tools in CUTEr, such as USETUP, UFN, UFG, etc. to get the number of variables and evaluate the functions and derivatives of the problem which is written in a SIF file before calling our own algorithm code. Next we should use the SIF decoder to decode a SIF file produce the Fortran files which the evaluation tools will use. Then we should compile them and link them with our algorithm subroutine to build an executable program and run it to see the results. Notice that the evaluation tools subroutines such as USETUP, UFN, UFG, etc. have been compiled and build into a lib file—libcuter.a, so what we have to do is to link the lib and decoded Fortran files with our driver and our algorithm package. This process would be very tedious if we want to test a number of problems written in SIF format. So the CUTEr provides some interfaces which are written in shell language for our convenience. There are many interfaces existed in CUTEr, such as *sdcob*, *sdmns*, etc. Let us use the generic interface sdgen/gen for illustration. Other interfaces are almost the same. Their calling sequences are as follows

```
sdgen [-s] [-h] [-k] [-o j] [-l secs] [-debug] probname[.SIF],
gen [-n] [-h] [-s] [-k] [-o j] [-l secs] [-debug].
```

The purpose of sdgen is to SIF-decode probname.SIF, set environment variables defining object and specification files necessary to compile the main *genmin* executable, and launch *runpackage*. The script gen is similar, except that it assumes that the problem has already been decoded by the SIF decoder. The main executable is linked, compiled and run by runpackage. An important difference with previous versions of CUTE is that runpackage is independent of any interface. The arguments of *sdgen* and *gen* are thoroughly described in the man pages. The main object files for the supported packages (i.e. in this case, genma.o) should be placed in (or symlinked to from the directory) *$MYCUTER/ precision/bin*, while the required specification files should be placed in (or symlinked to from) the current directory.

When we specify the interface for our own algorithm package, what we have to do is to modify environment variables *PAC*, *PACKAGE*, *PACKOBJ* and *SPECS* define in *sdgen/gen* and save them in other files. For illustrative purposes, we assume the package for which one wishes to provide an interface is called *pack*, which is composed of "sub1.f", "sub2.f" and "sub3.c". The package will read some parameters such as difference step, output level, etc., from a data file named as *"pack.ini"*. The driver program is written in the file named as "packma.f" (which must be ended with "ma"). First Compile this program into an object file called "packma.o", and put it in the directory "*$MYCUTER/double/bin*" for double precision or put it in the directory "*$MYCUTER/single/bin*" for single precision. Next compile the double precision instance of the complete set of programs contained in the package (sub1.f, sub2.f and sub3.c for example) into object files (for example, sub1.o, sub2.o and sub3.o). Repeat this for the single precision instance of the package, and put them in the directory "*$MYCUTER/double/bin*" for double precision or in the directory "*$MYCUTER/single/bin*" for single precision instance. Next create the new directory *$CUTERDIR/common/src/pkg/pack*, and copy the file *$SPEC* (which is pack.ini in this example) to the directory. Then copy *sdgen* and *gen* to the two files *sdpack* and *pack* respectively, and modify environment variables *PAC*, *PACKAGE*, *PACKOBJ* and *SPECS* defined in the two files *sdpack* and *pack* as follows:

setenv PAC pack
setenv PACKAGE pack
setenv PACKOBJ "sub1.o sub2.o sub3.o"
setenv SPECS "pack.ini"

Finally issue the command

prompt% sdpack *PROB*

at the command prompt, where *PROB* is a SIF file's name, for instance *ROSENBR*, and we can test our algorithm in CUTEr now!

第二章 Subspace Quasi-Newton Trust Region Methods

This chapter studies subspace properties of trust region methods for unconstrained optimization, assuming the approximate Hessian is updated by quasi-Newton formulae. It is shown that the trial step obtained by solving the trust region subproblem is in the subspace spanned by all the gradient vectors computed. Thus, the trial step can also be defined by minimizing the quasi-Newton quadratic model in the subspace. Based on this observation, some subspace trust region algorithms are proposed. Numerical results on a lot of problems in CUTEr are also reported.

§2.1 Introduction

Consider the general unconstrained optimization problem

$$\min_{x\in\Re^n} f(x), \tag{2.1.1}$$

where $f(x)$ is a continuously differentiable function defined in $\Re^n$. Trust region methods for the unconstrained optimization problem (2.1.1) compute a trial step in each iteration. Such a trial step is required to be within a "trust region", which is a region near the current iterate point. Different choices of trust regions and different models give different trust region algorithms. The most commonly used model is the quadratic one, which gives the following trust region subproblem:

$$\min_{s\in\Re^n} \quad \phi_k(s) \equiv g_k^T s + \frac{1}{2} s^T B_k s \tag{2.1.2}$$

$$s.\ t. \quad \|s\|_2 \le \Delta_k, \tag{2.1.3}$$

where $g_k = g(x_k)$ is the gradient of the objective function at the current iterate x_k, B_k is an approximate Hessian, and $\Delta_k > 0$ is the trust region radius. In practice, B_k is updated by using quasi-Newton formulae. More detailed discussions on trust region algorithms can be found in Fletcher[15], Conn, Gould and Toint[10] and Yuan[42]. The following is a description of a model trust region algorithm for unconstrained optimization.

Algorithm 2.1.1. *(A Model Trust Region Algorithm)*

Step 1. *Given* $x_1 \in \Re^n$, $\Delta_1 > 0$, $0 < \tau_3 < \tau_4 < 1 < \tau_1$, $0 \leq \tau_0 \leq \tau_2 < 1$, $\tau_2 > 0$, $\epsilon \geq 0$, $k := 1$;

Step 2. *Solving (2.1.2)–(2.1.3) giving* s_k, *If* $\|s_k\| \leq \epsilon$ *then stop;*

Step 3. *Compute*

$$\eta_k = \frac{Ared_k}{Pred_k} = \frac{f(x_k) - f(x_k + s_k)}{\phi_k(0) - \phi_k(s_k)}. \tag{2.1.4}$$

Step 4. *Set*

$$x_{k+1} = \begin{cases} x_k + s_k & if \quad \eta_k > \tau_0, \\ x_k & otherwise; \end{cases} \tag{2.1.5}$$

Choose Δ_{k+1} *that satisfies*

$$\Delta_{k+1} = \begin{cases} [\tau_3\|s_k\|, \tau_4\Delta_k] & if \quad \eta_k < \tau_2, \\ [\Delta_k, \tau_1\Delta_k] & otherwise; \end{cases} \tag{2.1.6}$$

Step 5. *Update* B_{k+1}, $k := k + 1$, *go to Step 2.*

In this chapter, we study the case when the matrix B_k is updated by quasi-Newton methods, namely B_{k+1} is generated by adding a modification to B_k and satisfies the quasi-Newton equation

$$B_{k+1}s_k = y_k = g(x_k + s_k) - g(x_k). \tag{2.1.7}$$

When the number of variables is very large, the solution of (2.1.2)–(2.1.3) would be very cost. So people developed various approximate methods for the solution of (2.1.2)–(2.1.3), such as the dogleg ([29], [30]) and double dogleg techniques ([12]), the truncated CG method ([36], [43]), two dimensional subspaces minimization method ([4]), subspace CG methods ([39]) and subspace-iterated methods ([7]). We think that how to solve the subproblem (2.1.2)–(2.1.3) as exactly as possible in an acceptable effort is still worth studying. A recent work on line search methods given by Gill and Leonard[17] developed a practical reduced Hessian method. Improvements and modifications on such methods are made by Vlček and Lukšan[38]. Motivated by these results, we explore the subspace properties of trust region methods when the approximate Hessian is updated by quasi-Newton formulae. It is found that the trial step s_k defined by the trust region subproblem is also always in the subspace spanned by $g(x_1)$, $g(x_2)$, ..., $g(x_k)$, even if B_k is not positive definite (If B_k is positive definite, the result would be obvoius by the study on line search methods, see, e.g. [17] and [38]). Therefore it is equivalent to solve the trust region subproblem within this subspace. Based on this observation, we can solve a smaller trust region subproblem exactly in the algorithm, which would reduce the computations, particularly for early iterations.

The chapter is organized as follows. The equivalence of the trust region subproblem and that in the subspace is proved in the next section. In §2.3, two subspace quasi-Newton trust region algorithms are proposed. Some numerical results using BFGS and SR1 updating formulae are reported in §2.4.

§2.2 Subspace Properties of the Trial Step

In this section, we study the subspace properties of the trial step s_k at the k-th iteration, which is assumed to be the solution of the trust region subproblem (2.1.2)–(2.1.3). The following Lemma, about the trial step s_k, can be found in Gay[16] and Sorensen[35].

Lemma 2.2.1. *The vector $s_k \in \Re^n$ is a global solution of (2.1.2)–(2.1.3) if and only if $\|s_k\| \leq \Delta_k$ and there exists a scalar $\lambda_k \geq 0$ such that $B_k + \lambda_k I$ is positive semi-definite and*

$$\begin{aligned} (B_k + \lambda_k I)s_k &= -g_k, && (2.2.1) \\ \lambda_k(\Delta_k - \|s_k\|) &= 0. && (2.2.2) \end{aligned}$$

Consider a special case when B_k is a two by two block diagonal matrix having the form

$$B_k = \begin{bmatrix} B_k(1:r,1:r) & 0 \\ 0 & \sigma I \end{bmatrix} \tag{2.2.3}$$

where $B_k(1:r,1:r)$ is the r-th leading submatrix of B_k(see Golub and Van Loan[18]). From (2.2.1) and (2.2.3), it is easy to see that $s_k(r+1:n)$ (the last $n-r$ components of s_k) are all zero if $g_k(r+1:n) = 0$. This simple example shows that if B_k has certain structure, the trial step will be always in a subspace.

In the above example, the subspace is spanned by the first r coordinate vectors $e_i (i = 1, ..., r)$. A similar result is also true for other subspaces. If $B_k + \lambda_k I$ is positive definite, the result would follow immediately from the theory on line search methods, see, e.g. [17] and [38]. When $B_k + \lambda_k I$ is not invertible, the result is still true for trust region methods.

Lemma 2.2.2. *Let S_k be an r $(1 \leq r \leq n)$ dimensional subspace in $\Re^n$, and $Z_k \in \Re^{n\times r}$ is an orthonormal basis matrix of S_k, namely*

$$S_k = Span\{Z_k\}, \quad Z_k^T Z_k = I_r. \tag{2.2.4}$$

Suppose that $g_k \in S_k$ and B_k is a symmetric matrix satisfying

$$B_k z \in S_k, \qquad \forall\, z \in S_k \tag{2.2.5}$$

$$B_k u = \sigma u, \qquad \forall\, u \in S_k^{\perp}, \tag{2.2.6}$$

where $\sigma > 0$ is a positive number. Then problem (2.1.2)–(2.1.3) is equivalent to

$$\min_{\bar{s}\in\Re^r} \quad \bar{\phi}_k(\bar{s}) \equiv \bar{g}_k^T \bar{s} + \frac{1}{2}\bar{s}^T \bar{B}_k \bar{s} \tag{2.2.7}$$

$$s.\ t. \qquad \|\bar{s}\| \leq \Delta_k, \tag{2.2.8}$$

where $\bar{g}_k = Z_k^T g_k$ and $\bar{B}_k = Z_k^T B_k Z_k$. That is to say, if s_k is a solution of (2.1.2)–(2.1.3), then $Z_k^T s_k$ is a solution of (2.2.7)–(2.2.8). On the other hand, if $\bar{s}_k$ is a solution of (2.2.7)–(2.2.8), $Z_k \bar{s}_k$ must be a solution of (2.1.2)–(2.1.3).

Proof. Let $U_k \in \Re^{n\times(n-r)}$ be a matrix such that $[U_k, Z_k]$ is an $n \times n$ orthogonal matrix. Then for each $s \in \Re^n$, there exists one and only one pair $\bar{s} \in \Re^r,\ u \in \Re^{n-r}$ such that $s = Z_k \bar{s} + U_k u$. Thus, it follows that

$$\begin{aligned} \phi_k(s) \quad &= \quad g_k^T s + \frac{1}{2}s^T B_k s = g_Z^T \bar{s} + g_U^T u \\ &\quad + \frac{1}{2}\bar{s}^T Z_k^T B_k Z_k \bar{s} + \frac{1}{2}u^T U_k^T B_k U_k u + \bar{s}^T Z_k^T B_k U_k u, \end{aligned} \tag{2.2.9}$$

where $g_Z = Z_k^T g_k,\ g_U = U_k^T g_k$. By $g_k \in Span\{Z_k\}$ and (2.2.5)–(2.2.6) we get

$$B_k U_k = \sigma U_k,\ Z_k^T B_k U_k = \sigma Z_k^T U_k = 0,\ g_U = U_k^T g_k = 0. \tag{2.2.10}$$

Hence, it follows from (2.2.9)–(2.2.10) that

$$\phi_k(s) = (g_Z^T \bar{s} + \frac{1}{2}\bar{s}^T Z_k^T B_k Z_k \bar{s}) + \frac{1}{2}\sigma u^T u. \tag{2.2.11}$$

By the orthogonality of Z_k and U_k, we have $\|s\|^2 = \|\bar{s}\|^2 + \|u\|^2$. Now the above relation indicates that (2.1.2)–(2.1.3) is equivalent to

$$\min_{\bar{s}\in\Re^r, u\in\Re^{n-r}} \quad (g_Z^T\bar{s} + \frac{1}{2}\bar{s}^T Z_k^T B_k Z_k \bar{s}) + \frac{1}{2}\sigma u^T u \tag{2.2.12}$$

$$s.\ t. \qquad \|\bar{s}\|^2 + \|u\|^2 \le \Delta_k^2, \tag{2.2.13}$$

with the relation $s = Z_k\bar{s} + U_k u$. Because $\sigma > 0$, it is easy to see that the above problem is equivalent to (2.2.7)–(2.2.8). Thus, (2.1.2)–(2.1.3) is equivalent to (2.2.7)–(2.2.8) with $s = Z_k\bar{s}$. Therefore the lemma is proved. □

Lemma 2.2.3. *Suppose $B_1 = \sigma I$, $\sigma > 0$. The matrix updating formula is any one chosen from amongst SR1, PSB and Broyden family, and B_k is the k-th updated matrix. s_k is obtained by solving (2.1.2)–(2.1.3) exactly and globally and $x_{k+1} = x_k + s_k$, $g_k = \nabla f(x_k)$. Let $\mathcal{G}_k = Span\{g_1, g_2, \cdots, g_k\}$. Then $s_k \in \mathcal{G}_k$, and for any $z \in \mathcal{G}_k$, $w \in \mathcal{G}_k^\perp$, we have*

$$B_k z \in \mathcal{G}_k, \quad B_k u = \sigma u. \tag{2.2.14}$$

Proof. The SR1 formula, PSB formula and Broyden family formulae (see, e.g., Dennis and Schnabel[13] and Fletcher[15]) can be represented as

$$B_{k+1} = B_k + \frac{r_k r_k^T}{r_k^T s_k}, \tag{2.2.15}$$

$$B_{k+1} = B_k + \frac{r_k s_k^T + s_k r_k^T}{s_k^T s_k} - \frac{(r_k^T s_k) s_k s_k^T}{(s_k^T s_k)^2}, \tag{2.2.16}$$

$$B_{k+1} = B_k - \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k} + \frac{y_k y_k^T}{s_k^T y_k} + \theta_k (s_k^T B_k s_k) w_k w_k^T, \tag{2.2.17}$$

where $s_k = x_{k+1} - x_k$, $y_k = g_{k+1} - g_k$, $r_k = y_k - B_k s_k$, $w_k = \frac{y_k}{s_k^T y_k} - \frac{B_k s_k}{s_k^T B_k s_k}$. We only prove the results for PSB formula (2.2.16) by induction. Similar results can be proved for other update formulae.

For $k = 1$, by $B_1 = \sigma I$ and Lemma 2.2.1, we get $s_1 = -\frac{1}{\sigma+\lambda_1} g_1 \in \mathcal{G}_1$. For any $z \in \mathcal{G}_1$, $u \in \mathcal{G}_1^\perp$, we have $B_1 z = \sigma z \in \mathcal{G}_1$, $B_1 u = \sigma u$. Hence the lemma is true for $k = 1$.

Assume that the lemma is true for $k = i$, then for any $z \in \mathcal{G}_i$ and $u \in \mathcal{G}_i^\perp$, we have

$$s_i \in \mathcal{G}_i, \quad B_i z \in \mathcal{G}_i, \quad B_i u = \sigma u. \tag{2.2.18}$$

Now we prove (2.2.14) for $k = i+1$. It is easy to see that $\mathcal{G}_i \subseteq \mathcal{G}_{i+1}$ and $\mathcal{G}_{i+1}^{\perp} \subseteq \mathcal{G}_i^{\perp}$. By induction assumption (2.2.18), we get $B_i s_i \in \mathcal{G}_i$, $r_i = y_i - B_i s_i \in \mathcal{G}_{i+1}$. For any $u \in \mathcal{G}_{i+1}^{\perp} \subseteq \mathcal{G}_i^{\perp}$, we have $r_i^T u = 0$, $s_i^T u = 0$ and $B_i u = \sigma u$ by (2.2.18). Then by (2.2.16) we have $B_{i+1} u = B_i u = \sigma u$ for any $u \in \mathcal{G}_{i+1}^{\perp}$. Hence the second relation in (2.2.14) is true for $k = i+1$. For any $z \in \mathcal{G}_{i+1}$, we have a unique decomposition $z = z_1 + z_2$, where $z_1 \in \mathcal{G}_i$, $z_2 \in \mathcal{G}_i^{\perp} \cap \mathcal{G}_{i+1}$. By (2.2.16) and (2.2.18), we get $B_i z_1 \in \mathcal{G}_i \subseteq \mathcal{G}_{i+1}$, $B_i z_2 = \sigma z_2 \in \mathcal{G}_{i+1}$, and

$$\begin{aligned} B_{i+1} z &= B_i z + t_1 s_i + t_2 r_i = B_i z_1 + B_i z_2 + t_1 s_i + t_2 r_i \\ &= B_i z_1 + \sigma z_2 + t_1 s_i + t_2 r_i, \end{aligned}$$

where t_1, t_2 are scalars. Therefor $B_{i+1} z \in \mathcal{G}_{i+1}$ for any $z \in \mathcal{G}_{i+1}$. Then by $g_{i+1} \in \mathcal{G}_{i+1}$ and Lemma 2.2.2, we get $s_{i+1} = Z_{i+1} z_{i+1} \in \mathcal{G}_{i+1}$ for some z_{i+1}. So we obtain the results by induction . $\square$

By Lemma 2.2.2 and Lemma 2.2.3, we obtain the following theorem.

Theorem 2.2.4. *Let Z_k be a normal orthogonal basis matrix of the subspace $\mathcal{G}_k = Span\{g_1, g_2, \cdots, g_k\}$. Suppose $B_1 = \sigma I, \sigma > 0$, and B_k is the k-th updated matrix given by one formula among SR1, PSB and Broyden family. Then the solution of (2.1.2)–(2.1.3) is equivalent to solving (2.2.7)–(2.2.8). Let s_k be a solution of (2.1.2)–(2.1.3), then $s_k \in \mathcal{G}_k$.*

Lemma 2.2.5. *Let $Z \in \Re^{n\times r}$ be a column orthogonal matrix such that $Z^T Z = I_r$. Suppose that $s_k \in Span\{Z\}$, and the updated matrix $B_{k+1} = Update(B_k, s_k, y_k)$ is obtained by any one formula of SR1,PSB and Broyden family formulae. Denote $\bar{B}_{k+1} = Z^T B_{k+1} Z$, $\tilde{B}_k = Z^T B_k Z, \tilde{s}_k = Z^T s_k, \tilde{y}_k = Z^T y_k$, then $\bar{B}_{k+1} = Update(\tilde{B}_k, \tilde{s}_k, \tilde{y}_k)$.*

Proof. By the assumption $s_k \in Span\{Z\}$ and $Z^T Z = I_r$, we have $s_k = Z Z^T s_k$, and

$$\begin{aligned} &s_k^T y_k = (Z^T s_k)^T Z^T y_k = \tilde{s}_k^T \tilde{y}_k \\ &s_k^T B_k s_k = (Z^T s_k)^T Z^T B_k Z (Z^T s_k) = \tilde{s}_k^T \tilde{B}_k \tilde{s}_k \\ &Z^T B_k s_k = Z^T B_k Z (Z^T s_k) = \tilde{B}_k \tilde{s}_k \end{aligned} \tag{2.2.19}$$

Then by (2.2.19) and multiply Z^T on the left side, Z on the right side of one formula of (2.2.15)–(2.2.17), we obtain the result. $\square$

Theorem 2.2.4 tells us that we can solve the quadratic subproblem (2.1.2)–(2.1.3) by solving (2.2.7)–(2.2.8) in the subspace with the reduced gradient and the reduced approximate Hessian, provided the initial Hessian approximation is appropriately chosen. It follows from Lemma 2.2.3 and Lemma 2.2.5 that in fact the reduced approximate Hessian $\bar{B}_k = Z_k^T B_k Z_k$ of B_k in the subspace $\mathcal{G}_k = Span\{g_1, g_2, \cdots, g_k\}$ can be obtained by updating the reduced matrix $\tilde{B}_{k-1} = Z_k^T B_{k-1} Z_k$, where Z_k is the orthonormal basis matrix of the subspace $\mathcal{G}_k$. By the relation between Z_k and Z_{k-1}, we can get $\tilde{B}_{k-1}$ by $\bar{B}_{k-1} = Z_{k-1}^T B_{k-1} Z_{k-1}$ easily, which will be described in detail in the next section. These subspace properties can reduce the amount of computation significantly during the early iterations.

§2.3 The Algorithm

Due to the subspace properties studied in the previous section, we can construct subspace trust region algorithms based on the traditional trust region philosophy. Let $\|\cdot\|$ denote Euclidean norm. Suppose at the k-th iteration, Z_k have been obtained, which is the orthonormal basis matrix of $\mathcal{G}_k = Span\{g_1, \cdots, g_k\}$. $\bar{s}_k$ is obtained by solving (2.2.7)–(2.2.8) and $s_k = Z_k \bar{s}_k$, $x_{k+1} = x_k + s_k$, $g_{k+1} = \nabla f(x_{k+1})$. Then we have to compute Z_{k+1}, $\bar{g}_{k+1} = Z_{k+1}^T g_{k+1}$ and $\bar{B}_{k+1} = Z_{k+1}^T B_{k+1} Z_{k+1}$ for next iteration. For numerical stability, we could use reorthogonalization procedure (see Daniel et al.[11], Golub and Van Loan[18] and Stewart[37]) to obtain Z_{k+1} by the decomposition

$$g_{k+1} = Z_k u_k + \rho_{k+1} z_{k+1}, \tag{2.3.1}$$

where $u_k = Z_k^T g_{k+1},\ z_{k+1} \perp Span\{Z_k\},\ \|z_{k+1}\| = 1,\ \rho_{k+1} = \|(I - Z_k Z_k^T) g_{k+1}\| \geq 0$. By (2.3.1) and $s_k, g_k \in Span\{Z_k\}$ we obtain

$$z_{k+1}^T g_{k+1} = \rho_{k+1},\ z_{k+1}^T s_k = 0,\ z_{k+1}^T g_k = 0. \tag{2.3.2}$$

If $\rho_{k+1} > 0$, g_{k+1} is accepted and we have $Z_{k+1} = [Z_k, z_{k+1}]$; otherwise, $g_{k+1} \in Span\{Z_k\}$ is rejected and we have $Z_{k+1} = Z_k$. If $Z_{k+1} = [Z_k, z_{k+1}]$, by Lemma 2.2.3 we have

$$B_k z_{k+1} = \sigma z_{k+1}. \tag{2.3.3}$$

Then by (2.3.1)–(2.3.3) we get

$$\begin{aligned}
\bar{g}_{k+1} &= Z_{k+1}^T g_{k+1} = \begin{bmatrix} Z_k^T g_{k+1} \\ z_{k+1}^T g_{k+1} \end{bmatrix} = \begin{bmatrix} u_k \\ \rho_{k+1} \end{bmatrix}, \\
\tilde{s}_k &= Z_{k+1}^T s_k = \begin{bmatrix} \bar{s}_k \\ 0 \end{bmatrix}, \\
\tilde{y}_k &= Z_{k+1}^T y_k = \begin{bmatrix} Z_k^T (g_{k+1} - g_k) \\ z_{k+1}^T (g_{k+1} - g_k) \end{bmatrix} = \begin{bmatrix} u_k - \bar{g}_k \\ \rho_{k+1} \end{bmatrix}, \\
\tilde{B}_k &= Z_{k+1}^T B_k Z_{k+1} = \begin{bmatrix} Z_k^T B_k Z_k & \sigma Z_k^T z_{k+1} \\ \sigma z_{k+1}^T Z_k & \sigma z_{k+1}^T z_{k+1} \end{bmatrix} = \begin{bmatrix} \bar{B}_k & 0 \\ 0 & \sigma \end{bmatrix}.
\end{aligned} \tag{2.3.4}$$

If $Z_{k+1} = Z_k$, by (2.3.1–2.3.2) we have

$$\begin{aligned}
\bar{g}_{k+1} &= Z_{k+1}^T g_{k+1} = Z_k^T g_{k+1} = u_k, \\
\tilde{s}_k &= Z_{k+1}^T s_k = Z_k^T s_k = \bar{s}_k, \\
\tilde{y}_k &= Z_{k+1}^T y_k = Z_k^T y_k = u_k - \bar{g}_k, \\
\tilde{B}_k &= Z_{k+1}^T B_k Z_{k+1} = Z_k^T B_k Z_k = \bar{B}_k.
\end{aligned} \tag{2.3.5}$$

According to Lemma 2.2.5, the reduced approximate Hessian $\bar{B}_{k+1} = Z_{k+1}^T B_{k+1} Z_{k+1}$ in the subspace $Span\{Z_{k+1}\}$ can be obtained by any formula among the SR1,PSB and Broyden family, using $\tilde{B}_k, \tilde{s}_k, \tilde{y}_k$ computed in (2.3.4) or (2.3.5). For example, if we choose the BFGS formula and we have computed $\tilde{B}_k, \tilde{s}_k, \tilde{y}_k$ in (2.3.4) or (2.3.5), by Lemma 2.2.5 we obtain

$$\bar{B}_{k+1} = \tilde{B}_k - \frac{\tilde{B}_k \tilde{s}_k \tilde{s}_k^T \tilde{B}_k}{\tilde{s}_k^T \tilde{B}_k \tilde{s}_k} + \frac{\tilde{y}_k \tilde{y}_k^T}{\tilde{s}_k^T \tilde{y}_k}. \tag{2.3.6}$$

Then by Theorem 2.2.4 we can solve (2.2.7)–(2.2.8) with the reduced matrix $\bar{B}_{k+1}$ and the reduced gradient $\bar{g}_{k+1}$ to obtain the trial step $\bar{s}_{k+1}$ and $s_{k+1} = Z_{k+1}\bar{s}_{k+1}$.

Algorithm 2.3.1. *The Subspace Quasi-Newton Trust Region Algorithm for Unconstrained Optimization*

Step 1. *Given* $x_1 \in \Re^n$, $\Delta_1 > 0$, $0 \le \Delta_{\min} \le 10^{-6}$, $\varepsilon \ge \varepsilon_M \ge 0$, $1 > \nu \ge 0$, $\sigma_1 > 0$, $0 \le \tau_0 < \tau_1 < 0.5 < \tau_2 < 1$, $0 < c_1 < 0.1 < c_2 \le 0.5, c_2 \le c_3 < 1 < c_4$. *Choosing one matrix updating formula amongst SR1,PSB and Broyden family. Compute* $f(x_1)$ *and* $g_1 = \nabla f(x_1)$. *Set* $\bar{B}_1 = \sigma_1$, $\bar{g}_1 = \|g_1\|$, $Z_1 = \left[\frac{g_1}{\|g_1\|}\right]$. *Set* $r := 1$, $k := 1$.

Step 2. *Solve the subproblem (2.2.7)–(2.2.8) to get* $\bar{s}_k$ *and* $\bar{\phi}(\bar{s}_k)$. *If* $|\bar{\phi}(\bar{s}_k)| \leq \varepsilon_M$ *then stop. Otherwise compute* $s_k = Z_k\bar{s}_k$ *and* $\eta_k = \frac{f(x_k)-f(x_k+s_k)}{(-\bar{\phi}(\bar{s}_k))}$

Step 3. *If* $\eta_k \leq \tau_0$ *and* $\Delta_k > \Delta_{\min}$, *set* $\Delta_k := t_1\Delta_k$, *where* $c_1 \leq t_1 \leq c_2$, *and* t_1 *is obtained by using the interpolation technique proposed by Dennis and Schnabel[13]. Go to Step 2;*
If $\eta_k \leq \tau_0$ *and* $\Delta_k \leq \Delta_{\min}$, *the algorithm failed and stop;*
Otherwise, do next step.

Step 4. *Set* $x_{k+1} = x_k + s_k$ *and*

$$\Delta_{k+1} = \begin{cases} c_3\Delta_k, & \text{if } \eta_k < \tau_1 \\ c_4\Delta_k, & \text{if } \eta_k > \tau_2 \text{ and } \|s_k\| \geq 0.8\Delta_k \\ \Delta_k, & \text{otherwise} \end{cases} \tag{2.3.7}$$

Step 5. *Compute* $g_{k+1} = \nabla f(x_{k+1})$. *If* $\|g_{k+1}\| \leq \varepsilon$, *stop; otherwise do next step.*

Step 6. *Obtain (2.3.1) by the reorthogonalization procedure.*

Step 7. *If* $\rho_{k+1} > \nu\|g_{k+1}\|$, *set* $Z_{k+1} = [Z_k, z_{k+1}]$, $r := r + 1$. *Compute* $\tilde{B}_k, \tilde{s}_k, \tilde{y}_k$ *according to (2.3.4);*
Otherwise set $Z_{k+1} = Z_k$ *and compute* $\tilde{B}_k, \tilde{s}_k, \tilde{y}_k$ *according to (2.3.5).*

Step 8. *Obtain* $\bar{B}_{k+1} = Update(\tilde{B}_k, \tilde{s}_k, \tilde{y}_k)$ *by the chosen matrix updating formula in the subspace* $Span\{Z_{k+1}\}$. *Set* $k := k+1$ *and go to step 2.*

If we set $\nu = 0$ in the above algorithm and suppose that there is no floating point error, by Theorem 2.2.4 and Lemma 2.2.5 we know that the algorithm is the same as the traditional trust region algorithm. If $\nu > 0$, the algorithm would be different from the traditional trust region algorithm.

Theorem 2.3.2. *Suppose that* $\{x_k\}$ *is generated by Algorithm 2.3.1,* $f : \Re^n \to R$ *is bounded below and* $\nabla f(x)$ *is uniformly continuous. If* $\bar{B}_k$ *satisfies*

$$\sum_{k=1}^{\infty} 1/M_k = \infty, \tag{2.3.8}$$

where M_k *is the number*

$$M_k = \max_{1\leq i\leq k} \|\bar{B}_i\| + 1, \tag{2.3.9}$$

then the gradient norms $\{\|g_k\|; k = 1, 2, 3, \ldots\}$ *can not be bounded away from* *zero, that is*

$$\liminf_{k\to\infty} \|g_k\| = 0 \tag{2.3.10}$$

Proof. Let $\bar{s}_k$ be the solution of (2.2.7)–(2.2.8), then by Theorem 4 in Powell[31],

$$-\bar{\phi}(\bar{s}_k) \geq \frac{1}{2}\|\bar{g}_k\| \min\left(\Delta_k,\ \frac{\|\bar{g}_k\|}{\|\bar{B}_k\|}\right). \tag{2.3.11}$$

By (2.3.1), we have

$$\|g_k\|^2 = \|u_{k-1}\|^2 + \rho_k^2, \tag{2.3.12}$$

where $u_{k-1} = Z_{k-1}^T g_k$. If $\rho_k > \nu\|g_k\|$, g_k is accepted and $g_k \in Span\{Z_k\}$, so we have $g_k = Z_k Z_k^T g_k = Z_k \bar{g}_k$ and $\|g_k\| = \|\bar{g}_k\|$; Otherwise g_k is rejected and we have $\rho_k \leq \nu\|g_k\|$, by (2.3.5), (2.3.12) and $0 \leq \nu < 1$, we get

$$\|\bar{g}_k\| = \|u_{k-1}\| = \sqrt{\|g_k\|^2 - \rho_k^2} \geq \sqrt{1-\nu^2}\|g_k\|, \tag{2.3.13}$$

(2.3.13) is also right when $\|g_k\| = \|\bar{g}_k\|$, so we always have (2.3.13). By (2.3.11) and (2.3.13), we get

$$-\bar{\phi}(\bar{s}_k) \geq \tau\|g_k\| \min\left(\Delta_k,\ \frac{\|g_k\|}{\|\bar{B}_k\|}\right). \tag{2.3.14}$$

where $\tau = \frac{1}{2}(1-\nu^2) > 0$ is a positive constant. Then in the same way as Powell[33] and Nocedal and Yuan[28], we could get the result by contradiction. □

If we choose $0 < \nu < 1$ in the above algorithm, and $\rho_{k+1} \leq \nu\|g_{k+1}\|$ such that g_{k+1} is rejected, we are in fact lingering on the subspace $Span\{Z_k\}$ minimizing $F(z) = f(x_k + Z_k z)$ by a traditional trust region method on that space. So lingering steps in the subspace will not loop infinitely unless the minimizer lies in the manifold $x_k + Span\{Z_k\}$.

In the above algorithm, we could also use the reinitialization technique proposed by [24] and [17]. For reinitialization, we only need to compute an appropriate value σ_k and replace σ in (2.3.4) with σ_k when g_{k+1} is accepted and $k \geq m$, where m is some positive integer. We have considered the following 7 alternatives, i.e.,

$$\begin{aligned}
\sigma_k^{r0} &= 1,\ \sigma_k^{r1} = \frac{y_0^T y_0}{s_0^T y_0},\ \sigma_k^{r2} = \frac{s_k^T y_k}{\|s_k\|^2}, \\
\sigma_k^{r3} &= \min_{1\le i\le k}\{\frac{s_i^T y_i}{\|s_i\|^2}\},\ \sigma_k^{r4} = \frac{y_k^T y_k}{s_k^T y_k}, \\
\sigma_k^{r5} &= \frac{s_0^T y_0}{s_0^T s_0},\ \sigma_k^{r6} = \min_{1\le i\le k}\{\frac{y_i^T y_i}{s_i^T y_i}\}.
\end{aligned} \tag{2.3.15}$$

By numerical experiments, we find out that in the trust region framework, only considering the reinitialization alone, the performance of the algorithm is not satisfactory and cannot be favourably compared with the algorithm without reinitialization. It is necessary to combine reinitialization with an appropriate implementation of the lingering technique to obtain better numerical results. Suppose g_k is accepted in last iteration (which means $Z_k = [Z_{k-1}, z_k]$), and we have obtained $\bar{s}_k = (\bar{s}_1^{(k)}, \bar{s}_2^{(k)}, \cdots, \bar{s}_r^{(k)})$ by solving (2.2.7)–(2.2.8), where r is the number of columns of Z_k, which is the orthonormal basis matrix of $Span\{g_1, \cdots, g_k\}$. If $|\bar{s}_r^{(k)}| \le \mu_1\|\bar{s}\|$ and $\rho_k/\|g_k\| \le \mu_2$, where $1 > \mu_2 > 0.2 > \mu_1 > 0$, then we think that although g_k is accepted in last iteration, the direction z_k does not have sufficiently contribution to s_k compared with Z_{k-1}. So we can discard z_k and linger on the subspace $Span\{Z_{k-1}\}$. Then we compute $x_{k+1} = x_k + s_k$ and $g_{k+1} = \nabla f(x_{k+1})$. Suppose g_{k+1} is accepted in Step 5, then $Z_{k+1} = [Z_{k-1}, z_{k+1}]$. In this way, the dimension of $Span\{Z_k\}$ will not increase too quickly and we can reduce the amount of computation. If we set $\mu_1 = 0$ and z_k is discarded, we are in fact minimizing f in the subspace $Span\{Z_{k-1}\}$ as iteration proceeds. So unless the minimizer lies in the manifold, lingering steps will not loop infinitely.

Algorithm 2.3.3. *The Subspace Quasi-Newton Trust Region Algorithm with lingering and reinitialization*

Step 1. *Given* $1 > \mu_2 > 0.2 > \mu_1 > 0$ *and some positive integer* $m > 1$. *Other parameters are the same as Step 1 in Algorithm 2.3.1.*

Step 2–Step 5 are the same as Step 2–Step 5 in Algorithm 2.3.1.

Step 6. *If* $|\bar{s}_r^{(k)}| \le \mu_1\|\bar{s}_k\|$, g_k *is accepted in last iteration and* $\rho_k < \mu_2\|g_k\|$, *discard* z_k *and set* $Z_k = Z_{k-1}$; *otherwise* Z_k *is not changed.*

Step 7 is the same as Step 6 in Algorithm 2.3.1.

Step 8. *If g_{k+1} is accepted, compute σ_k according to one formula in (2.3.15) and replace σ in (2.3.4) with σ_k. Others are the same as Step 7 in Algorithm 2.3.1.*

Step 9 is the same as Step 8 in Algorithm 2.3.1.

Algorithm 2.3.3 now is quite different from the traditional trust region algorithms. However, the convergence result for Algorithm 2.3.3 can be proved similar to Theorem 2.3.2.

§2.4 Numerical Results

We have implemented Algorithm 2.3.1 and Algorithm 2.3.3 with different reinitialization parameters defined in (2.3.15), and compared them with a traditional trust region algorithm whose framework is the same as Algorithm 2.3.1 except that (2.1.2)–(2.1.3) is solved instead of (2.2.7)–(2.2.8), where B_k is also updated in the whole space. The matrix updating formulae we used are BFGS and SR1 respectively. When the BFGS formula is adopted to update the approximate Hessian B_k at each iteration, Algorithm 2.3.1 and Algorithm 2.3.3 using the reinitialization parameter $r0$–$r6$ in (2.3.15) are referred as "Sub1BFGSr0–6" and "Sub2BFGSr0–6" respectively, and the corresponding traditional trust region algorithm is referred as "TrBFGS". The matrix update is skipped for the above solvers if $s_k^T y_k \le 10^{-12}\|s_k\|\|y_k\|$. When the SR1 formula is adopted to update the approximate Hessian B_k at each iteration, Algorithm 2.3.1 and Algorithm 2.3.3 using the reinitialization parameter $r0$–$r6$ in (2.3.15) are referred as "Sub1SR1r0–6" and "Sub2SR1r0–6" respectively, and the corresponding traditional trust region algorithm is referred as "TrSR1". The matrix update is skipped for these algorithms if $\|\frac{r_k r_k^T}{r_k^T s_k}\| \le 10^{-12}$ according to [8], where $r_k = y_k - B_k s_k$.

In all the 30 solvers mentioned above, we calculate the trial step using the subroutine GQTPAR which is designed based on the ideas described in Moré and Sorensen[26] with the parameters "$rtol = 0.1, atol = 10^{-16}$". For all algorithms, the parameters in Step 1 is chosen as $\Delta_1 = 1, \Delta_{\min} = 10^{-15}$, $\varepsilon_M = 10^{-20}$, $\varepsilon = 10^{-5}$, $\nu = 10^{-8}$, $\tau_0 = 0$, $\tau_1 = 0.25$, $\tau_2 = 0.75$, $c_1 = 10^{-5}, c_2 = 0.22$, $c_3 = 0.5$

and $c_4 = 2$, and we set $m = 10$, $\mu_1 = 0.1, \mu_2 = 0.8$ for Algorithm 2.3.3. So all the algorithms are terminated if $\|g_k\| < 10^{-6}$, or if the trust region radius is too small, i.e., $\Delta_k < 10^{-15}$ at Step 3, or if the minimum of the quadratic subproblem is too close to zero, i.e., $|\bar{\phi}(\bar{s}_k)| \leq \varepsilon_M = 10^{-20}$ at Step 2. The algorithms were coded in C language using double precision, and the tests were performed on SGI O3800 with one CPU and the memory is limited as 800MB.

We use the performance profile proposed by [14] to display the performance of each solver on the set of test problems. Let $t_{p,s}$ denote the time to solve problem p by solver s. Define

$$r_{p,s} = \frac{t_{p,s}}{t_p^*}$$

where t_p^* is the lowest time required by any solver to solve problem p. If a solver does not solve a problem, the ratio $r_{p,s}$ is assigned a large number, M. Then, the performance profile for each code s, is

$$\rho_s(\tau) = \frac{\text{no. of problems s.t. } r_{p,s} \leq \tau}{\text{total no. of problems}}.$$

This performance profile will also be used to analyze the number of objective function evaluations and gradient evaluations required by each solver. For each solver on each test problem, if the computed $\|g_k\|_2 < 10^{-5}$ or $|f_k - f^*| < 10^{-8}$, where f^* is a local optimal object function value (where the corresponding $\|g_*\|_2 < 10^{-5}$), then we think that the solver succeeds in solving the problem; otherwise, we think the solver fails to solve it.

The test problems are chosen from CUTEr (see [1] and [19]) by the *selection* tool. There are 158 unconstrained test problems in CUTEr. Problems *fletchbv* and *indef* seem unbounded below even for $n = 100$, where n is the number of variables. There are more than one local optimal point for problems *vibrbeam* ($n = 8$) and *broydn7d* ($n = 500$), and different solvers get different local optimal points. None of the solvers can solve the problem *jimack* ($n = 1000$) in one hour. These five problems are excluded, and left 153 problems suitable for testing.

§2.4.1 Numerical results on problems with medium size.

We first test problems whose dimension are not bigger than 1000. There are 69 problems with $n < 50$ (the number of variables is in the parenthesis):

akiva(2), allinitu(4), bard(3), beale(2), biggs6(6), box3(3), brkmcc(2), brownbs(2), brownden(4), cliff(2), cube(2), denschna(2), denschnb(2), denschnc(2), denschnd(3), denschne(3), denschnf(2), djtl(2), edensch(36), engval2(3), expfit(2), growthls(3), gulf(3), hairy(2), hatfldd(3), hatflde(3), heart6ls(6), heart8ls(8), helix(3), hielow(3), hilberta(2), himmelbb(2), himmelbf(4), himmelbg(2), himmelbh(2), humps(2), jensmp(2), kowosb(4), loghairy(2), maratosb(2), mexhat(2), meyer3(3), osbornea(5), palmer1c(8), palmer1d(7), palmer2c(8), palmer3c(8), palmer4c(8), palmer5c(6), palmer6c(8), palmer7c(8), palmer8c(8), pfit1ls(3), pfit2ls(3), pfit3ls(3), pfit4ls(3), rosenbr(2), s308(2), sineval(2), sisser(2), snail(2),yfitu(3), zangwil2(2), 3pk(30), hilbertb(10), osborneb(11), parkch(15), stratec(10) , watson(12),

and 55 problems with $50 \le n < 500$:

chnrosnb(50), deconvu(61), errinros(50), hydc20ls(99), tointgor(50), tointpsp(50), tointqor(50), vareigvl(50), arglina(200), arglinb(200), arglinc(200), brownal(200), chainwoo(100), cosine(100), curly10(100), curly20(100), curly30(100), dixmaana(300), dixmaanb(300), dixmaanc(300), dixmaand(300), dixmaane(300), dixmaanf(300), dixmaang(300), dixmaanh(300), dixmaani(300), dixmaanj(300), dixmaank(300), dixmaanl(300), dixon3dq(100), eigenals(110), eigenbls(110), eigencls(462), engval1(100), extrosnb(100), fletcbv2(100), fletcbv3(100), fletchcr(100), fminsrf2(121), fminsurf(121), mancino(100), modbeale(200), ncb20(110), noncvxu2(100), noncvxun(100), penalty3(200), scosine(100), scurly10(100), scurly20(100), scurly30(100), sensors(100), sparsine(100), sparsqur(100), vardim(200), woods(100),

and 29 problems with $500 \le n \le 1000$:

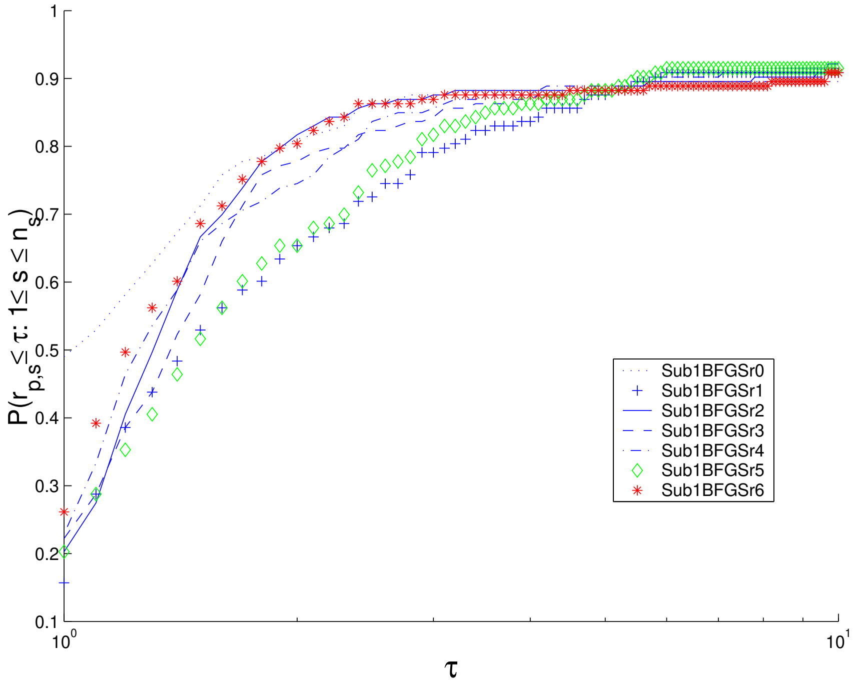
arwhead(500), bdqrtic(500), brybnd(500), cragglvy(500), dqdrtic(500), dqrtic(500), freuroth(500), genhumps(500), genrose(500), liarwhd(500), morebv(500), ncb20b(500), nondia(500), nondquar(500), nonmsqrt(529), penalty1(500), penalty2(500), powellsg(500), power(500), quartc(500), sbrybnd(500), schmvett(500), sinquad(500), srosenbr(500), tointgss(500), tquartic(500), tridia(500), eg2(1000), testquad(1000).

表 2.1: Hard Problems to Solve

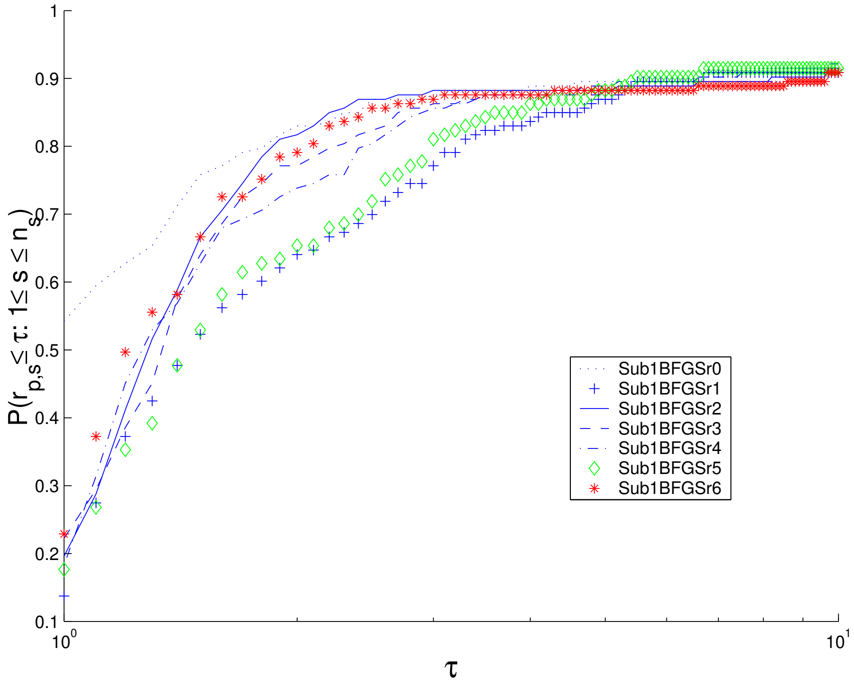
Name(Dim)	Func.Val.	$\|g\|_\infty$
HYDC20LS(99)	0.04175233	62.13303
NONMSQRT(529)	61.32027	0.01179786
PENALTY2(500)	5.380714e+39	3.955348e+16
PENALTY3(100)	0.0009999536	0.005382481
SCOSINE(100)	-99	3956.22
SCURLY10(100)	-10031.63	0.003027471
SCURLY30(100)	-10031.63	0.05164086

In Table 2.1, we listed problems on which none of the above 30 solvers can obtain a solution satisfying $\|g_k\|_2 < 10^{-5}$, and we think that all solvers failed to solve these problems. But they are also compared on due to the advantages of the performance profile proposed by [14].

First, we compared the implementations of Algorithm 2.3.1 using seven alternatives of σ_k (see (2.3.15)). In Figure 2.1 and Figure 2.2, the 7 solvers "Sub1BFGSr0–r6" are compared. We can see that the performances of the algorithms with the reinitialization parameters σ_k^{r1}–σ_k^{r6} cannot be favourably compared with the algorithm without reinitialization (i.e. $\sigma_k = \sigma_k^{r0}$) on about 65% problems of this test set. Except σ_k^{r0}, the most preferred reinitialization parameter is σ_k^{r6}, σ_k^{r2} and σ_k^{r4} are the second and the third most preferred ones respectively. Next we compared the implementations of Algorithm 2.3.3 using seven alternatives of σ_k (i.e. the 7 solvers "Sub2BFGSr0–r6") in Figure 2.3 and Figure 2.4. We can see that the reinitialization parameters σ_k^{r6}, σ_k^{r2} and σ_k^{r0} are the first three most preferred ones, and the differences among them are very small. Then we compared the six solvers "Sub1BFGSr0", "Sub1BFGSr4", "Sub1BFGSr6", "Sub2BFGSr0", "Sub2BFGSr2" and "Sub2BFGSr6" in Figure 2.5 and Figure 2.6. We can see that when the BFGS formula is used, Algorithm 2.3.3 with reinitialization parameters σ_k^{r0} (which means no reinitialization), σ_k^{r2} and σ_k^{r6} outperforms Algorithm 2.3.1 with reinitialization parameters σ_k^{r0}, σ_k^{r4} and σ_k^{r6} on about 70% problems, and the solver "Sub2BFGSr6" is the most preferred

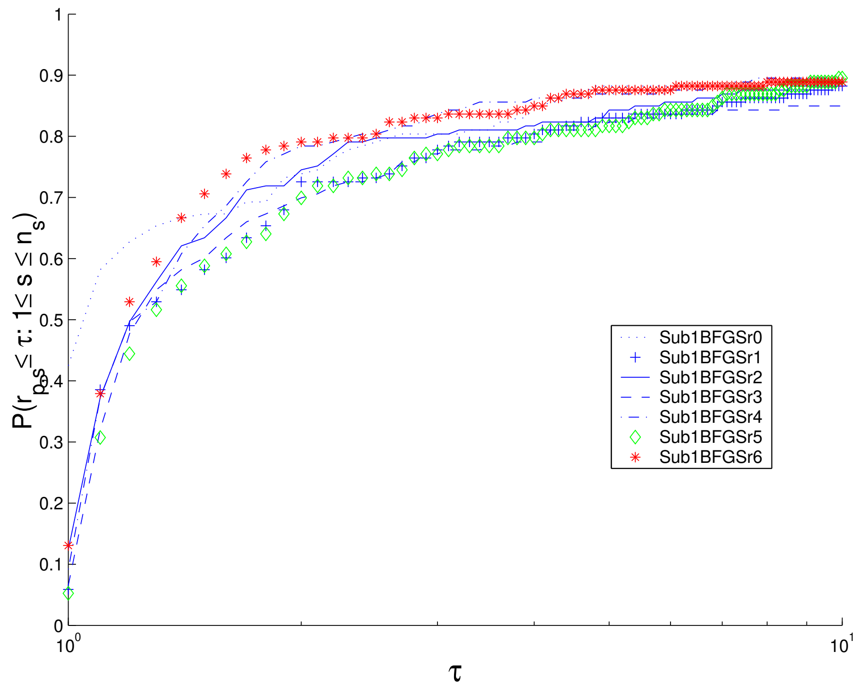


(a)

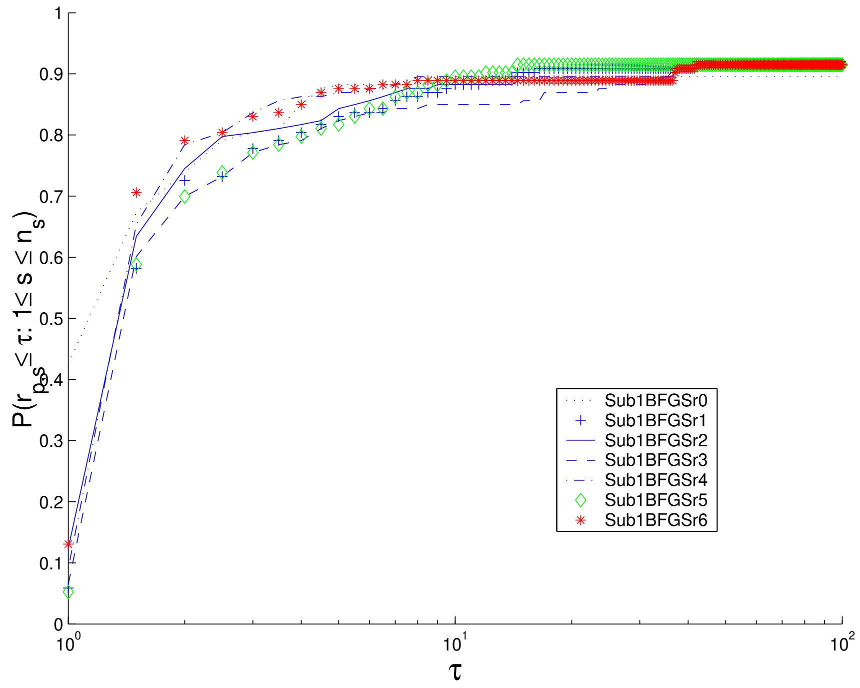


(b)

图 2.1: Performance profile on [0, 10] for the number of (a) function evaluations and (b) gradient evalutions

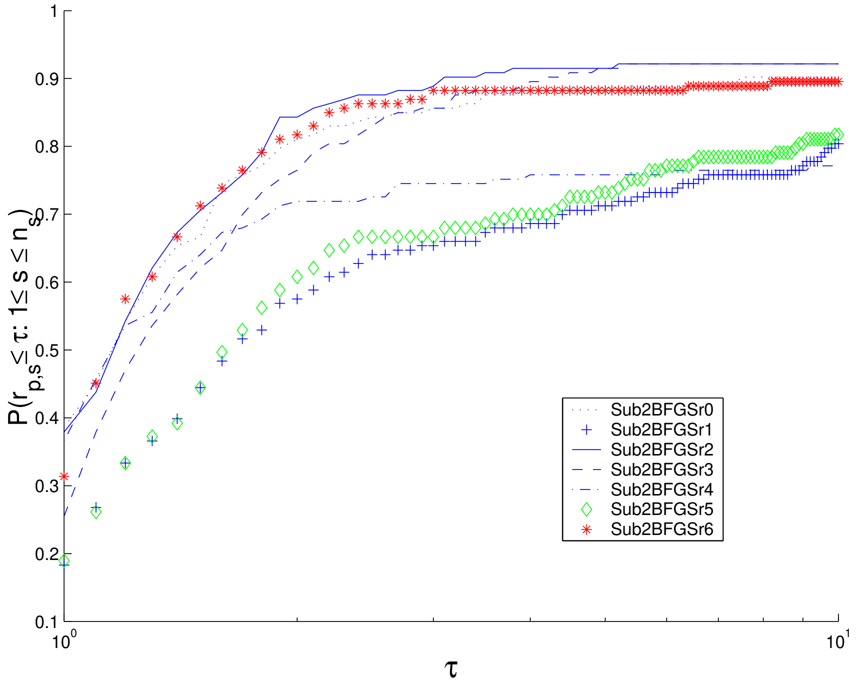


(a)

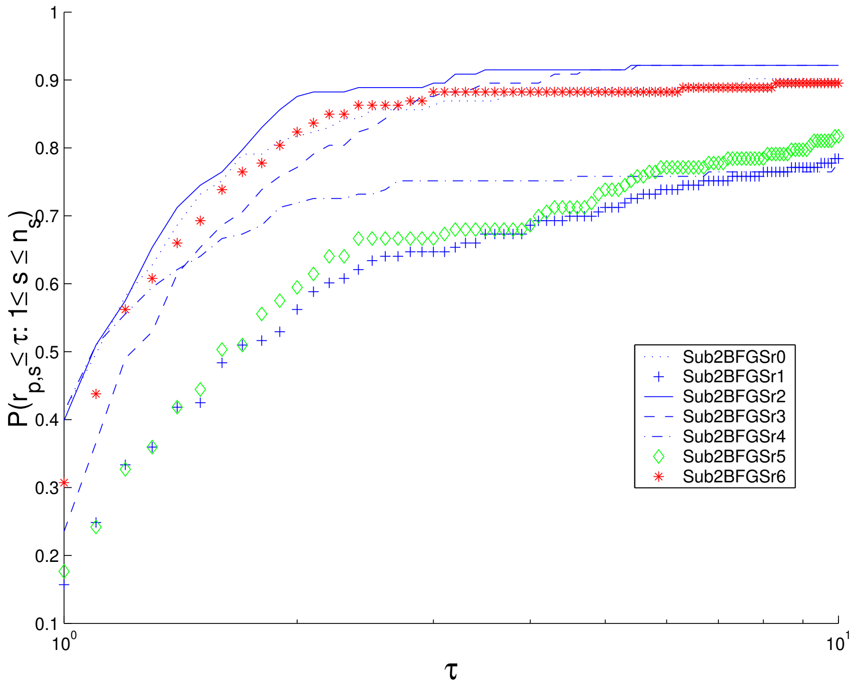


(b)

图 2.2: Performance profile for CPU time on intervals (a)[0, 10] (b)[0,100]

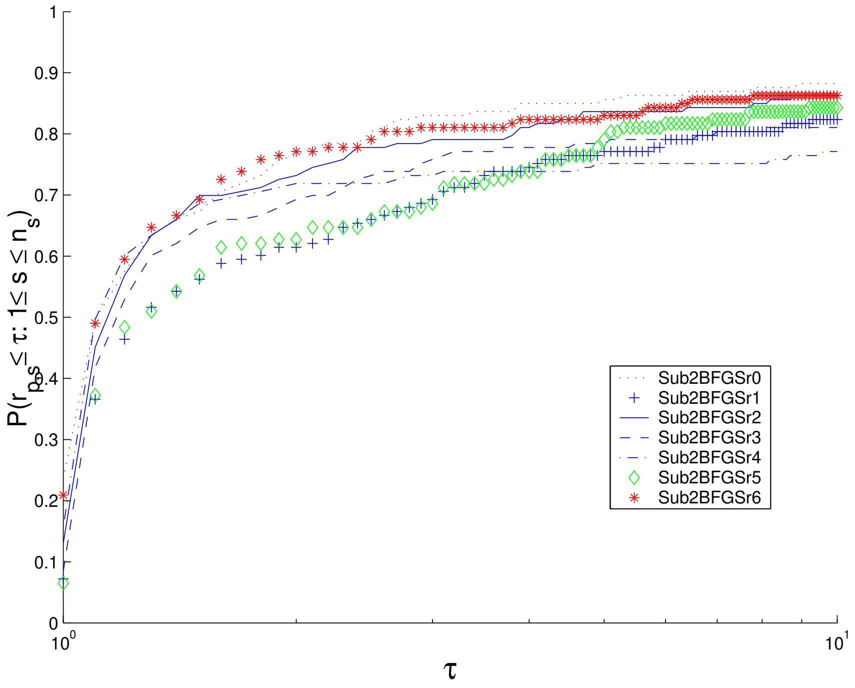


(a)

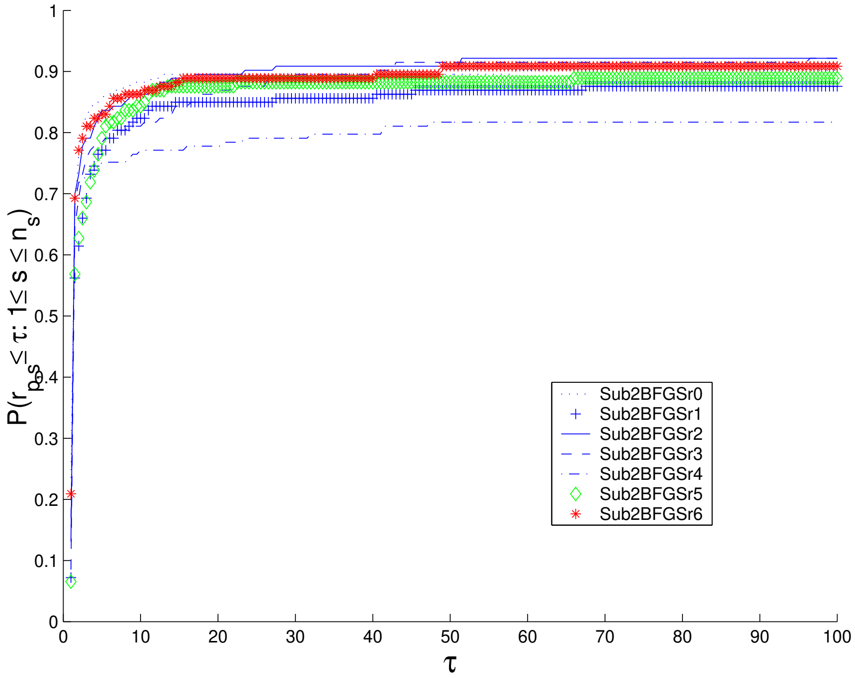


(b)

图 2.3: Performance profile on [0, 10] for the number of (a) function evaluations and (b) gradient evalutions

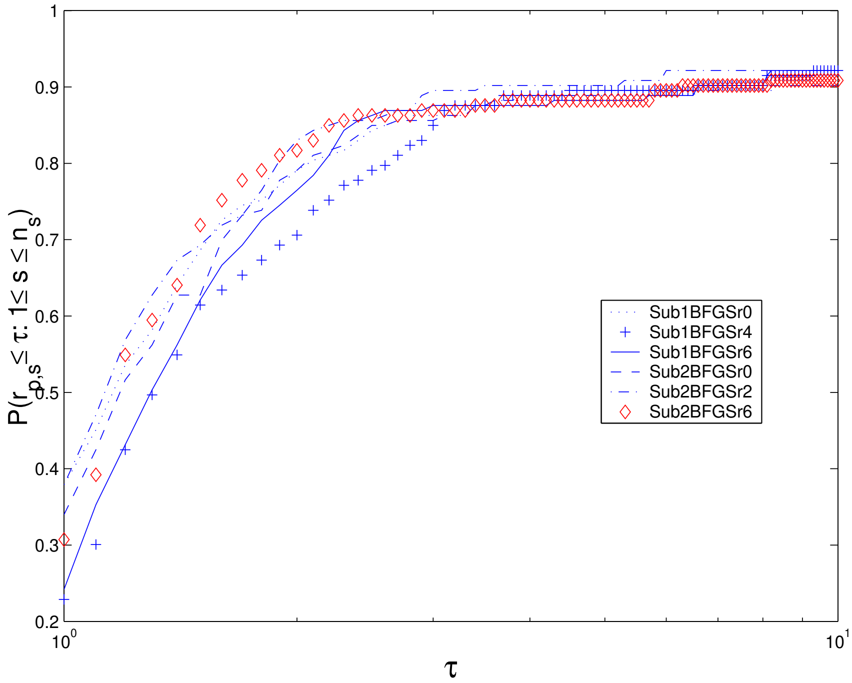


(a)

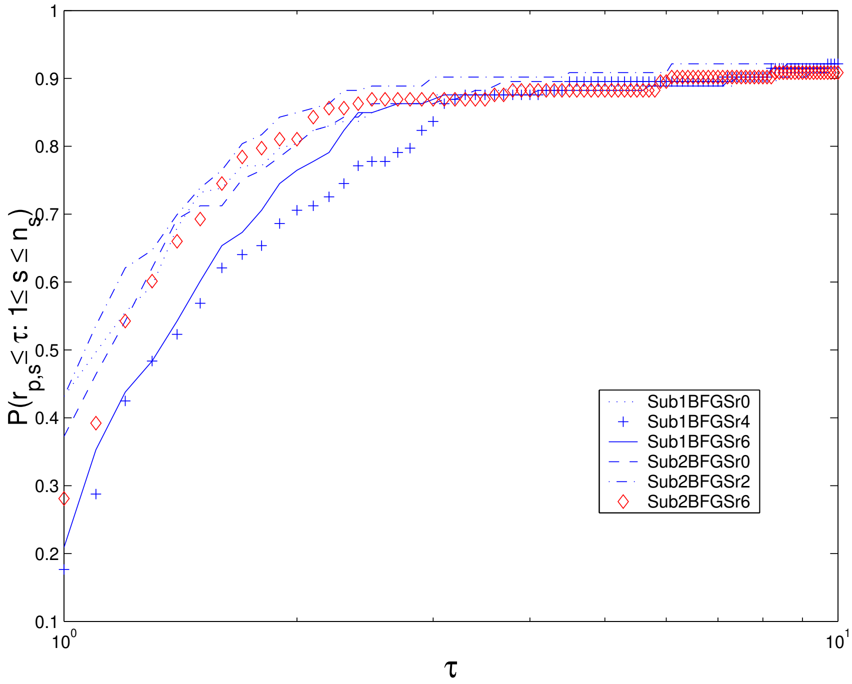


(b)

图 2.4: Performance profile for CPU time on intervals (a)[0, 10] (b)[0,100]

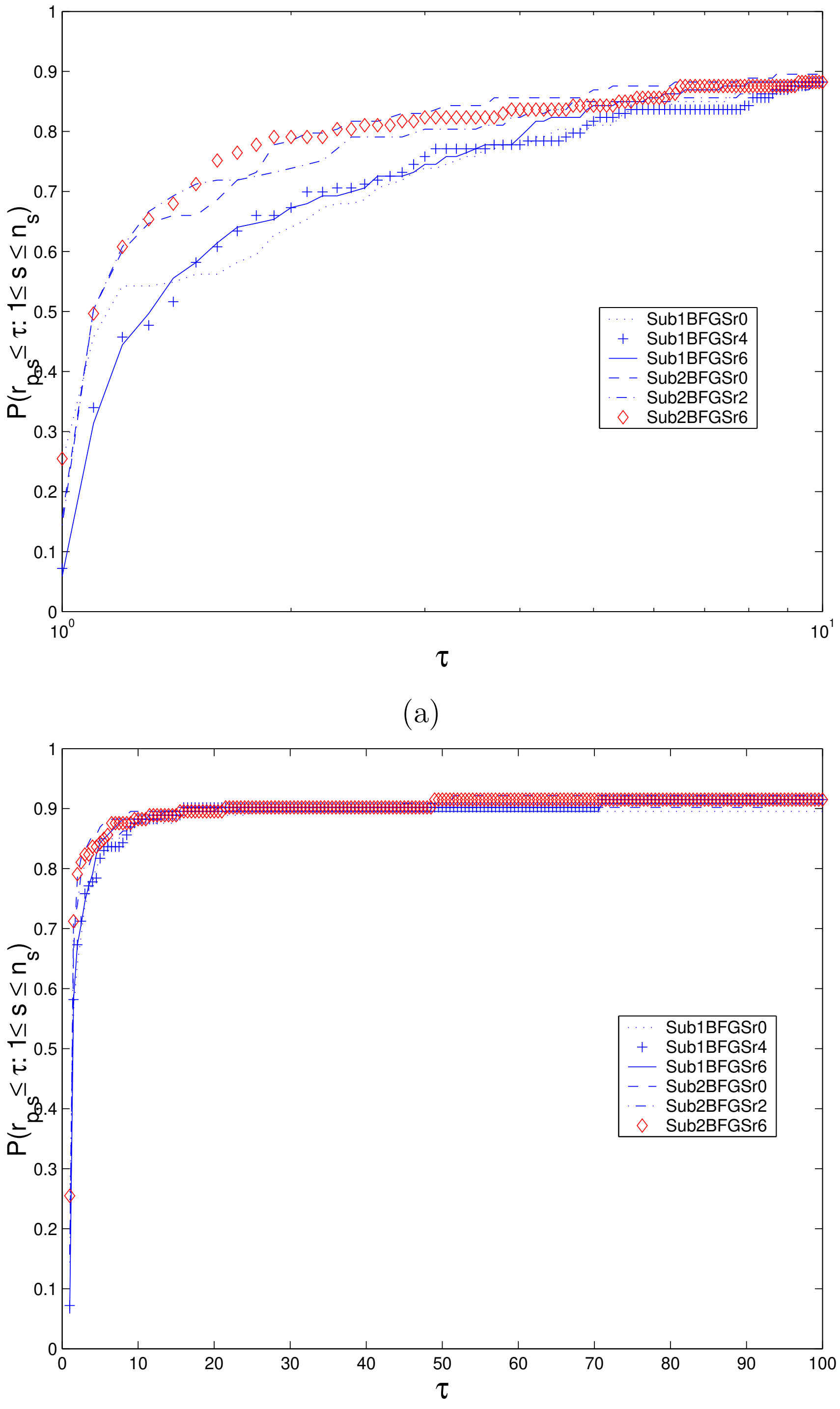


(a)



(b)

图 2.5: Performance profile on [0, 10] for the number of (a) function evaluations and (b) gradient evalutions



(a)

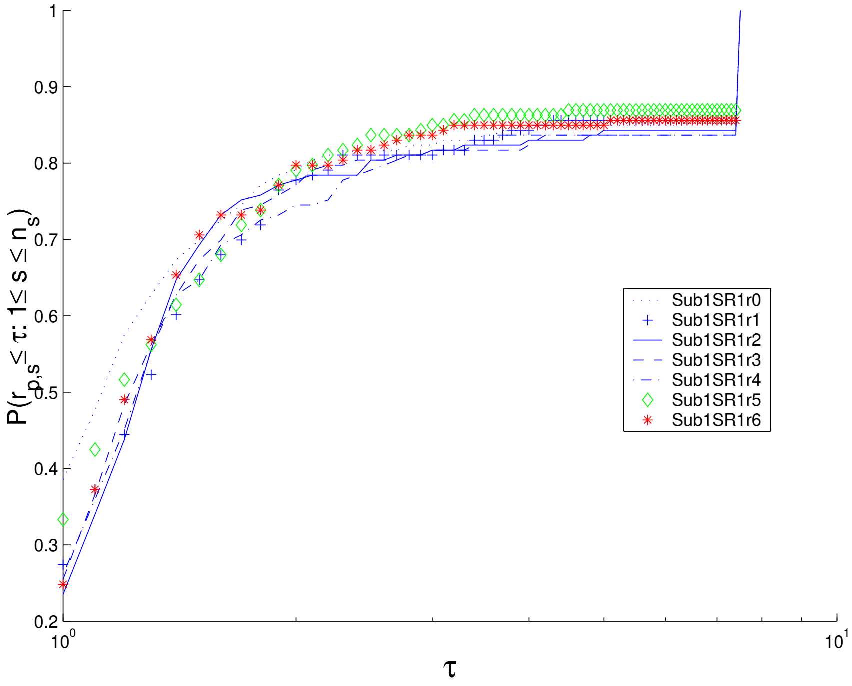
(b)

图 2.6: Performance profile for CPU time on intervals (a)[0, 10] (b)[0,100]

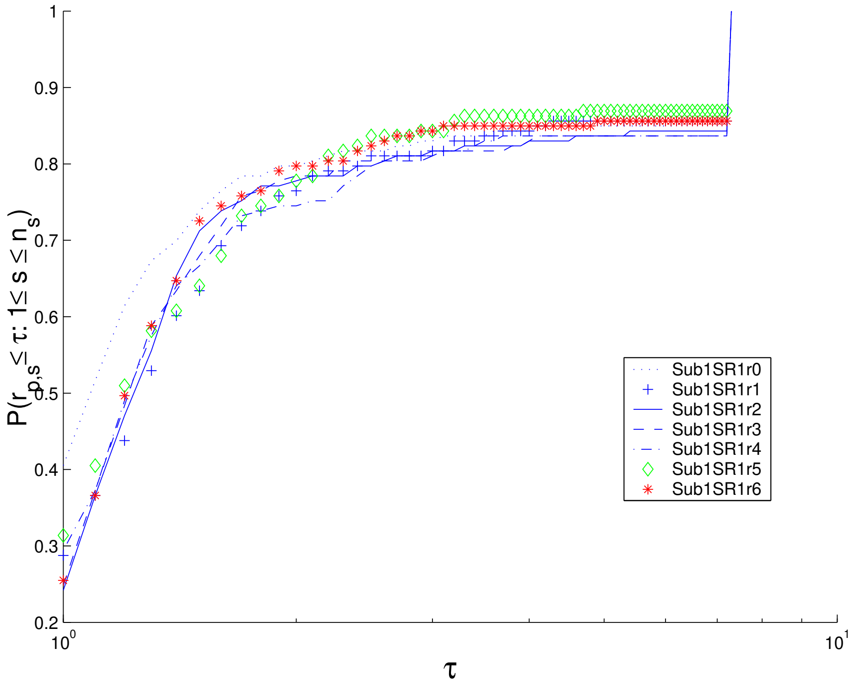
one on most problems. In Figure 2.6, we can see that when $\tau > 20$, the differences among the six solvers become very small, which means that the reliabilities of these solvers are almost the same.

When the SR1 formula is used, the numerical results are compared in Figure 2.7 – Figure 2.12. In Figure 2.7 and Figure 2.8, the 7 solvers “Sub1SR1r0–r6” are compared. We can see that the performances of the algorithms with the reinitialization parameters σ_k^{r1}–σ_k^{r6} cannot be favourably compared with the algorithm without reinitialization (i.e. $\sigma_k = \sigma_k^{r0}$) on about 65% problems of this test set, which is similar with the case where the BFGS formula is used. Except σ_k^{r0}, σ_k^{r5} and σ_k^{r6} are the first two most preferred reinitialization parameters among σ_k^{r1}–σ_k^{r6}. The performance of solvers with the reinitialization parameter σ_k^{r1} is almost the same as that with the reinitialization parameter σ_k^{r5}. Next the 7 solvers “Sub2SR1r0–r6” are compared in Figure 2.9 and Figure 2.10, which are implementations of Algorithm 2.3.3 with different reinitialization parameters defined in (2.3.15). When the number of objective function evaluations and the number of gradient evaluations are used as the performance metrics, by Figure 2.9 we can see that σ_k^{r0}, σ_k^{r3} and σ_k^{r6} are the first three most preferred reinitialization parameters; When CPU time is used as the performance metric, by Figure 2.10 we can see that the reinitialization parameters σ_k^{r0}, σ_k^{r1} and σ_k^{r5} are the first three most preferred ones, and the performance of the solver with the reinitialization parameter σ_k^{r1} is almost the same as that with reinitialization parameter σ_k^{r5}. Figure 2.10 is not very uniform with Figure 2.9, because the CPU time needed to solve the subproblem is difficult to predict when the matrix B_k is not positive definite. Then we compared the six solvers “Sub1SR1r0”, “Sub1SR1r5”, “Sub1SR1r6”, “Sub2SR1r0”, “Sub2SR1r5” and “Sub2SR1r6” in Figure 2.11 and Figure 2.12. We can see that when the SR1 formula is used, the framework of Algorithm 2.3.3 does not work, and solver “Sub1SR1r0” is the most preferred one on most problems. Solver “Sub1SR1r5” is the most reliable one. When the SR1 formula is used, how to adopt lingering technique needs further studies.

Now the six solvers “Sub1BFGSr0”, “Sub1BFGSr4”, “Sub2BFGSr6”, “Sub1-SR1r0”, “TrBFGS” and “TrSR1” are compared in Figure 2.13 and Figure 2.14. In

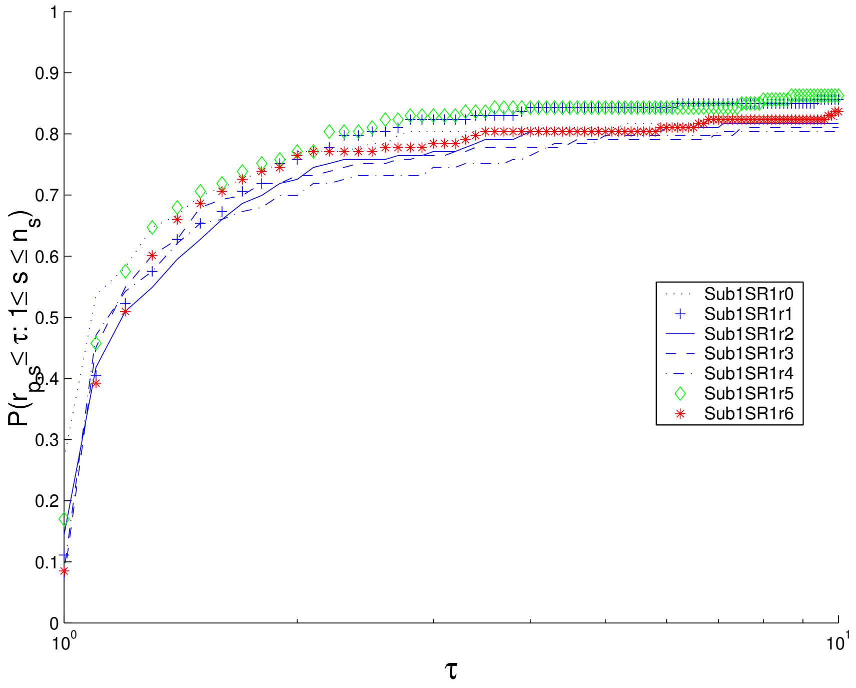


(a)

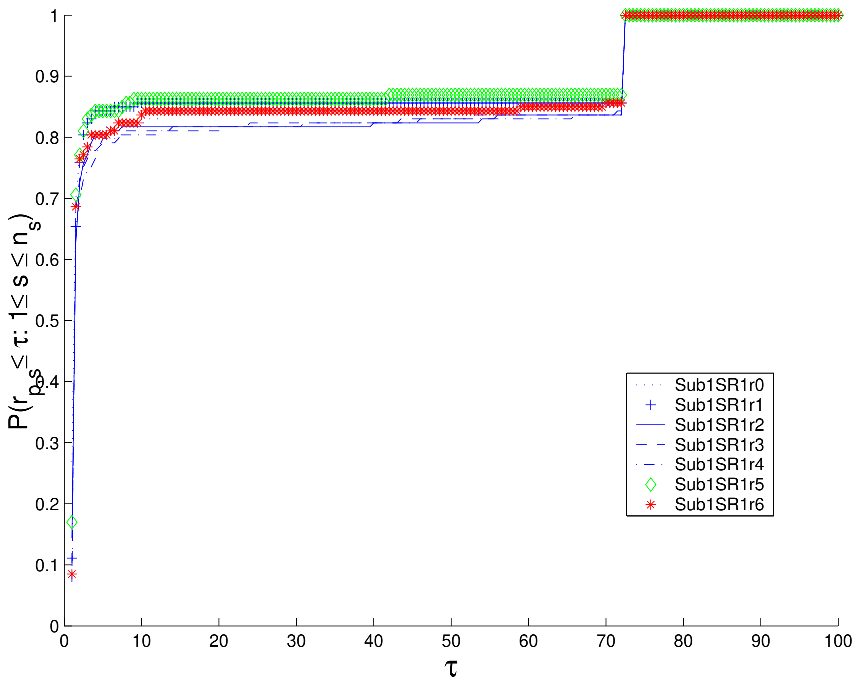


(b)

图 2.7: Performance profile on [0, 10] for the number of (a) function evaluations and (b) gradient evalutions

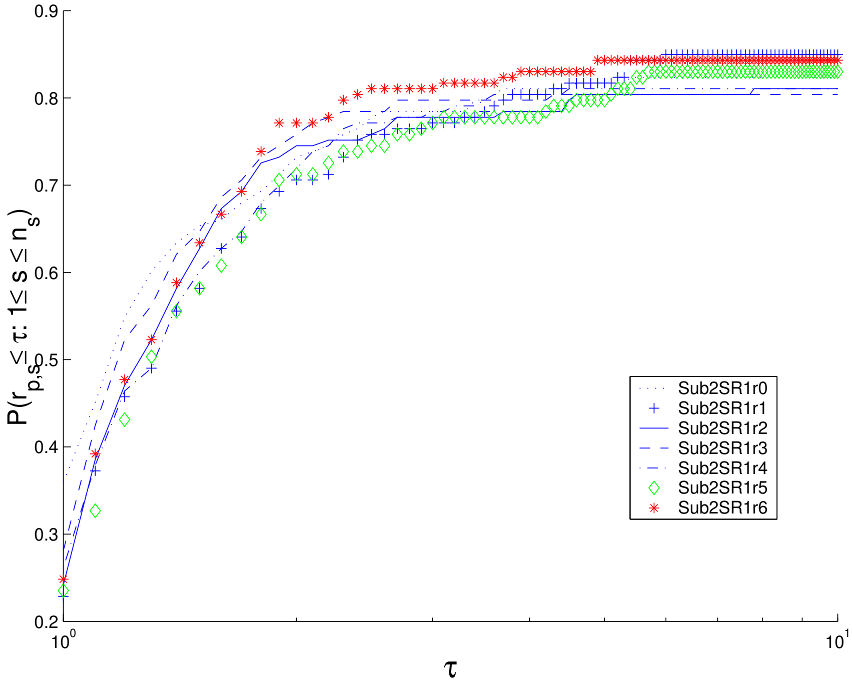


(a)

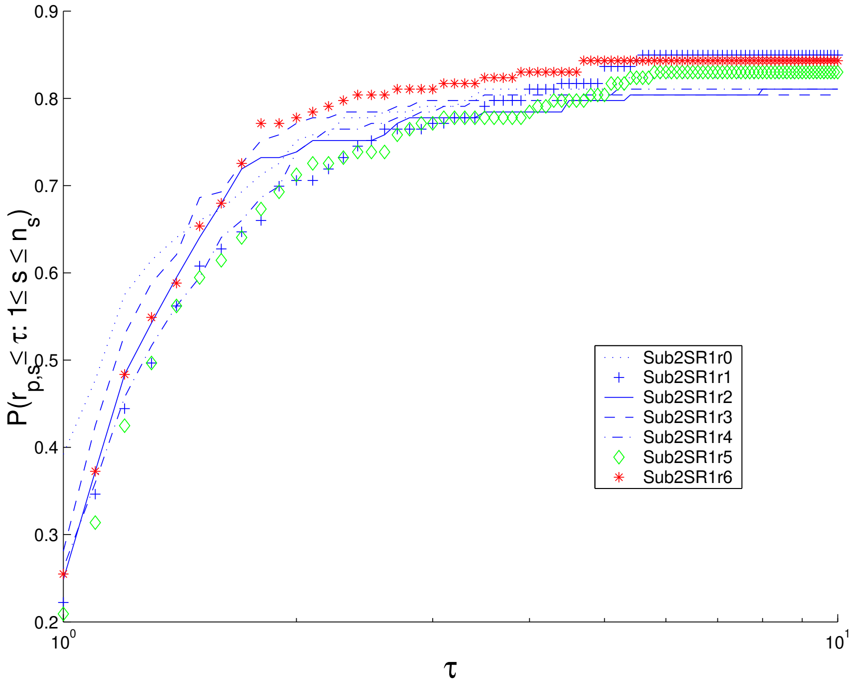


(b)

图 2.8: Performance profile for CPU time on intervals (a)[0, 10] (b)[0,100]

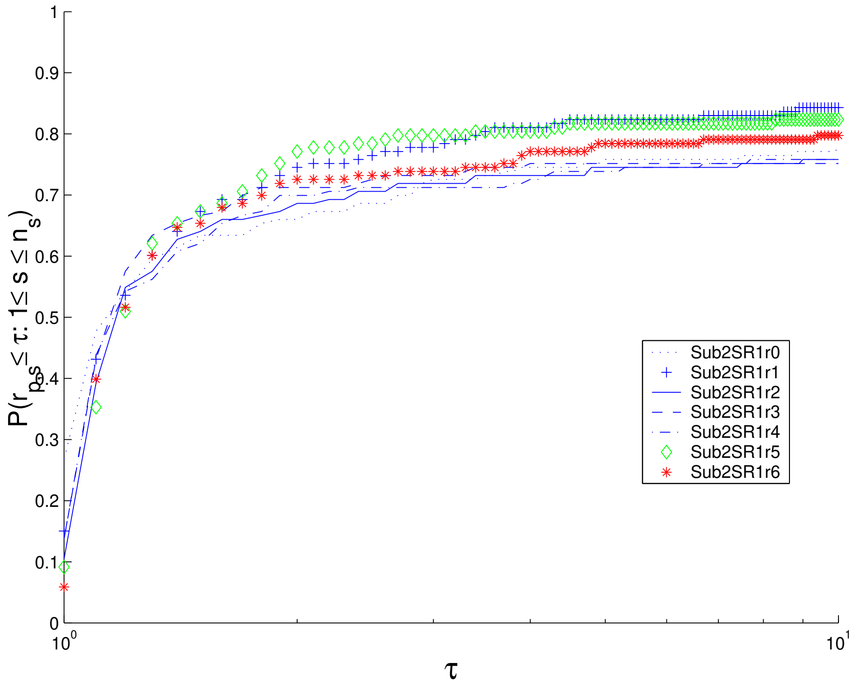


(a)

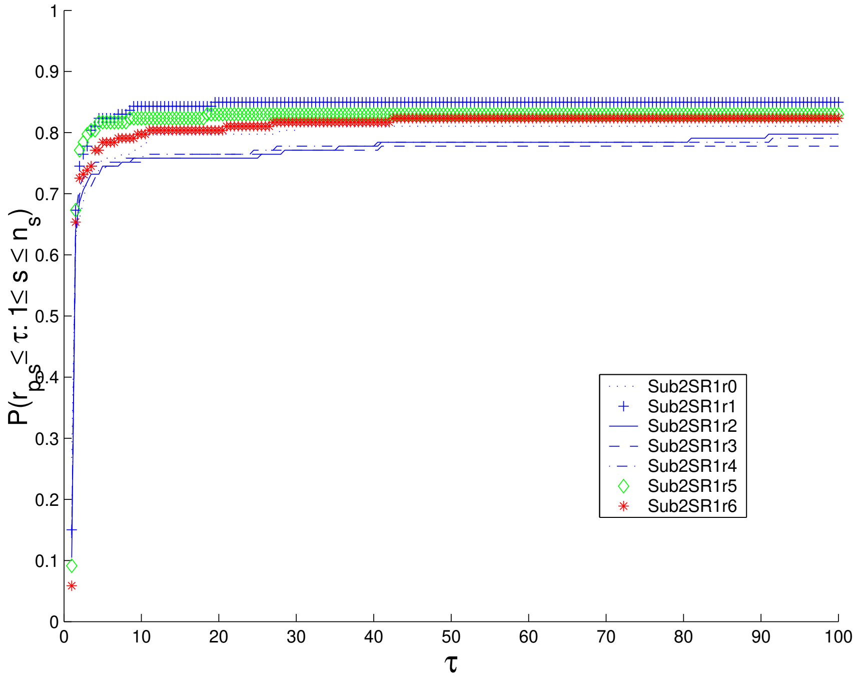


(b)

图 2.9: Performance profile on [0, 10] for the number of (a) function evaluations and (b) gradient evalutions

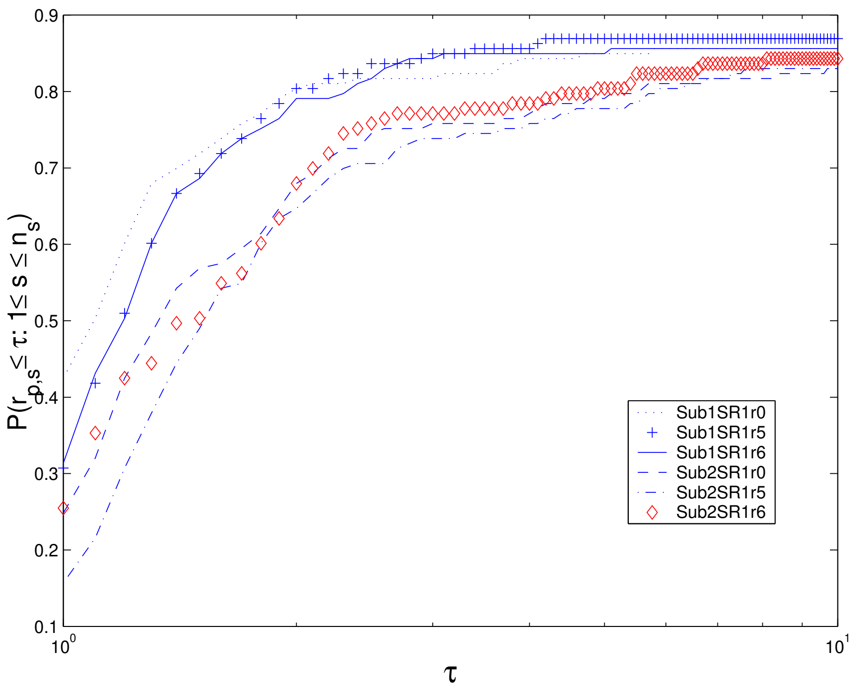


(a)

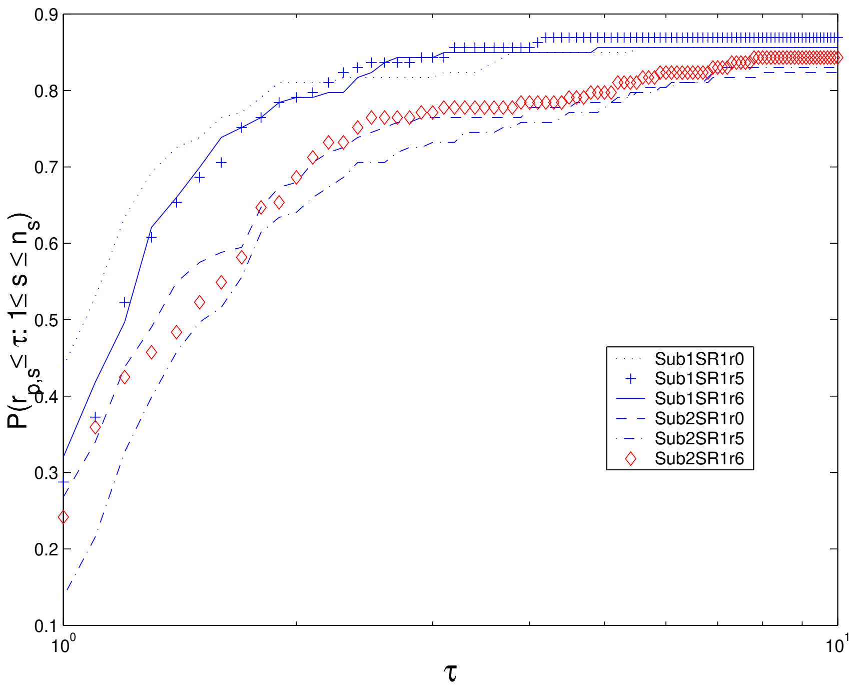


(b)

图 2.10: Performance profile for CPU time on intervals (a)[0, 10] (b)[0,100]

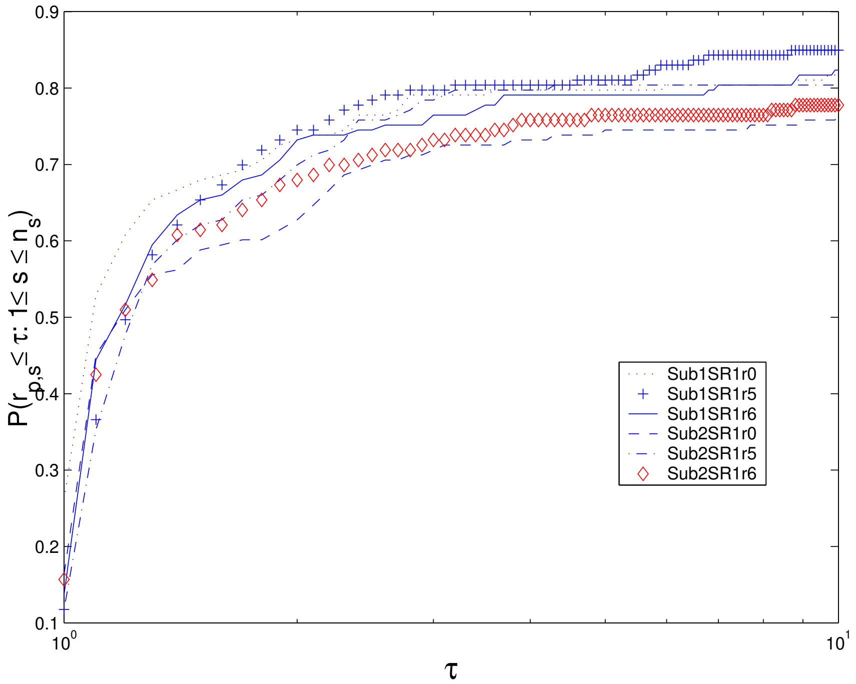


(a)

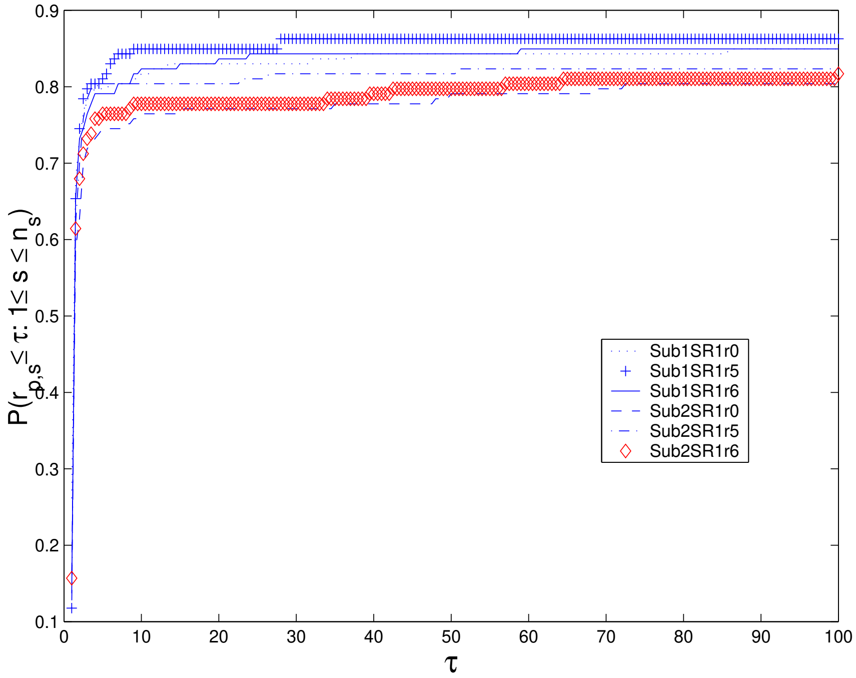


(b)

图 2.11: Performance profile on [0, 10] for the number of (a) function evaluations and (b) gradient evalutions

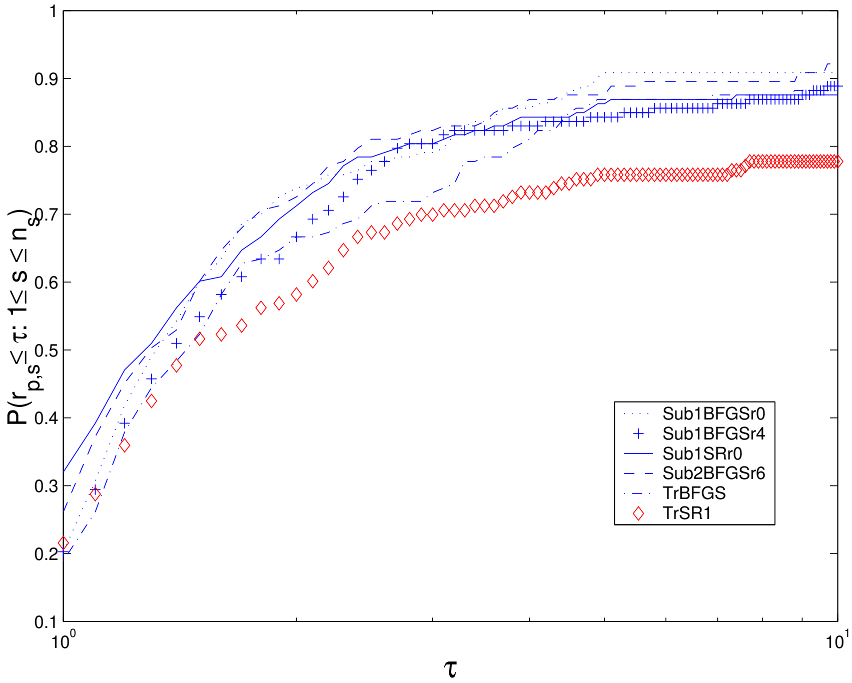


(a)

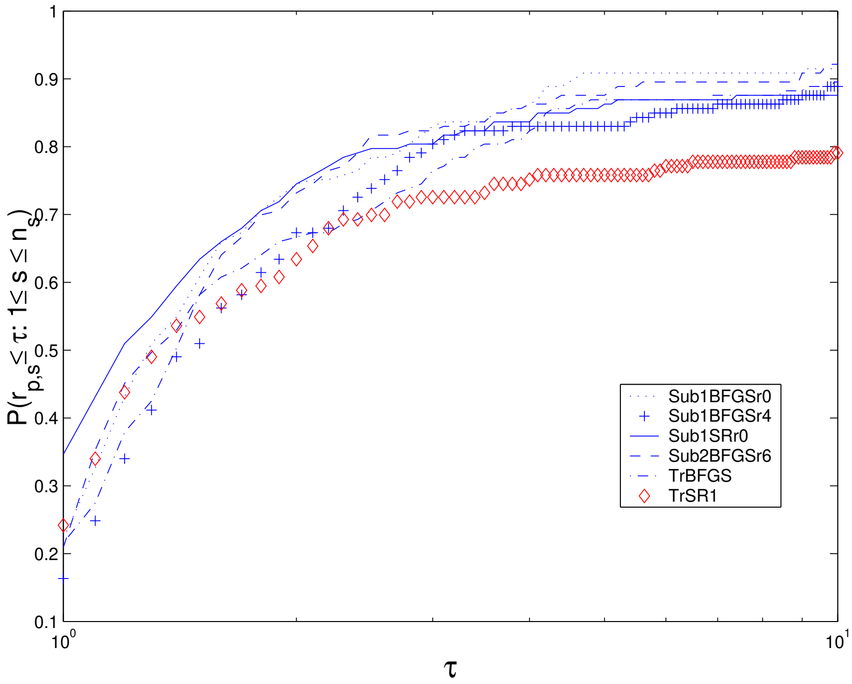


(b)

图 2.12: Performance profile for CPU time on intervals (a)[0, 10] (b)[0,100]

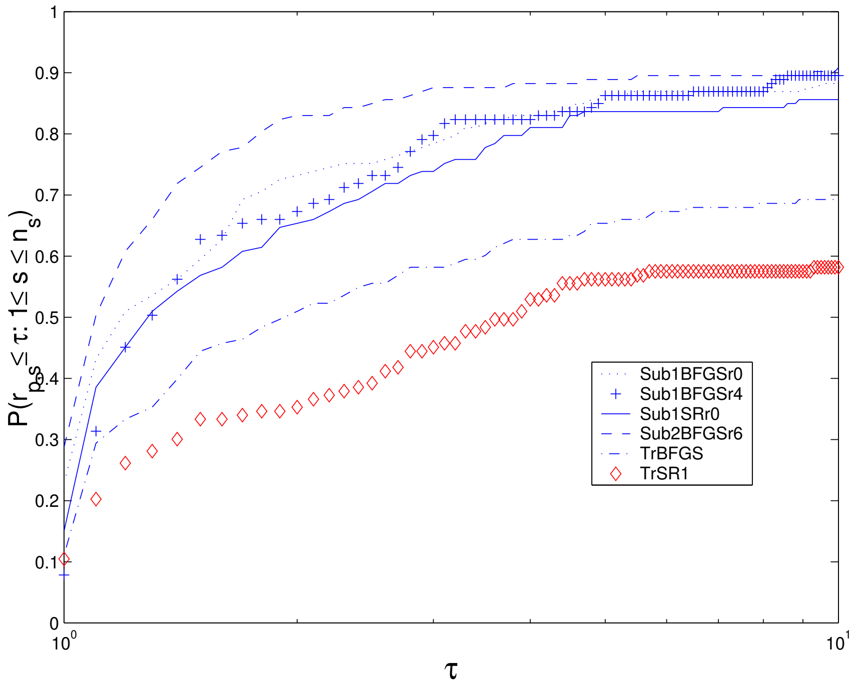


(a)

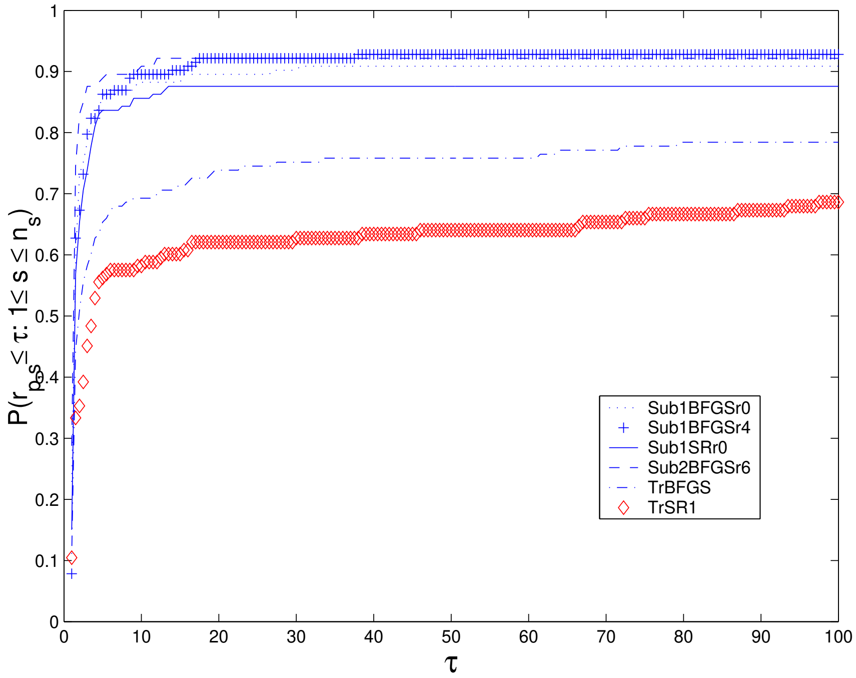


(b)

图 2.13: Performance profile on [0, 10] for the number of (a) function evaluations and (b) gradient evalutions



(a)



(b)

图 2.14: Performance profile for CPU time on intervals (a)[0, 10] (b)[0,100]

Figure 2.13, the number of objective function evaluations and the number of gradient evaluations are used as the performance metrics. We can see that the solver "Sub1SR1r0" is the most preferred one on about 60% problems, and "TrSR1" outperforms "TrBFGS" on about 50% problems. When CPU time is used as the performance metric, we can see from Figure 2.14 that the solver "Sub2BFGSr6" becomes the most preferred one. "Sub1BFGSr0" outperforms "Sub1SR1r0" and "TrBFGS" outperforms "TrSR1" now. We can learn from it that when the SR1 formula is used, the CPU time needed to solve the quadratic subproblem will cost much more than that when the BFGS formula is used, because the update matrix obtained by the SR1 formula perhaps is not positive definite. From the two figures, we can see that the solvers with a subspace implementation outperforms the two solvers "TrBFGS" and "TrSR1" which are implemented in the full space.

§2.4.2 Numerical results on problems with more variables.

Now we will test these solvers on problems with more variables. By Figures 2.1–2.14, we can see that the performance of solvers with the reinitialization parameter σ_k^{r5} is almost the same as that with the reinitialization parameter σ_k^{r1}. So the solvers So the solvers with the reinitialization parameter σ_k^{r5} are not tested in order to save CPU time. For each solver, we limited the CPU time to solve any problem as half an hour (i.e. 1800 seconds). If the CPU time needed to solve some problem is more than half an hour, the solver is stopped and we think the solver failed to solve this problem. After excluding problems none of the solvers can solve in half an hour, the following 73 problems are selected. Denote n as the number of variables. There are 10 problems with $100 \leq n \leq 500$: *arglina(200), arglinb(200), arglinc(200), brownal(200), fletchcr(100), mancino(100), penalty3(200), sensors(100), vardim(200),genrose(500)*; 44 problems with the $1000 \leq n \leq 3000$: *broydn7d(1000), chainwoo(1000), cosine(1000), curly10(1000), curly20(1000), curly30-(1000), dixmaana(3000), dixmaanb(3000), dixmaanc(3000), dixmaand(3000), dixmaane(3000), dixmaanf(3000), dixmaang(3000), dixmaanh(3000), dixmaani(1500), dixmaanj(3000), dixmaank(1500),dixmaanl(1500), dixon3dq(1000), edensch(2000), eigenals(2550), eigenbls(2550),eigencls(2652) , extrosnb(1000), fletcbv2(1000),fletcbv3(1000), modbeale(2000), ncb20b(1000), noncvxu2(1000), noncvxun(1000), nondquar(1000), non-*

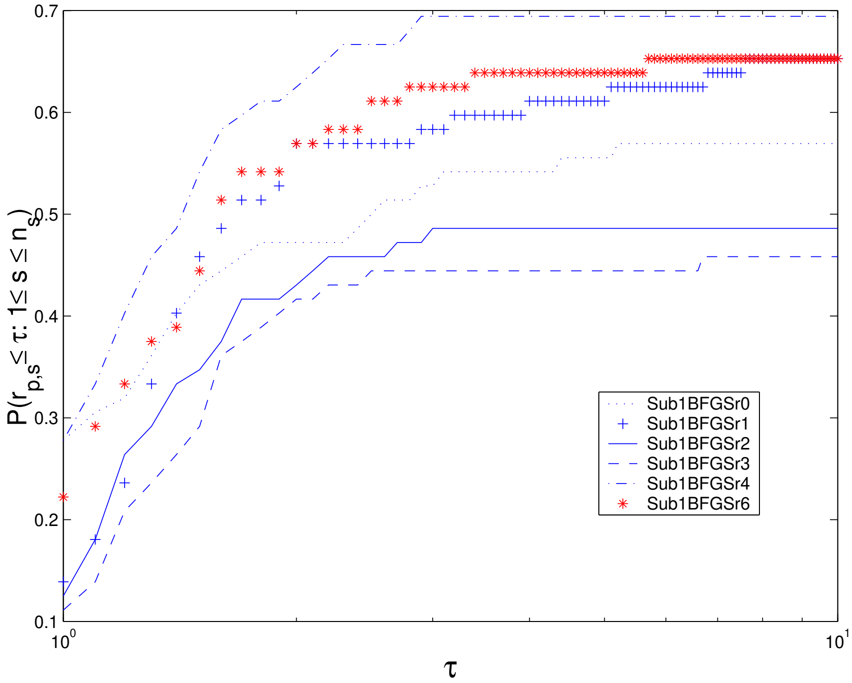
msqrt(1024), penalty1(1000), penalty2(1000), power(1000), sbrybnd(1000), scosine(1000), scurly10(1000), scurly20(1000), scurly30(1000), sparsqur(1000), testquad(1000), tridia(1000), woods(4000),
and 19 problems with $n \geq 5000$: *arwhead(5000), brybnd(5000), bdqrtic(5000), cragglvy(5000), dqdrtic(5000), engval1(5000), fminsrf2(5625), freuroth(5000), liarwhd(5000), morebv(5000), ncb20(5010), nondia(5000), powellsg(5000), schmvett(5000), sinquad(5000), sparsine(5000), srosenbr(5000), tointgss(5000), tquartic(5000).*

On any problem listed above, every solver terminated if $\|g_k\|_2 < 10^{-5}$ or $\|g_k\|_\infty < 10^{-6}$. In Table 2.2, we listed problems on which none of the solvers can obtain a solution satisfying $\|g_k\|_2 < 10^{-5}$ or $\|g_k\|_\infty < 10^{-6}$, and we think all solvers failed to solve these problems.

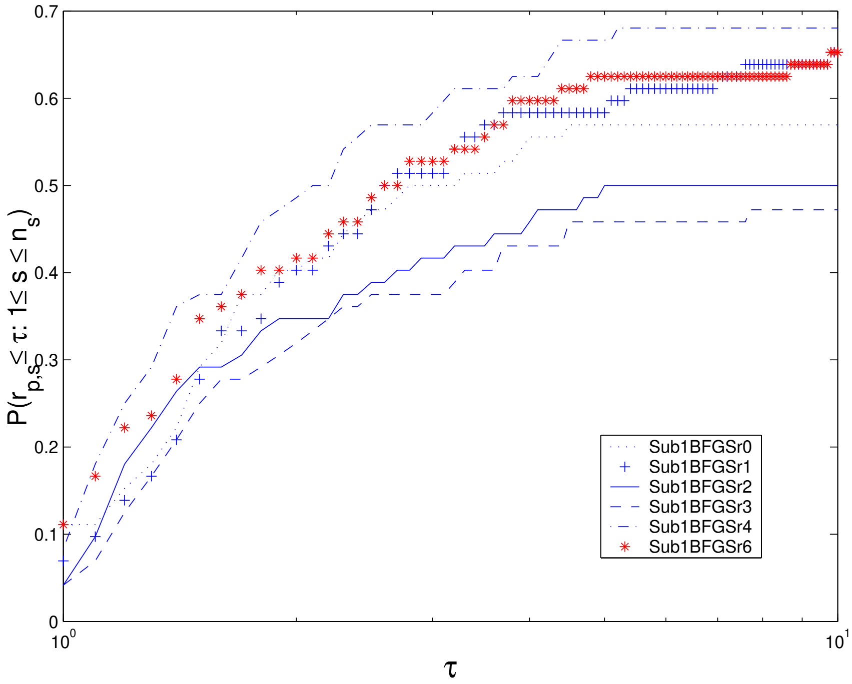
表 2.2: Hard Problems to Solve

Name(Dim)	Func.Val.	$\|g\|_\infty$
EIGENBLS(2550)	0.527474	0.2121526
EIGENCLS(2652)	399.6872	30.99925
FLETCBV3(1000)	-2.581354e+11	2.50299
FREUROTH(5000)	608006.2	0.004854594
NONMSQRT(1024)	92.33829	0.01946075
PENALTY2(1000)	1.446399e+83	2.050731e+38
PENALTY3(200)	0.0009989562	0.08085646
SBRYBND(1000)	3160.284	229170.1
SCOSINE(1000)	-979.0675	4004.883
SCURLY10(1000)	-71435.48	1013400
SCURLY20(1000)	-77530.27	95948.84
SCURLY30(1000)	-82023.76	3204406
SPARSINE(5000)	0.04806616	0.174282

When the BFGS formula is used, the implementations of Algorithm 2.3.1 and Algorithm 2.3.3 with different reinitialization parameters are compared in Figure 2.15–2.20. From these figures, we can see that for both Algorithm 2.3.1 and Algorithm 2.3.3, the most preferred reinitialization parameter is σ_k^{r4}, which is the same as that obtained by [24] and [17]. When the number of objective function

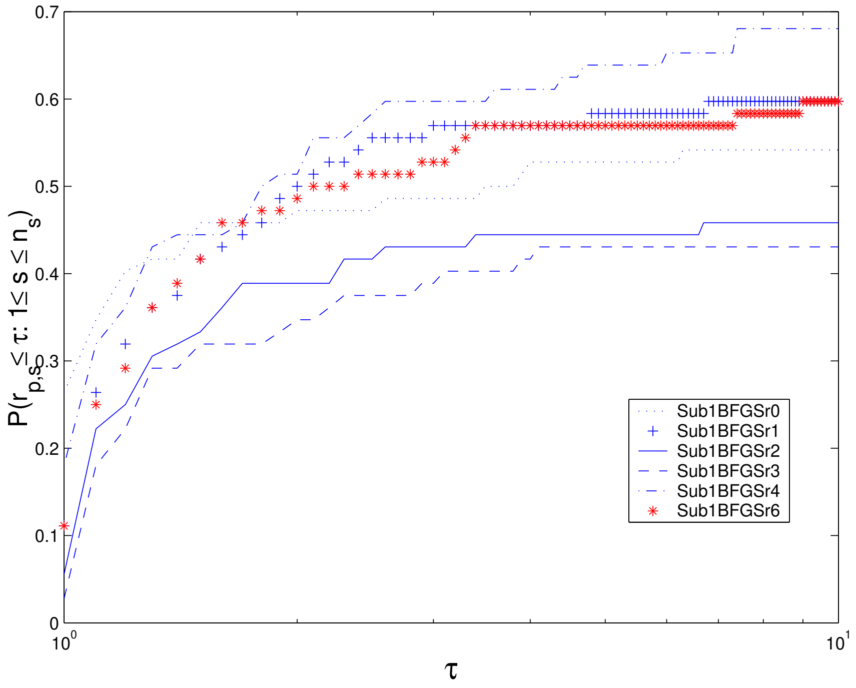


(a)

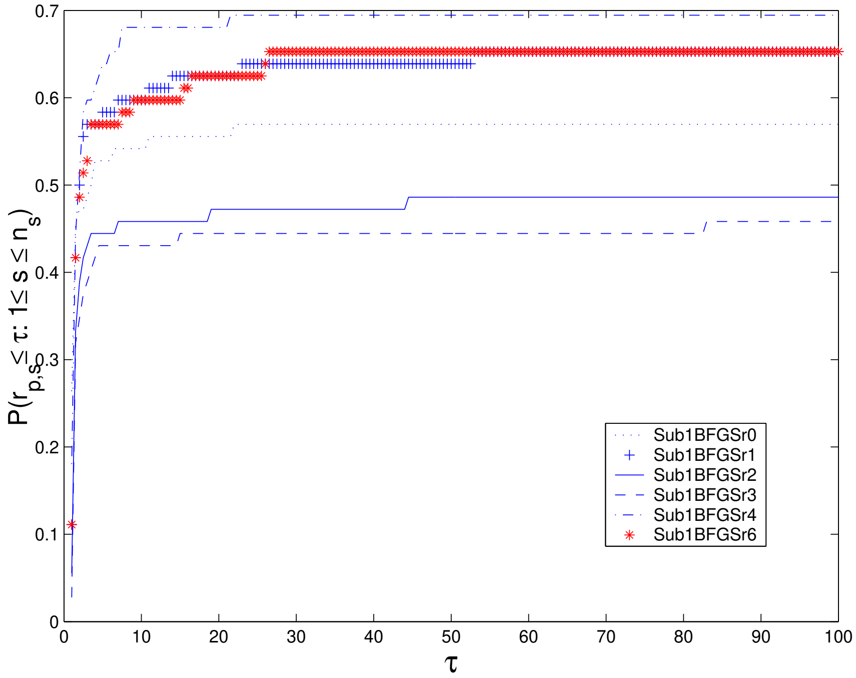


(b)

图 2.15: Performance profile on [0, 10] for the number of (a) function evaluations and (b) gradient evalutions

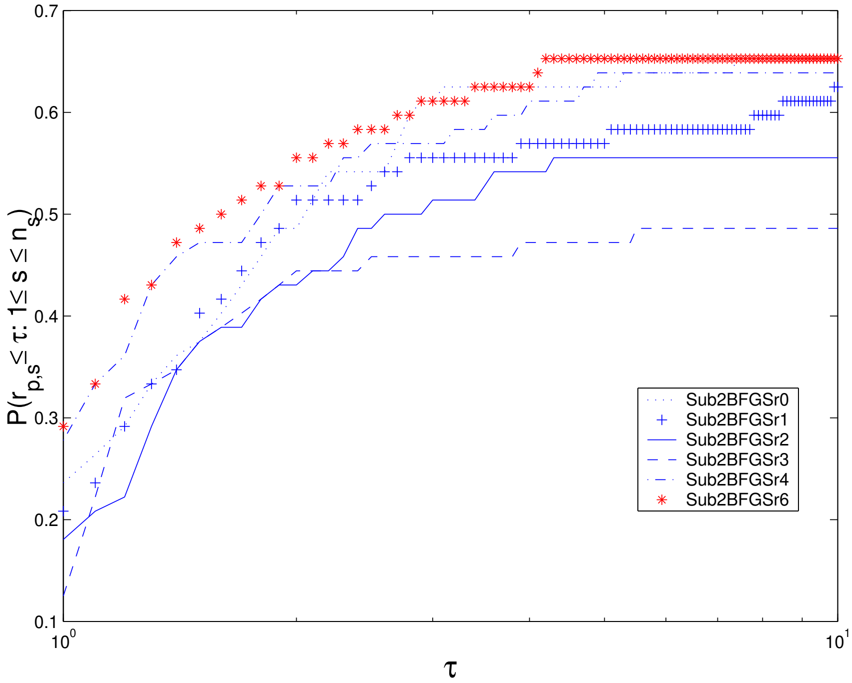


(a)

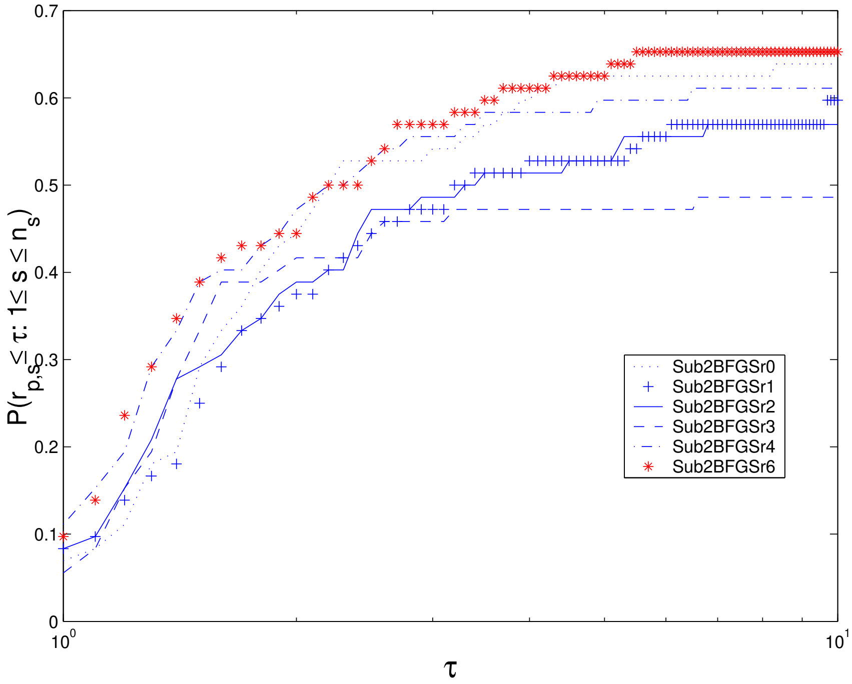


(b)

图 2.16: Performance profile for CPU time on intervals (a)[0, 10] (b)[0,100]

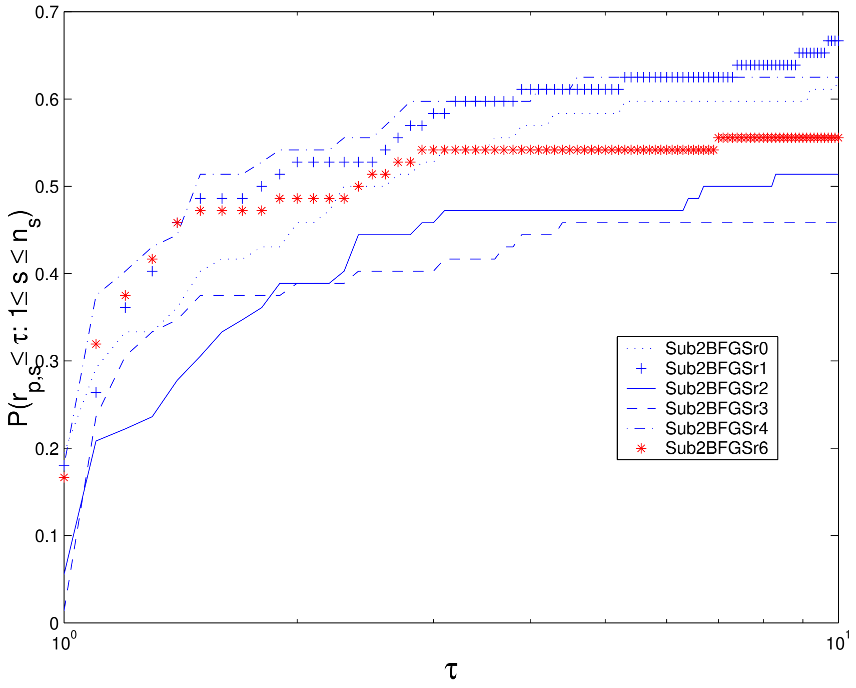


(a)

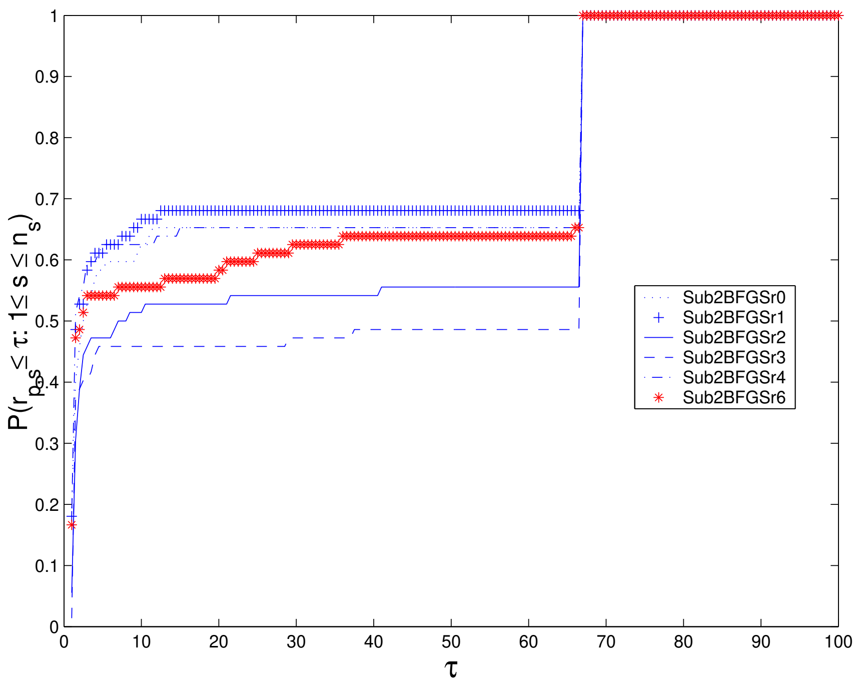


(b)

图 2.17: Performance profile on [0, 10] for the number of (a) function evaluations and (b) gradient evalutions

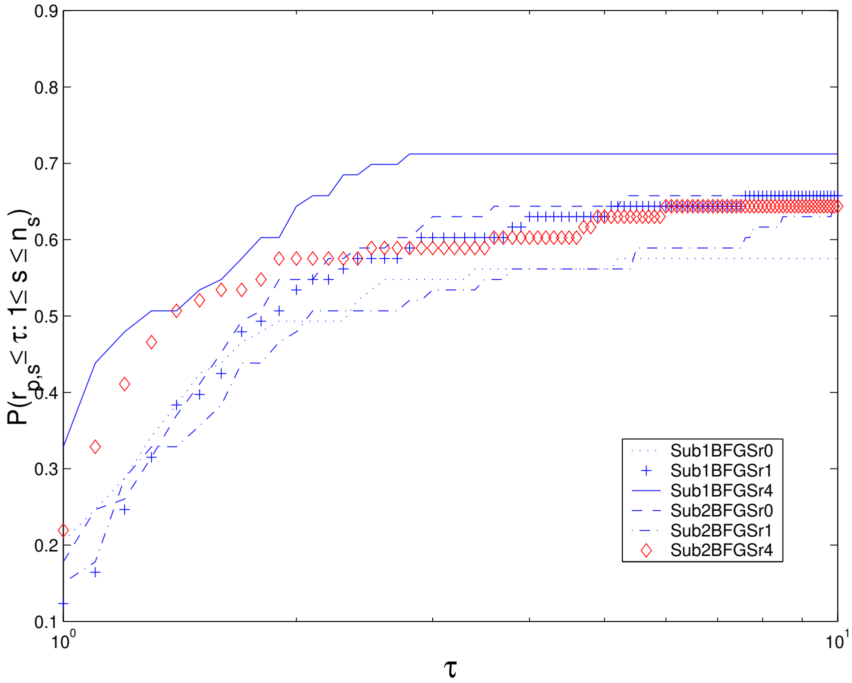


(a)

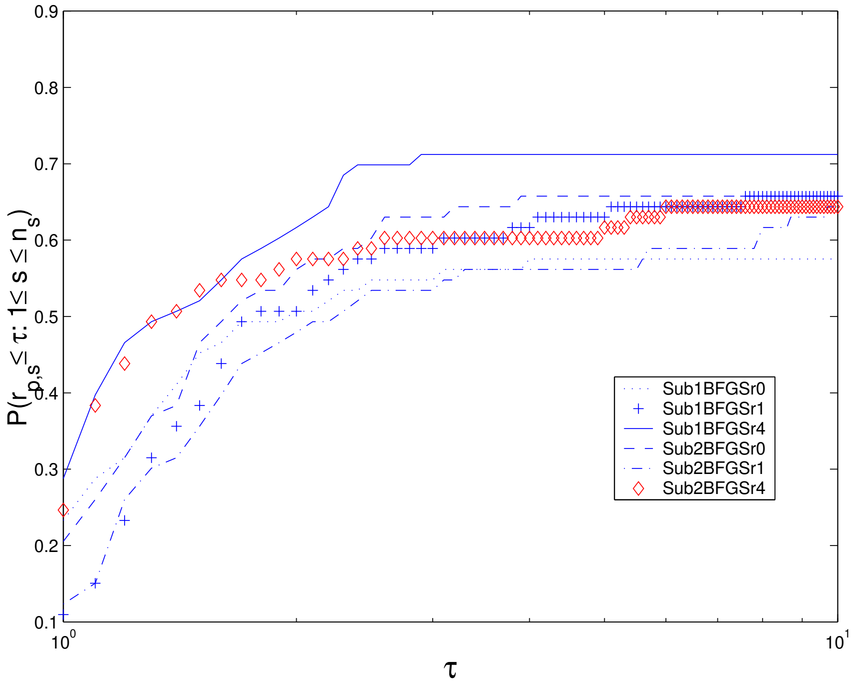


(b)

图 2.18: Performance profile for CPU time on intervals (a)[0, 10] (b)[0,100]

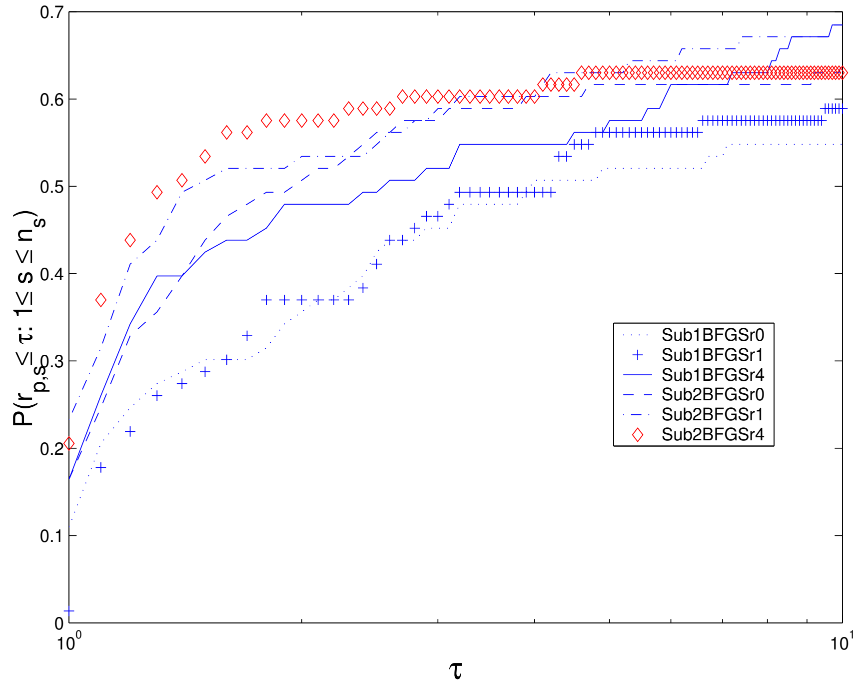


(a)

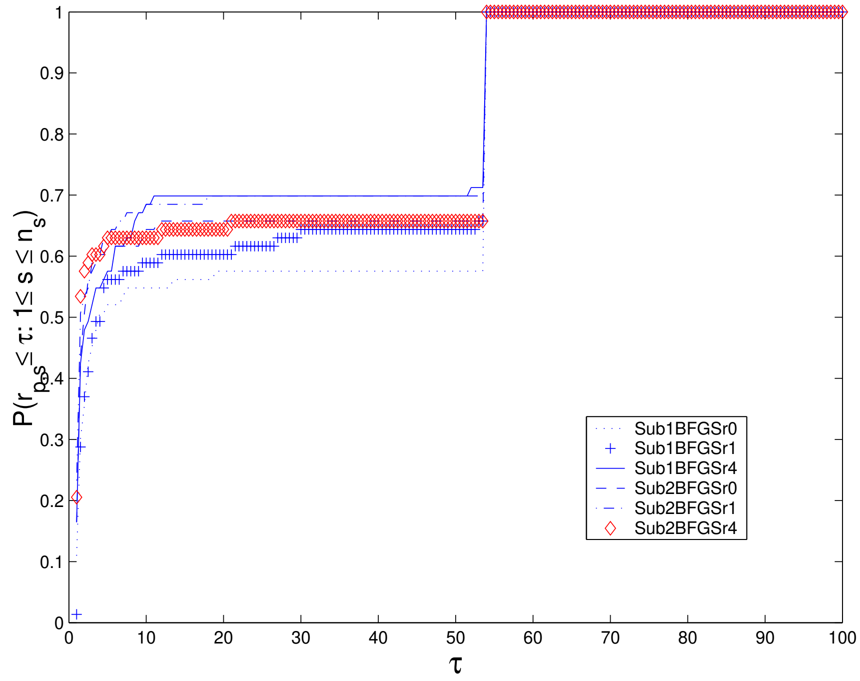


(b)

图 2.19: Performance profile on [0, 10] for the number of (a) function evaluations and (b) gradient evalutions



(a)



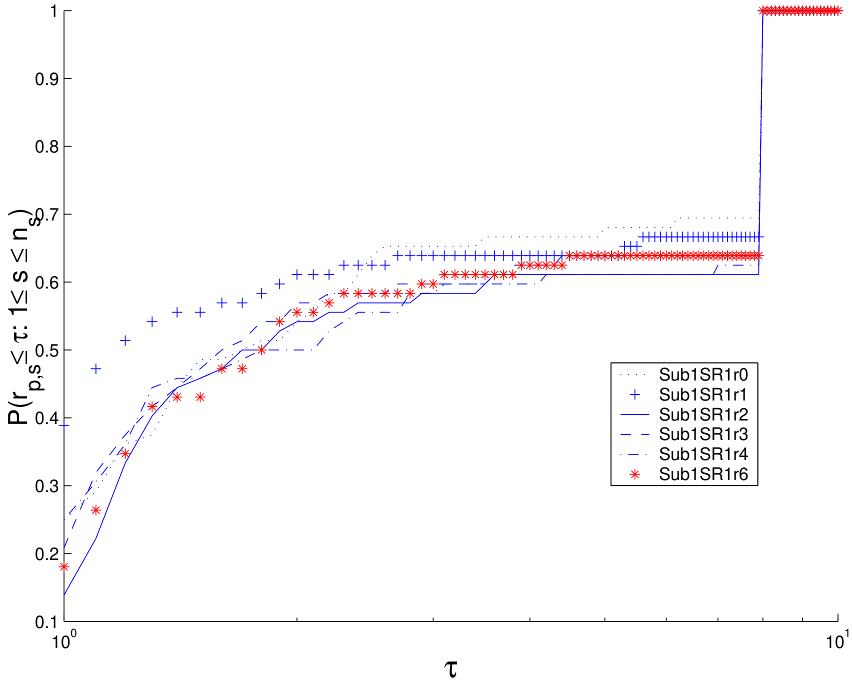
(b)

图 2.20: Performance profile for CPU time on intervals (a)[0, 10] (b)[0,100]

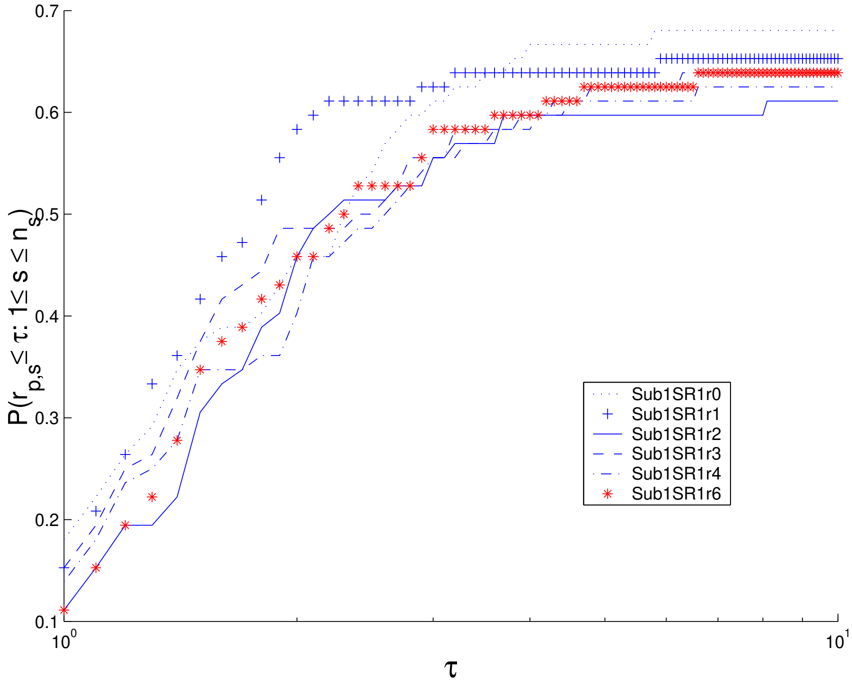
evaluations and the number of gradient evaluations are used as the performance metrics, we can see from Figure 2.15 that the solver "Sub1BFGSr4" outperforms "Sub2BFGSr4" and is the most preferred one. When CPU time is used as the metric, we can see that the solver "Sub2BFGSr4" becomes the most preferred one, and "Sub2BFGSr1" is the second one. This tells us that we gained much by using the lingering technique, which makes the dimension of subspaces increase slower.

When the SR1 formula is used, the numerical results for Algorithm 2.3.1 and Algorithm 2.3.3 with different reinitialization parameters defined in (2.3.15) are compared in Figure 2.21 – Figure 2.26. From Figure 2.21–Figure 2.24, we can see that the reinitialization parameter σ_k^{r1} is the most preferred one for both Algorithm 2.3.1 and Algorithm 2.3.3. From 2.25 (where the number of objective function evaluations and the number of gradient evaluations are used as the performance metrics), we can see that the solver "Sub1SR1r1", which is an implementation of Algorithm 2.3.1 with the reinitialization parameter σ_k^{r1}, is the most preferred one. When CPU time is used as the metric, the solver "Sub2SR1r1", which is an implementation of Algorithm 2.3.3 with the reinitialization parameter σ_k^{r1}, becomes the most preferred one, and it outperforms "Sub1SR1r1" on about 50% problems due to the lingering technique.

Finally, we compare the 7 solvers "Sub1BFGSr0", "Sub1BFGSr4", "Sub2BFGSr4", "TrBFGS", "Sub1SR1r0", "Sub1SR1r1" and "TrSR1" in Figure 2.27 and Figure 2.28. When the number of objective function evaluations and the number of gradient evaluations are used as the performance metrics, we can see from Figure 2.27 that the solver "Sub1SR1r1" outperforms "Sub1BFGSr4" on about 60% problems, and it is the most preferred one. "Sub1SR1r0" outperforms "Sub1BFGSr0", and "TrSR1" outperforms "TrBFGS" in Figure 2.27. When CPU time is used as the performance metric, we can see from Figure 2.28 that the solver "Sub2BFGSr4" becomes the most preferred one due to the lingering technique. "Sub1SR1r1" outperforms "Sub1BFGSr4" on about 50% problems, and "Sub1SR1r0" outperforms "Sub1BFGSr0". All the 5 subspace solvers outperform the two solvers "TrBFGS" and "TrSR1" which are implemented in the



(a)



(b)

图 2.21: Performance profile on [0, 10] for the number of (a) function evaluations and (b) gradient evalutions

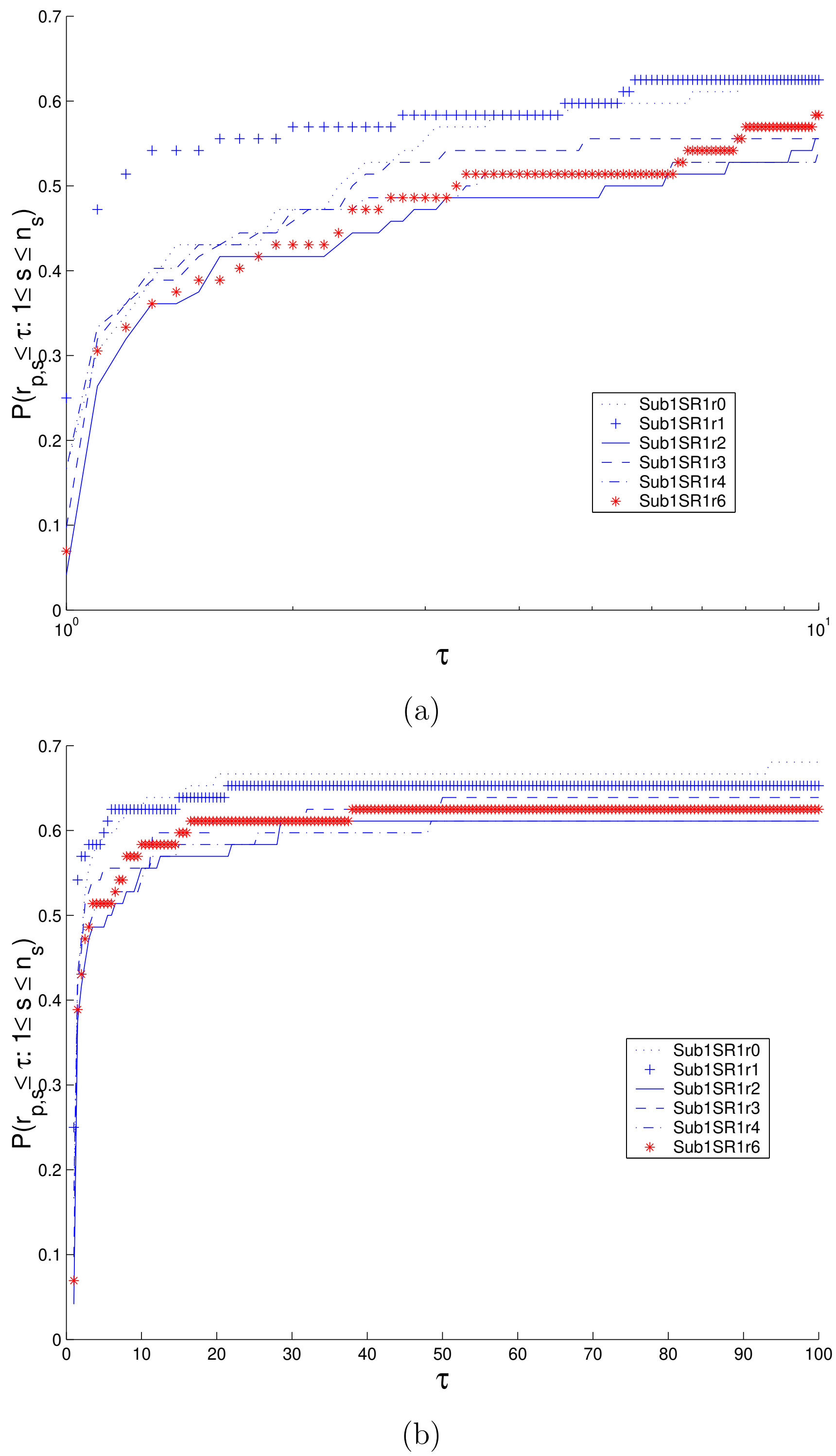
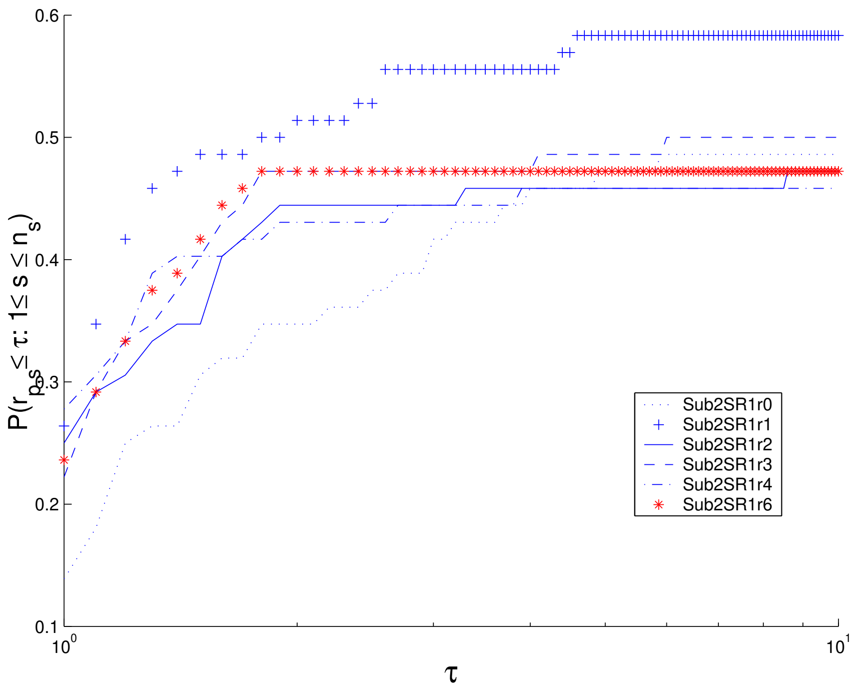
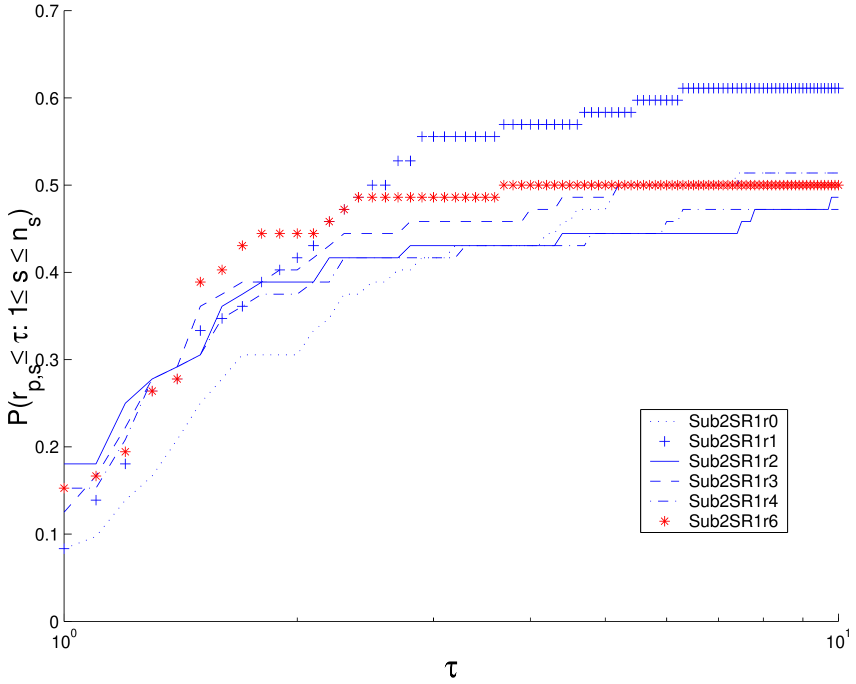


图 2.22: Performance profile for CPU time on intervals (a)[0, 10] (b)[0,100]

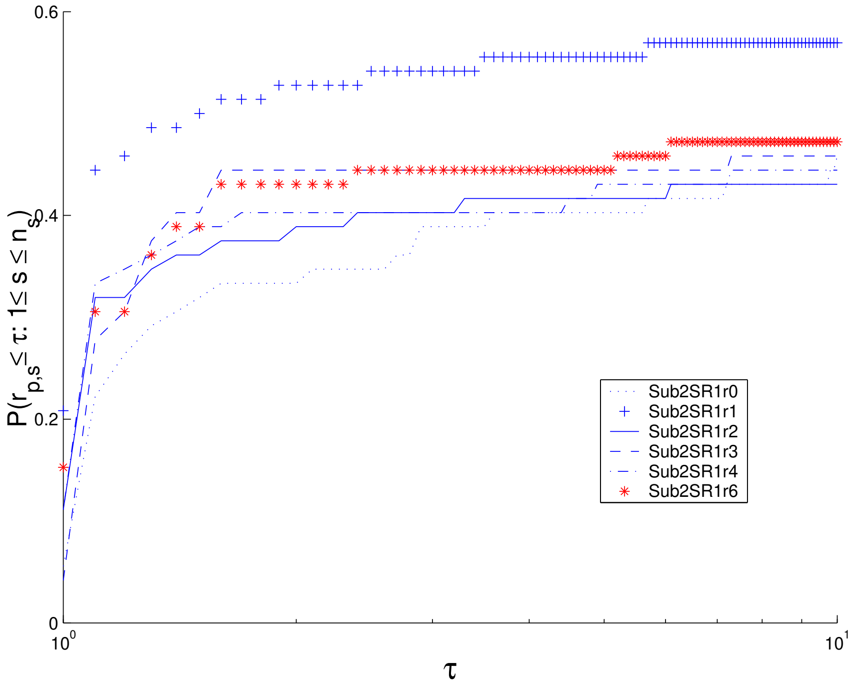


(a)

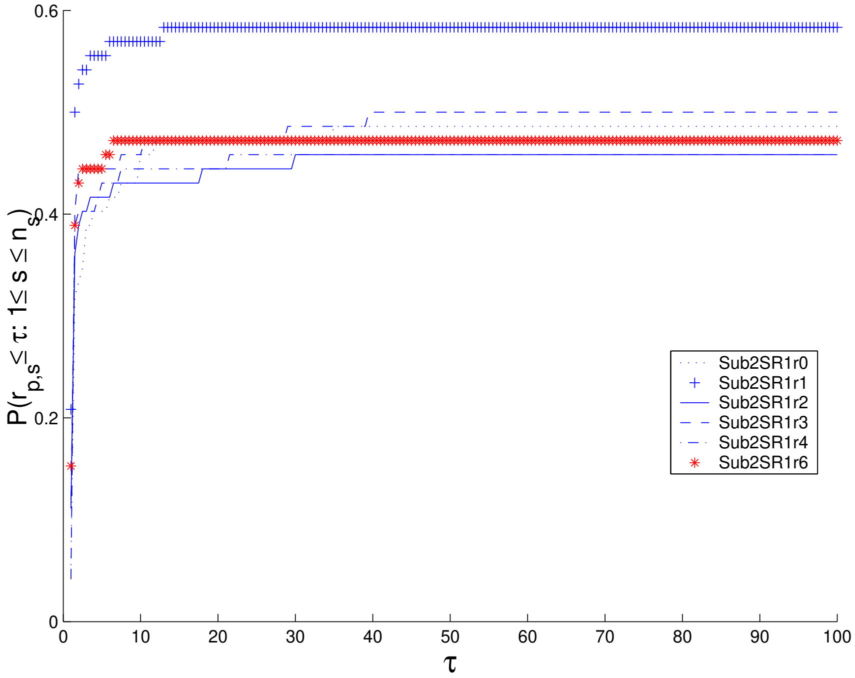


(b)

图 2.23: Performance profile on [0, 10] for the number of (a) function evaluations and (b) gradient evalutions

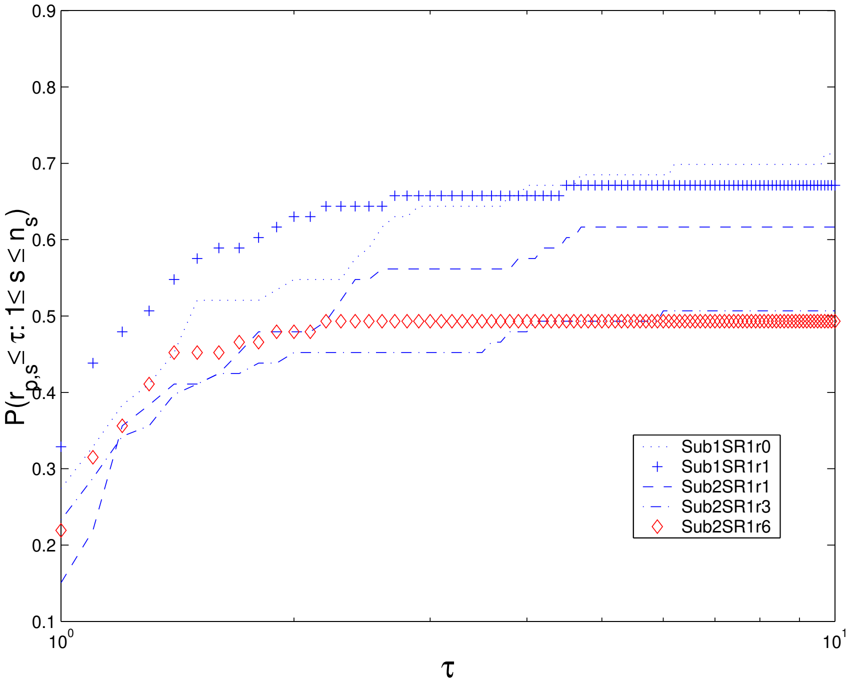


(a)

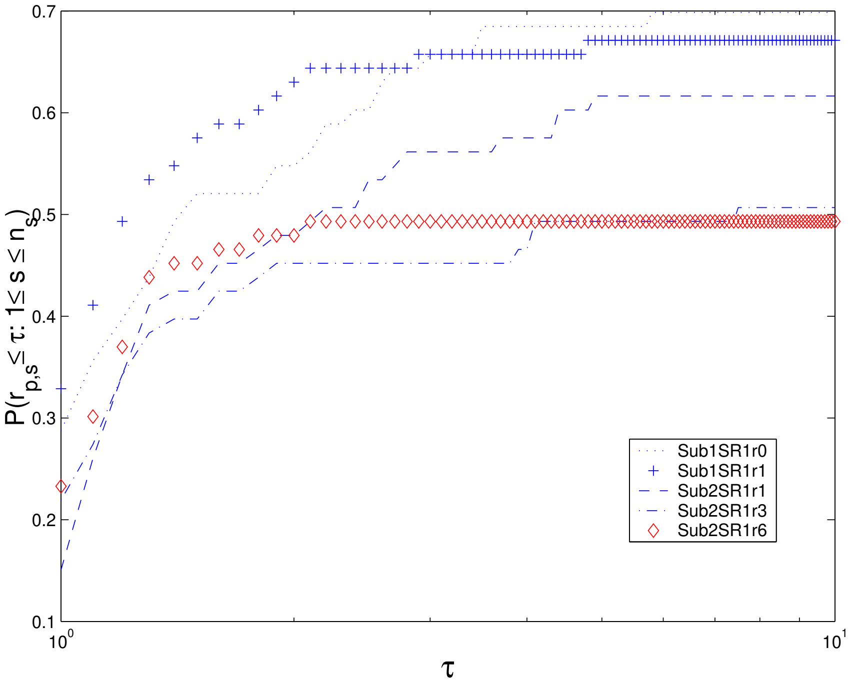


(b)

图 2.24: Performance profile for CPU time on intervals (a)[0, 10] (b)[0,100]

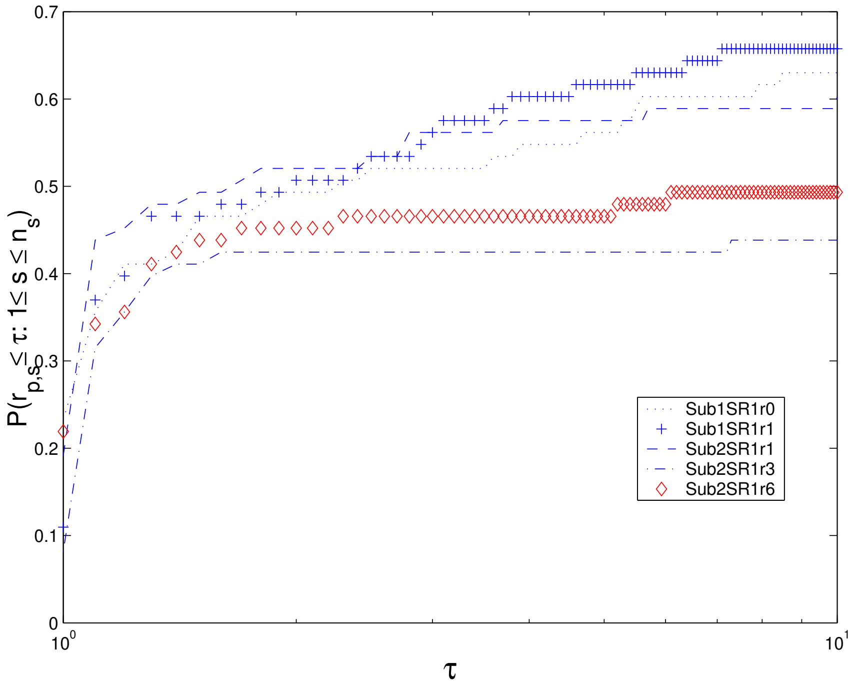


(a)

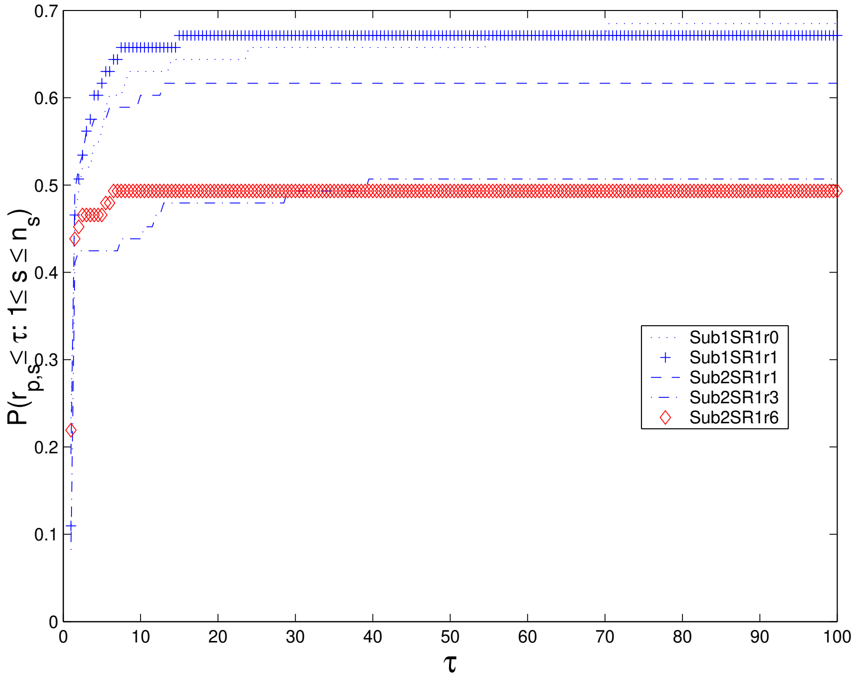


(b)

图 2.25: Performance profile on [0, 10] for the number of (a) function evaluations and (b) gradient evalutions

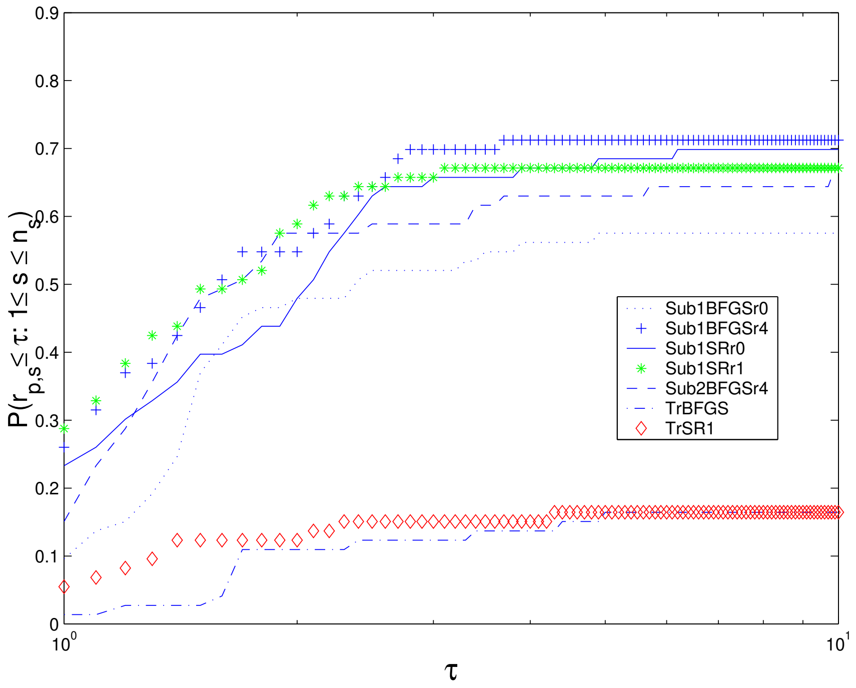


(a)

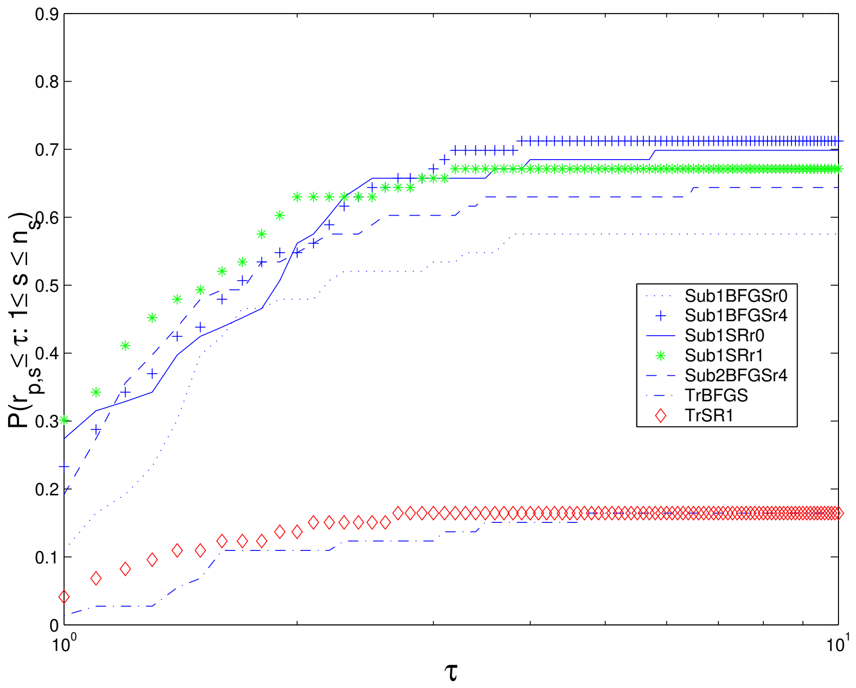


(b)

图 2.26: Performance profile for CPU time on intervals (a)[0, 10] (b)[0,100]

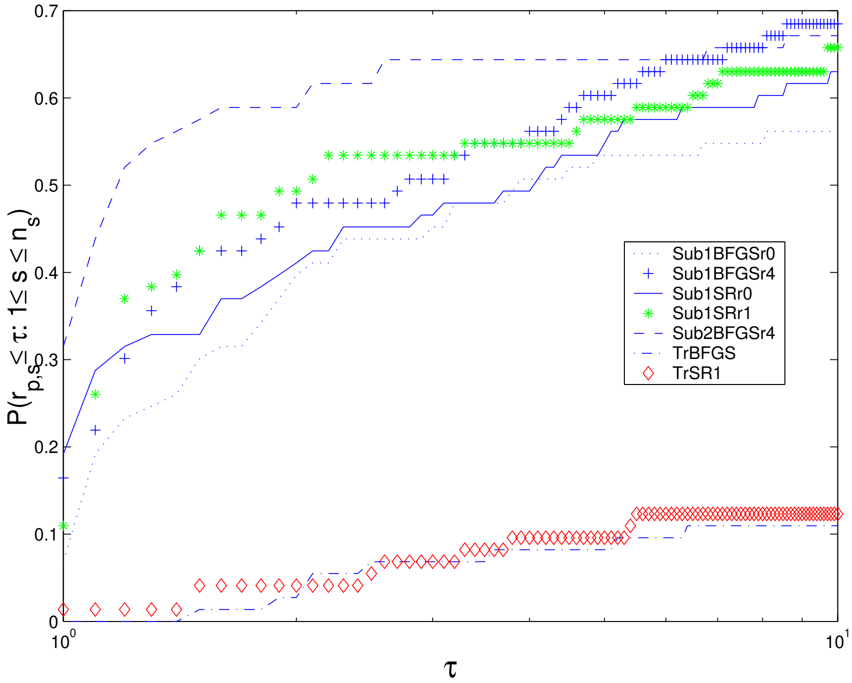


(a)

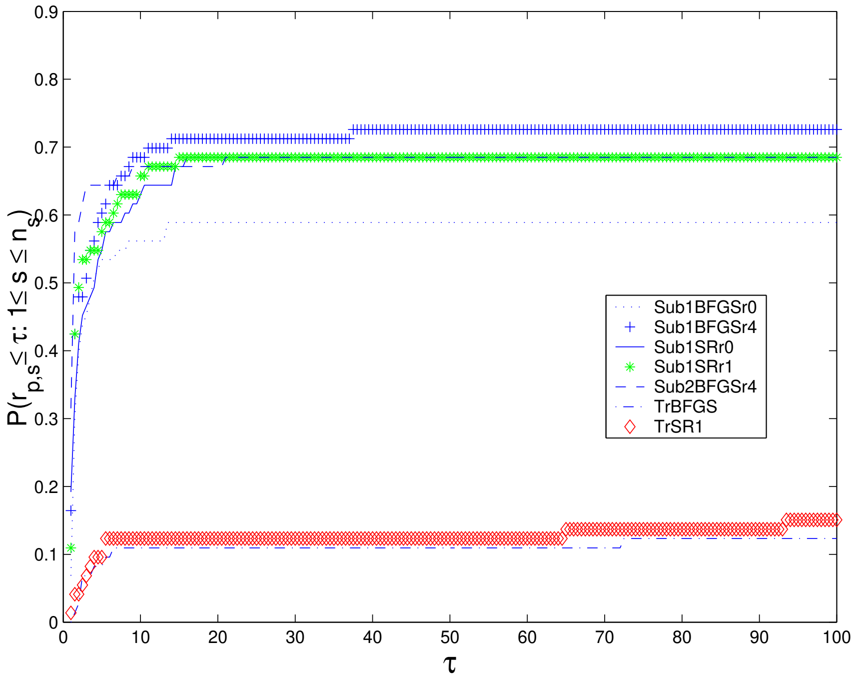


(b)

图 2.27: Performance profile on [0, 10] for the number of (a) function evaluations and (b) gradient evalutions



(a)



(b)

图 2.28: Performance profile for CPU time on intervals (a)[0, 10] (b)[0,100]

whole space. By comparing Figure 2.14 and Figure 2.28, we can see that the subspace solvers will win more over the solvers "TrBFGS" and "TrSR1" when the problems have more variables. For example, "Sub1BFGSr0" solved the problem *arwhead* with $n = 5000$ in 6 iterations, and the CPU time is only 0.69 seconds, but both "TrBFGS" and "TrSR1" failed to solve this problem in half an hour.

From these numerical results, we can conclude that the subspace trust region methods are more efficient in implementation than the conventional trust region methods which are implemented in the whole space.

第三章 Limited Memory Trust Region Methods

Two limited memory trust region algorithm for large scale unconstrained optimization are constructed by using subspace techniques and the compact representations for the BFGS formula and the SR1 formula proposed by [3]. The new methods can be viewed as modifications on the limited memory quasi-Newton method with trust regions. Due to the special structures of the subspace trust region subproblems, the method can be applied to very large scale problems. Convergence properties of the methods are analyzed. Numerical results for some typical large scale examples are reported and it shows that the two new limited memory algorithms which are in the trust region framework are promising.

§3.1 Introduction

Consider the following unconstrained optimization problem:

$$\min_{\boldsymbol{x}\in R^n} f(\boldsymbol{x}) \tag{3.1.1}$$

where n is large and analytic expressions for the object f and its gradient g are available.

Large scale nonlinear optimization problems are very important in scientific and engineering computation, and such problems are normally difficult to solve because of the large number of variables, the limited memory storage, and a large amount of computation at each iteration. Various approaches have been proposed, such as combined CG-QN algorithm in [2], partitioned quasi-Newton method in [21] and the limited memory BFGS(L-BFGS) in [27, 24], etc. . [24, 20] showed that limited memory quasi-Newton methods are effective for solving large scale unconstrained optimization problems.

The purpose of this chapter is to present a new trust region method using subspace technique for large scale unconstrained nonlinear optimization problems. It has been shown that the trust region method has good convergence properties in [35] and [34]. However, when the number of variables is large, the solution of the quadratic subproblem would be very cost. By analysis of the L-BFGS method, we find out that the quadratic subproblem with a trust region could be constructed in a subspace whose dimension is very small. Then the solution of the subproblem would need only a little effort even if the problem is very large.

L-BFGS method is a variation of standard BFGS method. Let x_k be the point at the k-th iteration, $g_k = \nabla f(x_k)$ and $\nabla^2 f(x_k)$ be the gradient and Hessian of f at the point x_k respectively. Denote

$$s_k = x_{k+1} - x_k, \quad y_k = g_{k+1} - g_k.$$

L-BFGS method choose a "basic matrix" γI and update it m times using the BFGS formula to get the current inverse Hessian approximation H_k by storing the most recent m pairs (s_i, y_i), $i = k-m, \cdots, k-1$. Denote

$$S = [s_{k-m}, \cdots, s_{k-1}], \quad Y = [y_{k-m}, \cdots, y_{k-1}].$$

By Theorem 2.2 in [3], the search direction d at the k-th iteration in L-BFGS can be represented as (where the subscript k is omitted for convenience)

$$\begin{aligned} d &= -Hg \\ &= -\left\{\gamma I + [S \quad \gamma Y] \begin{bmatrix} R^{-\mathrm{T}}(D + \gamma Y^{\mathrm{T}} Y) R^{-1} & -R^{-\mathrm{T}} \\ -R^{-1} & 0 \end{bmatrix} \begin{bmatrix} S^{\mathrm{T}} \\ \gamma Y^{\mathrm{T}} \end{bmatrix} \right\} g \\ &= -\gamma g + [S \quad \gamma Y] \begin{bmatrix} z_s \\ z_y \end{bmatrix} = [-g \quad S \quad Y] \begin{bmatrix} \gamma \\ z_s \\ \gamma z_y \end{bmatrix}, \end{aligned} \tag{3.1.2}$$

where

$$\begin{bmatrix} z_s \\ z_y \end{bmatrix} = -\begin{bmatrix} R^{-\mathrm{T}}(D + \gamma Y^{\mathrm{T}} Y) R^{-1} & -R^{-\mathrm{T}} \\ -R^{-1} & 0 \end{bmatrix} \begin{bmatrix} S^{\mathrm{T}} g \\ \gamma Y^{\mathrm{T}} g \end{bmatrix}.$$

By formula (3.1.2), we could see that the search direction obtained by the L-BFGS method belongs to the subspace expanded by $[-g, S, Y]$. When some step in the direction d_k is not satisfactory, the line-search algorithms choose a shorter step length, and the information about the Hessian of f in the subspace is not used. We can not say that the descent direction d_k in (3.1.2) is the best one, and we could expect to get better solution by searching in the subspace spanned by $[-g, S, Y]$ with the trust region techniques.

At the moment, the main idea of limited memory trust region methods is to solve the quadratic subproblem (2.1.2)–(2.1.3) in some economical way, where $B_k = \sigma_k I + A_k W_k^{-1} A_k^{\mathrm{T}}$ is the compact representation of BFGS or SR1 due to [3]. [23] proposed an algorithm under the framework of trust region methods by using the "double dogleg"technique to solve the the quadratic subproblem (2.1.2)–(2.1.3). [5] constructed an algorithm which used the Sherman-Morrison-Woodbury formula to solve the quadratic subproblem due to the compact representations of B_k. [6] proposed an algorithm which used QR decompositon and eigenvalue decomposition to solve the quadratic subproblem. All these algorithms solve the quadratic subproblem in the whole space whose dimension is n (the number of variables). By (3.1.2), we can see that in fact we can solve the quadratic subproblem (2.1.2)–(2.1.3) in the subspace spanned by $[-g, S, Y]$ due to the compact representations proposed by [3]. In sections 3.2 an algorithm without orthogonalization is discussed, and an algorithm with orthogonalized basis for the subspace is studied in sections 3.3.

§3.2 The Algorithm without Orthogonalization

In order to avoid numerical problems, define A_k as

$$A_k = \left[\frac{-g_k}{\|g_k\|}, \frac{s_{k-m}}{\|s_{k-m}\|}, \cdots, \frac{s_{k-1}}{\|s_{k-1}\|}, \frac{y_{k-m}}{\|y_{k-m}\|}, \cdots, \frac{y_{k-1}}{\|y_{k-1}\|}\right]. \tag{3.2.1}$$

So A_k is a $n \times (2m+1)$ matrix. Denote the subspace spanned by the columns of A_k by $Span(A_k)$, and

$$F_k(z) = f(x_k + A_k z). \tag{3.2.2}$$

By Taylor's series theorem,

$$F_k(z) \approx f(x_k) + z^{\mathrm{T}} A_k^{\mathrm{T}} g_k + \frac{1}{2} z^{\mathrm{T}} A_k^{\mathrm{T}} \nabla^2 f(x_k) A_k z. \tag{3.2.3}$$

Now the main problem is how to get an approximation to $A_k^{\mathrm{T}} \nabla^2 f(x_k) A_k$ in the subspace $Span(A_k)$ by the most recently computed m pairs $(s_i, y_i), i = k-m, \cdots, k-1$. We have two ways to do this by L-BFGS updating formula. The first way is to get the approximation B_k to the Hessian $\nabla^2 f(x_k)$ by formula (2.17) in [3],

$$B_k = \sigma_k I - [\sigma_k S_k \quad Y_k] \begin{bmatrix} \sigma_k S_k^{\mathrm{T}} S_k & L_k \\ L_k^{\mathrm{T}} & -D_k \end{bmatrix}^{-1} \begin{bmatrix} \sigma_k S_k^{\mathrm{T}} \\ Y_k^{\mathrm{T}} \end{bmatrix}, \tag{3.2.4}$$

where

$$S_k = \begin{cases} [s_0, \ldots, s_{k-1}] & \text{if } k \le m \\ [s_{k-m}, \ldots, s_{k-1}] & \text{otherwise} \end{cases}, \quad Y_k = \begin{cases} [y_0, \ldots, y_{k-1}] & \text{if } k \le m \\ [y_{k-m}, \ldots, y_{k-1}] & \text{otherwise} \end{cases} \quad (3,2.5)$$

$$(L_k)_{i,j} = \begin{cases} \tilde{s}_{i-1}^T \tilde{y}_{j-1} & \text{if } i > j \\ 0 & \text{otherwise} \end{cases}, \quad i, j = \begin{cases} 1, 2, \ldots, k & \text{if } k \le m \\ 1, \ldots, m & \text{otherwise} \end{cases}, \quad (3.2.6)$$

$$\tilde{s}_i = \begin{cases} s_i & \text{if } k \le m \\ s_{k-m+i} & \text{otherwise} \end{cases}, \quad \tilde{y}_i = \begin{cases} y_i & \text{if } k \le m \\ y_{k-m+i} & \text{otherwise} \end{cases}. \quad (3.2.7)$$

and

$$\sigma_k = \frac{y_k^{\mathrm{T}} s_k}{\|s_k\|^2}, \quad D_k = \begin{cases} \operatorname{diag}[s_0^T y_0, \ldots, s_{k-1}^T y_{k-1}] & \text{if } k \le m \\ \operatorname{diag}[s_{k-m}^T y_{k-m}, \ldots, s_{k-1}^T y_{k-1}] & \text{otherwise} \end{cases}. \quad (3.2.8)$$

Then we compute $A_k^{\mathrm{T}} B_k A_k$ and $A_k^{\mathrm{T}} g_k$ as follows to form the subproblem in formula (3.2.3).

In order to compute the inverse matrix in (3.2.4), we can compute the Cholesky decomposition of $\sigma_k S_k^T S_k + L_k D_k^{-1} L_k^T$ to obtain $J_k J_k^{\mathrm{T}}$, where J_k is a lower triangular matrix, then we could get

$$\begin{bmatrix} \sigma_k S_k^{\mathrm{T}} S_k & L_k \\ L_k^{\mathrm{T}} & -D_k \end{bmatrix}^{-1} = M_1^{\mathrm{T}} \begin{bmatrix} I_m & 0 \\ & -D^{-1} \end{bmatrix} M_1, \quad (3.2.9)$$

where

$$M_1 = \begin{bmatrix} J_k^{-1} & J_k^{-1} L_k D_k^{-1} \\ 0 & I_m \end{bmatrix}.$$

Now compute $A_k^{\mathrm{T}} B_k A_k$. First compute

$$M_2 = M_1 \begin{bmatrix} \sigma_k S_k^{\mathrm{T}} \\ Y_k^{\mathrm{T}} \end{bmatrix} A_k. \quad (3.2.10)$$

Then by (3.2.4) we could obtain

$$\bar{B}_k = A_k^{\mathrm{T}} B_k A_k = \sigma_k A_k^{\mathrm{T}} A_k - M_2^{\mathrm{T}} \begin{bmatrix} I_m & 0 \\ & -D^{-1} \end{bmatrix} M_2, \quad (3.2.11)$$

$$\bar{g}_k = A_k^{\mathrm{T}} g_k. \tag{3.2.12}$$

The trust region subproblem is

$$\begin{aligned} \min \quad & \psi_k(z) \equiv \bar{g}_k^{\mathrm{T}} z + \frac{1}{2} z^{\mathrm{T}} \bar{B}_k z \\ s.t. \quad & \|z\| \le \Delta_k. \end{aligned} \tag{3.2.13}$$

At each iteration, we need to compute only one row and one column to update $S_k^{\mathrm{T}} S_k$, $S_k^{\mathrm{T}} Y_k$ and $Y_k^{\mathrm{T}} Y_k$, which are needed in (3.2.9), (3.2.10) and (3.2.11).

The second way is to compute the m pairs $(\bar{s}_i, \bar{y}_i), i = k-m, \cdots, k-1$ in the subspace $Span(A_k)$ directly, which is expanded by the columns of the matrix A_k, and A_k is defined in (3.2.1). Then we could compute the approximation $\bar{B}_k$ to the Hessian in the subspace directly by the limited memory BFGS updating formula (3.2.4). In the subspace $Span(A_k)$, the m pairs $(\bar{s}_i, \bar{y}_i)$ are equal to $z_{i+1} - z_i$ and $\nabla F(z_{i+1}) - \nabla F(z_i)$ respectively, where

$$x_k + A_k z_i = x_i, i = k - m, \cdots, k - 1, \tag{3.2.14}$$

according to formula (3.2.2). Suppose $k \ge m$, by (3.2.14) we could get

$$\begin{aligned} & z_k = [0, \ldots, 0]^{\mathrm{T}}, \\ & z_{k-1} = [\overbrace{0, \ldots, 0}^{m}, -\|s_{k-1}\|, \overbrace{0, \ldots, 0}^{m}]^{\mathrm{T}}, \\ & \cdots\cdots \\ & z_{k-i} = [\overbrace{0, \ldots, 0}^{m+1-i}, -\|s_{k-i}\|, \ldots, -\|s_{k-1}\|, \overbrace{0, \ldots, 0}^{m}]^{\mathrm{T}}, \\ & \cdots\cdots \\ & z_{k-m} = [0, -\|s_{k-m}\|, \ldots, -\|s_{k-1}\|, 0, \ldots, 0]^{\mathrm{T}}, \end{aligned}$$

and the gradient $\bar{g}_i$ of $F(z)$(defined in (3.2.2)) at point z_i is $A_k^{\mathrm{T}} g_i$, where $i = k, \cdots, k - m$. So we obtain

$$\begin{aligned} & \bar{s}_i = z_{i+1} - z_i = \|s_i\| e_{i+m+2-k}, \\ & \bar{y}_i = \bar{g}_{i+1} - \bar{g}_i = A_k^{\mathrm{T}} g_{i+1} - A_k^{\mathrm{T}} g_i = A_k^{\mathrm{T}} y_i, \end{aligned} \tag{3.2.15}$$

where $k - m, \cdots, i = k - 1$, and e_j is the j-th unit vector in R^{2m+1}. By formula (3.2.4), we obtain B_k which is an approximation to the Hessian $\nabla^2 F(0)$ in the subspace $Span(A_k)$ as

$$\bar{B}_k = \sigma_k I - [\sigma_k \bar{S}_k \quad \bar{Y}_k] \begin{bmatrix} \sigma_k \bar{S}_k^{\mathrm{T}} \bar{S}_k & \bar{L}_k \\ \bar{L}_k^{\mathrm{T}} & -\bar{D}_k \end{bmatrix}^{-1} \begin{bmatrix} \sigma_k \bar{S}_k^{\mathrm{T}} \\ \bar{Y}_k^{\mathrm{T}} \end{bmatrix}, \tag{3.2.16}$$

where $\bar{S}_k, \bar{L}_k$ and $\bar{D}_k$ are obtained by substituting s_i, y_i with $\bar{s}_i$, $\bar{y}_i$ respectively in formula (3.2.5)–(3.2.8), σ_k is the same as in (3.2.8), and we have

$$\bar{S}_k = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ \|s_{k-m}\| & 0 & \cdots & 0 \\ 0 & \|s_{k-m+1}\| & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \|s_{k-1}\| \\ \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}. \tag{3.2.17}$$

It is easy to see that

$$\bar{S}_k^{\mathrm{T}} \bar{Y}_k = S_k^{\mathrm{T}} Y_k \quad \bar{Y}_k = A_k^{\mathrm{T}} Y_k \quad S_k^T S_k = Diag(\|s_{k-m}\|^2, \cdots, \|s_{k-1}\|^2).$$

We can compute the inverse matrix by formula in (3.2.16) by (3.2.9), and get the trust region subproblem (3.2.13), where $\bar{B}_k$ is a $(2m+1) \times (2m+1)$ matrix, $z \in R^{2m+1}$, $m \ll n$.

Next problem is how to evaluate the approximation to the object f by the subspace and the subspace trust region subproblem. We can do this in the same way as in other trust region methods (see ([13, 15, 28]), and perhaps we have to solve the subspace trust region subproblem several times to get a proper trust region radius. We can also solve the problem min $F_k(z) \equiv f(x_k + A_k z_k)$ in the subspace by iterating the trust region method and solve the subproblem (3.2.13) several times(about 2–5) to get a better local solution.

Algorithm 3.2.1. *The Algorithm without Orthogonalization*

Step 1. *Given* $x_0 \in R^n, \Delta_0 > 0,\ \varepsilon \geq 0,\ B_0 = \sigma_0 I, \sigma_0 > 0,\ S_0 = Y_0 = \mathbf{0}$, $0 < \tau_1 < \tau_3 < \tau_2 < \tau_4 < 1, c_1 > c_2 \geq c_3 \geq c_4 > 1$. *Compute* $f(x_0),\ g_0,\ A_0 = \left[-\frac{g_0}{\|g_0\|}\right]$, *Let* $k = 0$.

Step 2. *If* $\|g_k\|_2 \leq \varepsilon$ *then stop.*

Step 3. *Solve the subproblem (3.2.13) to get* z_k *and* $\psi(z_k)$. *Compute*

$$\begin{aligned} \hat{s}_k &= A_k z_k \\ \rho_k &= \frac{f(x_k) - f(x_k + \hat{s}_k)}{(-\psi(z_k))} \end{aligned} \tag{3.2.18}$$

Step 4. *If* $\rho_k < \tau_1$, *let* $\Delta_k \leftarrow \Delta_k / c_1$ *(we could also use the interpolation and backtracking techniques proposed by [13, 28] here) and go to step 3; Otherwise, do next step.*

Step 5. *If* $\rho_k > \tau_4$, *let* $\Delta_k \leftarrow c_4 \Delta_k$ *and solve the subproblem (3.2.13) again to get* $\tilde{z}_k$ *and* $\tilde{\rho}_k$. *If* $\tilde{\rho}_k \geq \tau_2$, *set* $z_k = \tilde{z}_k$ *and* $\rho_k = \tilde{\rho}_k$*; Otherwise do not change* z_k *and* ρ_k. *Let*

$$\begin{aligned} s_k &= \hat{s}_k \equiv A_k z_k \\ x_{k+1} &= x_k + s_k \\ \Delta_{k+1} &= \begin{cases} \Delta_k / c_3, & \textit{if } \rho_k < \tau_2 \\ c_4 \Delta_k, & \textit{if } \rho_k > \tau_3 \textit{ and } \|z_k\| = \Delta_k \\ \Delta_k, & \textit{otherwise} \end{cases} \end{aligned} \tag{3.2.19}$$

Step 6. *Compute* $g_{k+1} = \nabla f(x_{k+1})$, $y_k = g_{k+1} - g_k$. *Let* $k \leftarrow k+1$ *and update* A_k *in (3.2.1), update* S_k, Y_k *in (3.2.5) or* $\bar{S}_k$, $\bar{Y}_k$ *in (3.2.17) to obtain* $\bar{B}_k$ *according to (3.2.11) or (3.2.16),* $\bar{g}_k$ *according to (3.2.12), then go to step 2.*

In the above algorithm, the parameter could be chosen as $\tau_1 = 0.001$, $\tau_2 = 0.2$, $\tau_3 = 0.7$, $\tau_4 = 0.9$, and $c_1 = 4, c_2 = c_3 = c_4 = 2$. The parameter m should be chosen between 3 and 8. We could also loop between step 2 and step 4 several times(2–5) in order to solve the problem $\min F_k(z) \equiv f(x_k + A_k z)$ by trust region method in the subspace $Span(A_k)$ to get a better local solution. Similar ideas can be found in [40, 41]. The details would be a little tedious, but they do not influence the following convergence analysis.

The convergence analysis of the Algorithm 3.2.1 is in the same way as in [31, 35, 34]. By the definition of A_k in (3.2.1), we get $\|A_k\|_{\mathrm{F}} = \sqrt{\tilde{m}}$, where $\tilde{m} = 2m + 1$. So

$$\sqrt{\tilde{m}} = \|A_k\|_{\mathrm{F}} \geq \|A_k\| \geq \|A_k\|_{\mathrm{F}} / \sqrt{\tilde{m}} = 1 \tag{3.2.20}$$

where $\| \ \|$ denotes the Euclidean norm.

Theorem 3.2.2. *Let $f : R^n \to R$ be convex, bounded below and twice continuous differentiable. There exists a positive constant $M > 0$ such that the Hessian satisfy $\|\nabla^2 f(x)\| \le M$ for all x in the level set $\{x : f(x) \le f(x_1)\}$. Then $\lim_{k\to\infty} g_k = 0$ and the loops between step 2 and step 3 must be finite if $\varepsilon > 0$.*

Proof. By Theorem 4 in [31], let z_k be the solution of (3.2.13), then

$$-\psi(z_k) \ge \frac{1}{2}\|\bar{g}_k\| \min\left(\Delta_k,\ \frac{\|\bar{g}_k\|}{\|\bar{B}_k\|}\right). \tag{3.2.21}$$

First let us prove the boundedness of $\|\bar{B}_k\|$. If $\bar{B}_k$ is defined by (3.2.11), by (3.2.20)

$$\|\bar{B}_k\| = \|A_k^{\mathrm{T}} B_k A_k\| \le \tilde{m}\|B_k\|. \tag{3.2.22}$$

Because B_k is symmetric and positive definite, by $\|\nabla^2 f(x)\| \le M$ we have

$$\begin{aligned}
&y_k = \nabla f(x_{k+1}) - \nabla f(x_k) = \int_0^1 \nabla^2 f(x_k + \theta s_k) s_k \,\mathrm{d}\theta = \bar{G}_k s_k,\\
&\sigma_k = \frac{y_k^{\mathrm{T}} s_k}{\|s_k\|^2} \le M,\\
&\frac{\|y_i\|^2}{y_i^{\mathrm{T}} s_i} = \frac{s_i^{\mathrm{T}} \bar{G}_k^2 s_i}{s_i^{\mathrm{T}} \bar{G}_k s_i} \le M \text{ for all } i \ge 1,
\end{aligned} \tag{3.2.23}$$

where $\bar{G}_k = \int_0^1 \nabla^2 f(x_k + \theta s_k)\,\mathrm{d}\theta$ is positive definite or positive semi-definite by the convexity of f. So in the same way as [32] it follows that

$$\|B_k\| \ \le \mathrm{tr}(B_k) = \mathrm{tr}(\sigma_k I) - \sum_{i=k-m}^{k-1} \frac{\|B_k^{(i)}\|^2}{s_i^{\mathrm{T}} B_k^{(i)} s_i} + \sum_{i=k-m}^{k-1} \frac{\|y_i\|^2}{s_i^{\mathrm{T}} y_i} \le (n+m)M, \tag{3.2.24}$$

where $B_k^{(i)}$ is the i-th updated matrix by BFGS formula from the basic matrix $\sigma_k I$, σ_k is defined in (3.2.8). By (3.2.22) and (3.2.24), we know that $\|\bar{B}_k\|$ is bounded. If we obtain $\bar{B}_k$ by (3.2.15)-(3.2.17), we have

$$\begin{aligned}
&A_k \bar{s}_i = s_i, \quad \bar{y}_i = A_k^{\mathrm{T}} y_i \text{ for } i = k-m, \cdots, k-1,\\
&\|\bar{s}_i\| = \|s_i\|, \bar{s}_i^{\mathrm{T}} \bar{y}_i = \bar{y}_i^{\mathrm{T}} \bar{s}_i = y_i^{\mathrm{T}} s_i = s_i^{\mathrm{T}} y_i.
\end{aligned} \tag{3.2.25}$$

Then by (3.2.20), (3.2.23) and (3.2.25)

$$\begin{aligned}
&\bar{\sigma}_k = \frac{\bar{y}_k^{\mathrm{T}} \bar{s}_k}{\|\bar{s}_k\|^2} = \frac{y_k^{\mathrm{T}} s_k}{\|s_k\|^2} \le M,\\
&\frac{\|\bar{y}_i\|^2}{\bar{y}_i^{\mathrm{T}} \bar{s}_i} = \frac{y_i^{\mathrm{T}} A_k A_k^{\mathrm{T}} y_i}{y_i^{\mathrm{T}} s_i} \le \tilde{m}\frac{\|y_i\|^2}{y_i^{\mathrm{T}} s_i} \le \tilde{m} M.
\end{aligned} \tag{3.2.26}$$

So in the same way, we know that $\|\bar{B}_k\|$ is bounded by (3.2.24). Suppose

$$\|\bar{B}_k\| \leq M_2 \quad \text{for all} \quad k \geq 1. \tag{3.2.27}$$

By (3.2.1), we have

$$\|\bar{g}_k\| = \|A_k^{\mathrm{T}} g_k\| \geq \|g_k\|. \tag{3.2.28}$$

If $\|g_k\|$ is bounded away from 0, there exists a constant $\varepsilon_1 > 0$, such that $\|g_k\| \geq \varepsilon_1$ for all $k \geq 1$. By (3.2.21), (3.2.27) and (3.2.28), we get

$$|\psi(z_k)| = -\psi(z_k) \geq \frac{1}{2}\varepsilon_1 \min\left(\Delta_k,\ \frac{\varepsilon_1}{M_2}\right). \tag{3.2.29}$$

Because f is bounded below, $\{f(x_k)\}$ decrease monotonically, there must exist a f^* such that $f(x_k) \to f^*$. By step 2–step 4 of the algorithm,

$$f(x_k) - f(x_{k+1}) \geq \tau_1|\psi(z_k)| \geq \frac{\tau_1\varepsilon_1}{2} \min\left(\Delta_k,\ \frac{\varepsilon_1}{M_2}\right).$$

So we get $\Delta_k \to 0$, and there exists a $k_1 \geq 1$, such that

$$|\psi(z_k)| = -\psi(z_k) \geq \frac{\varepsilon_1}{2}\Delta_k \quad \text{for all } k \geq 1, \tag{3.2.30}$$

and

$$\begin{aligned} |\rho_k - 1| &= \left|\frac{f(x_k) - f(x_k + A_k z_k) + \bar{g}_k^{\mathrm{T}} z_k + \frac{1}{2} z_k^{\mathrm{T}} \bar{B}_k z_k}{-\psi(z_k)}\right| \\ &= \frac{|-\int_0^1 (1-t) z_k^{\mathrm{T}} A_k^{\mathrm{T}} \nabla^2 f(x_k + tA_k z_k) A_k z_k \,\mathrm{d}t + \frac{1}{2} z_k^{\mathrm{T}} \bar{B}_k z_k|}{|\psi(z_k)|} \\ &\leq \frac{1}{\varepsilon_1} \cdot \frac{(\tilde{m}M + M_2)\|z_k\|^2}{\Delta_k} \leq \frac{\tilde{m}M + M_2}{\varepsilon_1}\Delta_k \to 0. \end{aligned} \tag{3.2.31}$$

So there exist $k_2 \geq k_1$, such that $\rho_k > \tau_2$ for all $k \geq k_2$. By step 4 of the algorithm, there exists a $\bar{\Delta} > 0$ such that $\Delta_k \geq \bar{\Delta}$ for all $k \geq k_2$. This contradicts with $\Delta_k \to 0$. So $\|g_k\|$ can not be bounded away from 0. Now in the same way as [35] or [34], we could get $\lim_{k\to\infty} g_k = 0$. (3.2.30) and (3.2.31) also tell us that the loops between step 2 and step 3 must be finite if $\|g_k\| \geq \varepsilon > 0$. □

By the proof of Theorem 3.2.2, we know that there are many ways for choosing the basis(i.e. the columns of A_k) of subspace which will not influence the convergence result.

表 3.1: Numerical Results for Algorithm 3.2.1

Prob.	Dimension	TRSub	L-BFGS
1	500	19/57/36	37/47/47
1	5000	24/60/46	33/48/48
1	10000	27/67/53	33/48/48
2	1000	51/144/67	53/58/58
3	100	28/124/48	106/111/111
3	1000	30/109/48	—
3	5000	35/142/64	52/61/61
3	10000	39/137/70	52/61/61

We have test the new algorithm on some typical large scale problem on a microcomputer which has an Intel-based 32bit CPU and the main frequency is 1.2GHz. The method for constructing the subproblem is formula (3.2.11) and (3.2.12), and the trust region radius is set to $\Delta_{\max} = 5$ at each iteration and m is set to 6. In our implementation, the algorithm loops between step 2 and step 4 twice in order to get a better local solution, and it was counted as one iteration. We do not use interpolation or backtracking techniques in this version of the new code. The first problem is the "extended separable Rosenbrock function" with n=500, 5000, 10000, where n is the number of variables. The second problem is the "trigonometric" function with n=1000. The 3rd problem is the "extended Powell singular function with n=100, 1000, 5000, 10000. All these problems are listed in [25]. The results are reported in the form:

number of iterations/number of function evaluations/number of gradient evaluations

and the results are compared with the L-BFGS method.

In the Table 3.1, "TRSub" is the new code. The result for L-BFGS in the first line is obtained by executing the L-BFGS code (which is downloaded from netlib) on the microcomputer with m=6. Other results for L-BFGS are extracted from [24] with m=5. Comparing with L-BFGS, we could see that the number of gradient evaluations is almost the same, but the number of function evaluations is more. We think that the number of function evaluations can be decreased if trust region radius Δ_k is not set to $\Delta_{\max}$ at each iteration and interpolation or backtracking techniques are used. When we implement L-BFGS code on the microcomputer for the "extended separable Rosenbrock function" with n=5000, m=5, the line search method failed with "INFO=4", which means the step is at the lower bound STPMIN, which is 10^{-20} by default. These preliminary numerical results shows the new method is promising, and we could expect better numerical stability compared with line search limited memory methods.

§3.3 The Algorithm with Orthogonalization

According to [3], when $B_0 = \sigma_k I$, the compact representation for the BFGS formula is

$$B_k = \sigma_k I - [\sigma_k S_k \quad Y_k] \begin{bmatrix} \sigma_k S_k^{\mathrm{T}} S_k & L_k \\ L_k^{\mathrm{T}} & -D_k \end{bmatrix}^{-1} \begin{bmatrix} \sigma_k S_k^{\mathrm{T}} \\ Y_k^{\mathrm{T}} \end{bmatrix}, \tag{3.3.1}$$

and the compact representation for SR1 is

$$B_k = \sigma_k I + (Y_k - \sigma_k S_k)(D_k + L_k + L_k^{\mathrm{T}} - \sigma_k S_k^{\mathrm{T}} S_k)(Y_k - \sigma_k S_k)^{\mathrm{T}}. \tag{3.3.2}$$

where S_k, Y_k, D_k and Y_k are difined in (3.2.5)–(3.2.8). Both (3.3.1) and (3.3.2) can be represented as:

$$B_k = \sigma_k I + A_k W_k^{-1} A_k^{\mathrm{T}} \tag{3.3.3}$$

where $A_k = [\sigma_k S_k, Y_k]$ or $Y_k - \sigma_k S_k$, W_k is the corresponding matrix in (3.3.1) or (3.3.2). In fact we can solve the quadratic subproblem (2.1.2)–(2.1.3) exactly in the subspace spanned by $[-g_k, S_k, Y_k]$.

Theorem 3.3.1. *Let the columns of the matrix Z_k be the orthogonal basis of span$\{S_k, Y_k, g_k\}$. B_k is defined in (3.3.3). Then the solution of (2.1.2)–(2.1.3) is equivalent to solve:*

$$\begin{aligned} \min \quad & \psi(z) \equiv \bar{g}_k^{\mathrm{T}} z + \frac{1}{2} z^{\mathrm{T}} \bar{B}_k z \\ s.t. \quad & \|z\| \leq \Delta. \end{aligned} \tag{3.3.4}$$

where $\bar{g}_k = Z_k^{\mathrm{T}} g_k$ and $\bar{B}_k = Z_k^{\mathrm{T}} B_k Z_k$. That is to say, if s_k is a solution of (2.1.2)–(2.1.3), then $Z_k^T s_k$ is a solution of (3.3.4). On the other hand, if z_k is a solution of (3.3.4), then $Z_k z_k$ must be a solution of (2.1.2)–(2.1.3).

Proof. Let $Z_k \in R^{n\times r}$, $W_k \in R^{n\times(n-r)}$ be an orthogonal matrix such that $[Z_k, W_k]$ is an $n \times n$ orthogonal matrix. Then for any $s \in R^n$, there exists and only exists one pair $z \in R^r,\ w \in R^{n-r}$, such that $s = Z_k z + W_k w$, and we have

$$\begin{aligned} g_k^{\mathrm{T}} s + \frac{1}{2} s^{\mathrm{T}} B_k s = g_z^{\mathrm{T}} z + g_w^{\mathrm{T}} w + \frac{1}{2} z^{\mathrm{T}} Z_k^{\mathrm{T}} B_k Z_k z + \\ \frac{1}{2} w^{\mathrm{T}} W_k^{\mathrm{T}} B_k W_k w + z^{\mathrm{T}} Z_k^{\mathrm{T}} B_k W_k w \end{aligned} \tag{3.3.5}$$

where $g_z = Z_k^{\mathrm{T}} g_k, g_w = W_k^{\mathrm{T}} g_k$. By the definition of Z_k , W_k and A_k, we get $span\{A_k\} \subseteq span\{Z_k\}$, and

$$A_k^{\mathrm{T}} W_k = 0,\ B_k W_k = \sigma_k W_k,\ g_w = W_k^{\mathrm{T}} g_k = 0$$

Then by (3.3.5),

$$g_k^{\mathrm{T}} s + \frac{1}{2} s^{\mathrm{T}} B_k s = (g_z^{\mathrm{T}} z + \frac{1}{2} z^{\mathrm{T}} Z_k^{\mathrm{T}} B_k Z_k z) + \frac{1}{2} \sigma_k w^{\mathrm{T}} w$$

It is easy to see that $\|s\|^2 = \|z\|^2 + \|w\|^2$, so (2.1.2)–(2.1.3) is equivalent to

$$\begin{aligned} \min\ \ & (g_z^{\mathrm{T}} z + \frac{1}{2} z^{\mathrm{T}} Z_k^{\mathrm{T}} B_k Z_k z) + \frac{1}{2} \sigma_k w^{\mathrm{T}} w \\ s.t.\ \ & \|z\|^2 + \|w\|^2 \le \Delta^2. \end{aligned} \tag{3.3.6}$$

It is easy to see the solution of (3.3.3) is equivalent to solve (3.3.6) with $w = 0$, where $\bar{B}_k = Z_k^{\mathrm{T}} B_k Z_k$ and $\bar{g}_k = g_z$. □

Now according to Theorem 3.3.1, we could solve the quadratic subproblem (2.1.2)–(2.1.3) exactly by solving (3.3.3), where $z \in R^{2m+1}, 3 \le m \le 10$, and $m \ll n$ when n is large, and we could construct the limited memory trust region algorithms based on the traditional trust region philosophy as follows.

Algorithm 3.3.2. *The Limited Memory Trust Region Algorithm with Orthogonalization for Unconstrained Optimization*

Step 1. *Given* $x_0 \in \Re^n$, $\Delta_0 > 0$, $0 \le \tau_0 < \tau_1 < 0.5 < \tau_2 < 1$, $\epsilon \ge 0$, $\varepsilon_M > 0$. *Compute* $f(x_0)$ *and* $g_0 = \nabla f(x_0)$. *Set* $Z_0 = [\frac{g_0}{\|g_0\|}]$, $\bar{B}_0 = \sigma_0$, $\bar{g}_0 = \|g_0\|$. *Let* $r_0 := 1$, $k := 0$ 。

Step 2. *If* $\|g_k\| < \varepsilon$, *then stop.*

Step 3. *Solve (3.3.3) to obtain* z_k *and* $\psi_k(z_k)$. *if* $\|z_k\| < \varepsilon_M$, *stop; otherwise compute* $s_k = Z_k z_k$ *and do next step.*

Step 4. *If* $Pred_k = \psi_k(0) - \psi_k(z_k) < \epsilon_M$, *stop; otherwise compute*

$$\eta_k = \frac{Ared_k}{Pred_k} = \frac{f(x_k) - f(x_k + s_k)}{\psi_k(0) - \psi_k(z_k)}. \tag{3.3.7}$$

Step 5. *If* $\eta_k < \tau_0$, *set* $\Delta_k \leftarrow \Delta_k/4$ *and goto Step 3. Otherwise set* $x_{k+1} = x_k + s_k$ *and choose* Δ_{k+1} *which satisfies*

$$\Delta_{k+1} = \begin{cases} \Delta_k/2 & \text{if } \eta_k < \tau_1, \\ 2\Delta_k & \text{if } \eta_k > \tau_2 \text{ and } \|s_k\| > 0.8\Delta_k \\ \Delta_k & \text{otherwise} \end{cases} \tag{3.3.8}$$

Step 6. *Evaluate* g_{k+1}. *Compute the orthonormal basis matrix* Z_{k+1} *of* $Span\{g_{k+1}, S_{k+1}, Y_{k+1}\}$ *and its rank* r_{k+1}. *Compute* $\bar{g}_{k+1} = Z_{k+1}^T g_{k+1}$ *and* $\bar{B}_{k+1} = Z_{k+1}^T B_{k+1} Z_{k+1}$ *according to (3.3.1) (if the BFGS formula is used) or (3.3.2) (if the SR1 formula is used). Set* $k := k+1$ *and goto Step 2.*

The main problem in the implementation of the Algorithm 3.3.2 is how to compute the orthonormal basis matrix Z_{k+1}, the reduced gradient $\bar{g}_{k+1} = Z_{k+1}^T g_{k+1}$ and the reduced approximate Hessian $\bar{B}_{k+1} = Z_{k+1}^T B_{k+1} Z_{k+1}$.

For convenience, let us assume that $k \geq m$ in the following. Suppose that we have gotten the orthonormal basis matrix Z_k of the subspace $Span\{S_k, Y_k\}$ and its rank r_k. s_k, y_k and g_{k+1} have also been computed. In order to get the matrix $[g_{k+1}, S_{k+1}, Y_{k+1}]$, we have to remove the pair (s_{k-m}, y_{k-m}) and g_k from the matrix $[g_k, S_k, Y_k]$ before adding the pair (s_k, y_k) and g_{k+1}. In order to compute the QR decomposition of the matrix $[S_k, Y_k]$ when deleting or adding a pair of (s, y), we could store it in an interleaved way, and denote

$$\tilde{A}_k = [s_{k-m+1}, y_{k-m+1}, \cdots, s_{k-1}, y_{k-1}, s_k, y_k]. \tag{3.3.9}$$

Then according to Daniel, etc. [11], we could use reorthogonalization procedure to update the QR decomposition of $\tilde{A}_k$ conveniently when adding or deleting a pair of (s, y).

Suppose that we have gotten the QR decompostion of $\tilde{A}_k$, i.e.,

$$\tilde{A}_k = \tilde{Z}_k R_k. \tag{3.3.10}$$

where R_k is a trapezoidal matrix. Let R_k^s be composed of the odd columns of R_k, R_k^y be composed of the even columns of R_k, then

$$S_k = \tilde{Z}_k R_k^s, \quad Y_k = \tilde{Z}_k R_k^y, \tag{3.3.11}$$

and we have

$$A_k^{\mathrm{T}} \tilde{Z}_k = \begin{cases} [\sigma_k R_k^s, R_k^y]^{\mathrm{T}} & \text{if } A_k = [\sigma_k S_k, Y_k] \text{ (for BFGS)} \\ (R_k^y - \sigma_k R_k^s)^{\mathrm{T}} & \text{if } A_k = Y_k - \sigma_k S_k \text{ (for SR1),} \end{cases} \tag{3.3.12}$$

According to [11], we could find a unitary matrix U_k by a series of Givens rotations such that

$$U_k \tilde{R}_k = \begin{bmatrix} \tau & \hat{R}_k \\ t & 0 \end{bmatrix} \tag{3.3.13}$$

where $\hat{R}_k$ is still a trapezoidal matrix. Let $\hat{A}_k$ be the matrix which is obtained by deleting the first row s_{k-m} from the matrix $\tilde{A}_k$, $\hat{Z}_k$ be the matrix which is obtained by by deleting the first row from the matrix $Z_k U_k^T$. Then by (3.3.11) and (3.3.13) we get

$$\hat{A}_k = \hat{Z}_k \hat{R}_k, \tag{3.3.14}$$

which is the QR decomposition of the matrix $\hat{A}_k$. In the same way we could get the QR decomposition of the matrix which is obtained by deleting the first row y_{k-m} from the matrix $\hat{A}_k$ and get the corresponding QR decomposition. Let us still denote it as (3.3.14) for convenience, i.e., now $\hat{A}_k$ is the matrix deleted the pair (s_{k-m}, y_{k-m}) from $\tilde{A}_k$ in (3.3.14). Then by the reorthogonalization method (which is described in detail in Chapter 2), we could adding the two vectors s_k, y_k in sequence and obtain the QR decomposition for the matrix $\tilde{A}_{k+1}$, and we have

$$\tilde{A}_{k+1} = [s_{k-m+1}, y_{k-m+1}, \cdots, s_{k-1}, y_{k-1}, s_k, y_k] = \tilde{Z}_{k+1} R_{k+1} \tag{3.3.15}$$

where R_{k+1} is a trapezoidal matrix. Just as above, denote R_{k+1}^s as the matrix which is composed of the odd columns of R_{k+1} and R_{k+1}^y be the matrix composed of the even columns of R_{k+1}.

Next we have to get the QR decomposition of the matrix $[g_{k+1}, S_{k+1}, Y_{k+1}]$. According to (3.3.15), we have obtained the orthonormal basis matrix $\tilde{Z}_{k+1}$ of the subspace $Span\{S_{k+1}, Y_{k+1}\} = Span\{\tilde{A}_{k+1}\}$. Then by reorthogonalization method we get

$$g_{k+1} = \tilde{Z}_{k+1} u_k + \rho_{k+1} z_{k+1} \tag{3.3.16}$$

where $u_k = \tilde{Z}_{k+1}^T g_{k+1}$, $\quad \rho_{k+1} = g_{k+1}^T z_{k+1} = \sqrt{\|g_{k+1}\|^2 - \|u_k\|^2} \geq 0$. If $\rho_{k+1} = 0$, by (3.3.3), (3.3.15) and (3.3.16), we have $Z_{k+1} = \tilde{Z}_{k+1}$ and

$$\begin{aligned} \bar{g}_{k+1} &= Z_{k+1}^T g_{k+1} = u_k, \\ \bar{A}_{k+1} &= Z_{k+1}^T A_{k+1} = \begin{cases} [\sigma_{k+1} R_{k+1}^s, R_{k+1}^y] & \text{for BFGS} \\ R_{k+1}^y - \sigma_{k+1} R_{k+1}^s & \text{for SR1} \end{cases}, \\ \bar{B}_{k+1} &= Z_{k+1}^T B_{k+1} Z_{k+1} = \sigma_{k+1} I + \bar{A}_{k+1} W_{k+1}^{-1} \bar{A}_{k+1}^{\mathrm{T}}. \end{aligned} \tag{3.3.17}$$

If $\rho_{k+1} > 0$, by (3.3.3), (3.3.15) and (3.3.16), we have $Z_{k+1} = [\tilde{Z}_{k+1}, z_{k+1}]$ and

$$\begin{aligned} \bar{g}_{k+1} &= Z_{k+1}^T g_{k+1} = [u_k^T, \rho_{k+1}]^T, \\ \bar{A}_{k+1} &= Z_{k+1}^T A_{k+1} = \begin{bmatrix} \sigma_{k+1} R_{k+1}^s & R_{k+1}^y \\ 0 & 0 \end{bmatrix} \text{ for BFFS }, \\ &\left(\text{or} \quad \begin{bmatrix} R_{k+1}^y - \sigma_{k+1} R_{k+1}^s \\ 0 \end{bmatrix} \text{ for SR1} \right), \\ \bar{B}_{k+1} &= Z_{k+1}^T B_{k+1} Z_{k+1} = \sigma_{k+1} I + \bar{A}_{k+1} W_{k+1}^{-1} \bar{A}_{k+1}^{\mathrm{T}}. \end{aligned} \tag{3.3.18}$$

We summarize the above procedure for computing $\bar{g}_{k+1}$ and $\bar{B}_{k+1}$ in the following algorithm.

Algorithm 3.3.3. *The Algorithm for evaluating $\bar{g}_{k+1}$ and $\bar{B}_{k+1}$.*

Step 1. *Given a small positive number δ which satisfies $0 < \delta < 0.5$.*

Step 2. *If $k \geq m$, obtain (3.3.14) by removing s_{k-m}, y_{k-m} from $\tilde{A}_k$ in sequence through a series of Givens rotations.*

Step 3. *Using the reorhogonalization method two times to obtain (3.3.15) by adding s_k, y_k to $\hat{A}_k$ in sequence. Let R_{k+1}^s be the matrix which is composed of the odd columns of R_{k+1} and R_{k+1}^y be the matrix composed of the even columns of R_{k+1}.*

Step 4. *Using the reorhogonalization method to obtain (3.3.16).*

Step 5. *If $\rho_{k+1} > \delta \|g_{k+1}\|$, set $Z_{k+1} = [Z_k, z_{k+1}]$, $r_{k+1} = r_k + 1$, and compute $\bar{g}_{k+1}$ and $\bar{B}_{k+1}$ according to (3.3.18); Otherwise, set $Z_{k+1} = Z_k$, $r_{k+1} = r_k$, and compute compute $\bar{g}_{k+1}$ and $\bar{B}_{k+1}$ according to (3.3.17).*

We could get the same convergence result for Algorithm 3.3.2 as that for Algorithm 3.2.1.

Theorem 3.3.4. *Let $f : R^n \to R$ be convex, bounded below and twice continuous differentiable. There exists a positive constant $M > 0$ such that the Hessian*

表 3.2: Numerical Results for Algorithm 3.3.2(n=10000)

Prob	Algorithm 3.3.2		L-BFGS	
	Iter/NF/NG	f^*	Iter/NF/NG	f^*
6	67/72/67	5.784e-19	*/1/2/2*	5.961e+58
8	68/80/68	0.09900	71/85/85	0.09900
9	*1/1/1*	inf	*1/1/1*	inf
13	82/122/82	2.195e-08	65/76/76	2.170e-08
14	39/60/39	1.040e-11	38/51/51	2.495e-16
15	52/120/52	1.087e-08	51/61/61	1.276e-10

satisfy $\|\nabla^2 f(x)\| \le M$ *for all* x *in the level set* $\{x : f(x) \le f(x_1)\}$*. Then* $\lim\limits_{k\to\infty} g_k = 0$ *and the loops between step 2 and step 3 must be finite if* $\varepsilon > 0$.

Proof. Let z_k be the solution of (3.3.3). By the Theorem 4 in [31] we have

$$-\psi(z_k) \ge \frac{1}{2}\|\bar{g}_k\| \min\left(\Delta_k,\ \frac{\|\bar{g}_k\|}{\|\bar{B}_k\|}\right) \tag{3.3.19}$$

By (3.3.16) we get

$$\|g_k\|^2 = \|u_{k-1}\|^2 + \rho_k^2, \tag{3.3.20}$$

where $u_{k-1} = Z_{k-1}^T g_k$. According to Algorithm (3.3.3), when $\rho_k \le \delta\|g_k\|$, g_k is rejected and $Z_k = Z_{k-1}$, $\bar{g}_k = Z_k^T g_k = Z_{k-1}^T g_k = u_{k-1}$. Then by (3.3.20) and $0.5 > \delta > 0$ we get

$$\|\bar{g}_k\| = \|u_{k-1}\| = \sqrt{\|g_k\|^2 - \rho_k^2} \ge \sqrt{\|g_k\|^2 - \delta^2\|g_k\|^2} \ge \frac{1}{2}\|g_k\|. \tag{3.3.21}$$

When $\rho_k > \delta\|g_k\|$, we have $g_k \in span\{Z_k\}$, and $g_k = Z_k Z_k^T g_k = Z_k \bar{g}_k$. So we have $\|g_k\| = \|\bar{g}_k\|$. In both cases, we have (3.3.21). Then in the same way as that in the proof of Theorem 3.2.2, we could obtain the results. □

We have tested Algorithm 3.3.2 on six problems whose dimensions are variable, which are proposed in [25]. The number of variables for the six problems are all set as $n = 10000$. The parameters for Algorithm 3.3.2 and Algorithm 3.3.3 are chosen as follows: $\varepsilon = 10^{-6}$, $\varepsilon_M = 10^{-16}, \tau_0 = 0$, $\tau_1 = 0.25$, $\tau_2 = 0.75$, $\delta = 10^{-8}$. In Table 3.2, numerical results for Algorithm 3.3.2 and L-BFGS are compared. The first column is the number of the test problem proposed in [25]. The number of objective function evaluations and gradient evaluations are referred as "NF" and "NG" respectively, and the number of iterations is referred as "Iter" in the table. "f^*" is the optimal function value gotten by the two algorithms. The "*" on both side of the data indicated that the corresponding algorithm failed to get a result satisfying $\|g_k\| < \varepsilon = 10^{-6}$. In the table, we can see that both the algorithms failed on problem 9. Algorithm 3.3.2 succeeded in solving the problem 6, but L-BFGS method failed on this problem. When the sum of the number of the objective evaluations and the number of the gradient evaluations is used as the performance metric, Algorithm 3.3.2 wins on problem 8 and 14 and L-BFGS wins on the other two problems. These preliminary numerical results indicated the new algorithm is promising.

第四章 Conclusions and Further Works

In Chapter 1 we introduced the testing environment CUTEr for constrained and unconstrained nonlinear optimization algorithms. It is composed of three parts: the CUTEr evaluation tools and interfaces, the SIF decoder and test problems written in SIF format. The group partial separability for nonlinear function and its relation with the SIF format are discussed in detail. Methods to obtain, install, uninstall, resize and update CUTEr are inroduced. Fianally, how to test our own optimization packages written in C or FORTRAN language in the CUTEr environment is described in detail.

In Chapter 2 the subspace properties of trust region methods for unconstrained optimization are studied, provided that the approximate Hessian is updated by quasi-Newton formulae, and the initial Hessian approximation is appropriately chosen. Due to the subspace properties, we can solve the quadratic subproblem with reduced Hessian and reduced gradient to obtain the trial step, and we can update the approximate reduced Hessian in the subspace. The equivalence of the subspace trust region methods with the traditional trust region methods is shown in §2.2. When the dimension of the subspace is much smaller than the number of variables, the amount of computation can be reduced significantly. Numerical results in §2.4 indicates that the subspace trust region methods not only gained in CPU time, but also require less function and gradient evaluations than the traditional trust region methods which are implemented in the whole space. When trust region methods are implemented in subspaces, CPU time is significantly reduced for large-scale problems.

The subspace trust region algorithms can be different from the traditional trust region algorithms if we choose the parameter $1 > \nu > 0$ in Algorithm 2.3.1 or $\mu_1 > 0$ in Algorithm 2.3.3. A convergence theorem about the case is also given.

We could also use the reinitialization technique proposed by [24] and [17] conveniently in Algorithm 2.3.1. But only with reinitialization at each iteration, the numerical results are not as good as the case without reinitialization, although reinitialization should reduce the influence of a poor initial estimate of the Hessian from intuition. Thus the reinitialization technique for our subspace trust region algorithm is not effective as it for line search algorithms.

Combine reinitialization with lingering technique, Algorithm 2.3.3 obtains better numerical results when the BFGS formula is used, and the reinitialization parameter $\sigma_k = \frac{y_k^T y_k}{s_k^T y_k}$ is the most preferred one for large scale test problems. On the set of 153 problems whose size ranging from small to medium, the framework of Algorithm 2.3.3 does not work when the SR1 formula is used. How to combine reinitialization with lingering technique in this case still needs further studies.

In Chapter 3, two limited memory trust region algorithms for large scale unconstrained optimization are constructed, which is based on the analysis of the searching direction in limited memory BFGS method, but could also use other matrix updating formula, e.g., the SR1 formula. Just as proceeding from quasi-Newton methods with line searching to trust region method, we proceed from limited memory quasi-Newton methods with line searching to limited memory quasi-Newton trust region method. The size of the subspace trust region subproblems is very small(about 6–16). We can solve these subproblems very quickly and exactly, and the method can be applied to very large scale problems. Convergence properties of these methods are analyzed. Preliminary numerical results show that the implementations of the new algorithms are encouraging, and because of the strong convergence properties of trust region method and the special structure of the subspace trust region subproblem, we could expect to get better numerical stability compared with other line searching limited memory methods. The main problem in the new method is how to construct the subspace and how adjust the trust regions. There are many other ways for choosing the basis of the subspace. If the basis of the subspace is not chosen properly, the method may converge very slowly, although we can still prove its convergence in theory. There are also other matrix updating formula can be chosen, such as the SR1 formula. Further studies will show how to evaluate the effective of the subspace and better ways to update the subspaces and the trust regions.

参考文献

[1] I. Bongartz, A. R. Conn, N. I. M. Gould and Ph. L. Toint, *CUTE: Constrained and Unconstrained Testing Environment*, ACM Transactions on Mathematical Software, 21(1995), pp. 123–160.

[2] A.Buckley and A. LeNir,*QN-like variable storage conjugate gradients*, Math. Prog. 27(1983), pp. 155-175.

[3] R. Byrd, J. Nocedal and R. Schnabel, *Representations of Quasi-Newton Matrices and their use in Limited Memory Methods*, Mathematical Programming 63(1994), pp. 129–156.

[4] R. Byrd, R. B. Schnabel and G. A. Shultz, *Approximation solution of the trust region problem by minimization over two-dimensional subspaces*, Mathematical Programming, 40(1988),pp. 247–263.

[5] J. V. Burke and A. Wiegmann, *Notes on Limited Memory BFGS Updating in a Trust-Region Framework*, Preprint, Department of Mathematics, University of Washington, Seattle, WA, 1996. http://citeseer.ist.psu.edu/burke96notes.html.

[6] Burdakov and Yaxiang Yuan, *Limited Memory Trust Region Methods for Solving Large-Scale Unconstrained Optimization*, 2002, preprint paper.

[7] A. R. Conn, N. I. M. Gould, A. Sartenaer and Ph. L. Toint, *On the Iterated-Subspace Minimization Methods for Nonlinear Optimization with a Combination of General Equality and Linear Constraints*, in Proceedings on Linear and Nonlinear Conjugate Gradient-Related Methods, L. Adams and J.L. Nazareth, eds., SIAM, pp. 50–78, 1996.

[8] A. R. Conn, N. I. M. Gould and Ph. L. Toint, *Testing a class of methods for solving minimization problems with simple bounds on the variables*, Mathematics of Computation, 50(1988), pp. 399-430.

[9] A. R. Conn, N. I. M. Gould, and Ph. L. Toint,*LANCELOT: a Fortran package for Large-scale Nonlinear Optimization (Release A)*, Springer Series in Computational Mathematics, Springer Verlag, Heidelberg, Berlin, New York, 1992.

[10] A. R. Conn, N. I. M. Gould and Ph. L. Toint, *Trust-Region Methods*, SIAM Publications, Philadelphia, Pennsylvania, 2000.

[11] W.Daniel, W. B.Gragg, L.Kaufman, AND G. W.Stewart, *Reorthogonalization and stable algorithms for updating the Gram-Schmidt QR factorization*, Mathematics of Computation, 30(1976), pp. 772–795.

[12] J. E. Dennis and H. H. W. Mei, *Two new unconstrained optimization algorithms which use function and gradient values*, Journal of Optimization and

Applications, 28(1979), pp. 453-482.

[13] J. E. DENNIS, JR., AND R. B. SCHNABEL, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.

[14] E. D. DOLAN AND J. J. MORÉ, *Benchmarking optimization software with performance profiles*, Mathematical Programming, 91(2002), pp. 201–213.

[15] R. FLETCHER, *Practical Methods of Optimization*, 2nd edition, John Wiley and Sons, New York, 1987.

[16] D. M. GAY, *Computing optimal locally constrained steps*, SIAM J. Sci. Stat. Comput., 2(1981), pp. 186–197.

[17] P. E.GILL AND M. W.LEONARD, *Reduced-Hessian quasi-Newton methods for unconstrained optimization*, SIAM J. Optim. , 12(2001), pp. 209–237.

[18] G. H. GOLUB AND C. F. VAN LOAN, *Matrix Computations*, 3rd Edition, Johns Hopkins University Press, Baltimore and London, 1996.

[19] N. I. M. GOULD, D. ORBAN AND PH. L. TOINT, *CUTEr and sifdec: A Constrained and Unconstrained Testing Environment, revisited*, ACM Trans. Math. Softw., 29(2003), pp. 373–394.

[20] J. C.GILBERT AND C.LEMARÉCHAL, *Some numerical experiments with variable storage quasi-Newton algorithms*, Math. Prog. 45(1989), pp.407–436.

[21] A. GRIEWANK AND PH. L. TOINT, *Partitioned variable metric updates for large structured optimization problems*, Numerische Mathematik, 39, 119–139, 1982.

[22] A. GRIEWANK AND P. L. TOINT, *On the unconstrained optimization of partially separable objective functions*, In Nonlinear Optimization 1981, M.J.D. Powell, editor, , Academic Press, London, 1982. pp. 301-312.

[23] L.KAUFMAN, *Reduced storage, quasi-Newton trust region approaches to function optimization*, SIAM Journal on Optimization, 10(1999), pp. 56–69.

[24] D. C.LIU AND J.NOCEDAL, *On the limited memory BFGS method for large scale optimization*, Mathematical Programming, 45(1989), pp. 503-528.

[25] J. J. MORÉ, B. S. GARBOW AND K. E. HILLSTROM, *Testing unconstrained optimization software*, ACM Transactions on Mathematical Software 7(1981), pp. 17–41.

[26] J. J. MORÉ AND D. C.SORENSEN, *Computing a trust region step*, SIAM J. Sci. Stat. Comput., 4(1983), pp. 553–572.

[27] J. NOCEDAL, *Updating quasi-Newton matrices with limited storage*, Mathematics of Computation 35(1980), pp. 773–782.

[28] J. NOCEDAL AND Y. YUAN, *Combining trust region and line search techniques*,

in Advances in Nonlinear Programming, Proceedings of the 1996 International Conference on Nonlinear Programming, Y. Yuan, ed., Kluwer Academic Publishers,1998, pp. 153–176.

[29] M. J. D. POWELL, *A new algorithm for unconstrained optimization*, in Nonlinear Programming, : J.B. Rosen, O.L. Mangasarian and K. Ritter, eds., Academic Press, New York, 1970, pp. 31-66.

[30] M.J.D. POWELL, *A hybrid method for nonlinear equations*, in *Numerical Methods for Nonlinear Algebraic Equations*, P. Robinowitz, ed., Gordon and Breach Science, London, 1970, pp. 87-144.

[31] M. J. D. POWELL, *Convergence properties of a class of minimization algorithms*, in Nonlinear Programming, 2, O. L. Mangasarian, R. R. Meyer and S. M. Robinson, eds., Academic Press, New York, 1975, pp. 1–27.

[32] M. J. D. POWELL, *Some global convergence properties of a variable metric algorithm for minimization without exact line search*, in Nonlinear Programming, SIAM-AMS Proceedings IX, R. W. Cottle and C. E. Lemke, eds., SIAM, Philadelphia, PA, 1976, pp. 54–72.

[33] M. J. D. POWELL, *On the global convergence of trust region algorithms for unconstrained minimization*, Mathematical Programming, 29(1984), pp. 297–303.

[34] G. A. SCHULTZ, R. B. SCHNABEL AND R. H. BYRD, *A family of trust-region-based algorithms for unconstrained minimization with strong global convergence properties*, SIAM Journal on Numerical Analysis 22(1985), pp. 47–67.

[35] D. C. SORENSEN, *Newton's method with a model trust region modifications*, SIAM J. Numerical Analysis, 19(1982), pp. 409–426.

[36] T. STEIHAUG, *The conjugate gradient method and trust regions in large scale optimization*, SIAM J. Numerical Analysis, 20(1983), pp. 626–637.

[37] G. W. STEWART, *Matrix Algorithms, Volume 1: Basic Decompostions*, SIAM, Philadelphia, 1998.

[38] J.VLČEK AND L.LUKŠAN, *New variable metric methods for unconstrained minimization covering the large-scale case*, Technical report No. V876, October 2002, Institute of Computer Science, Academy of Sciences of the Czech Republic.

[39] J. STOER AND Y. YUAN, *A subspace study on conjugate gradient algorithms*, ZAMM Z. angew. Math. Mech., 75(1995),pp. 69–77.

[40] M. A.WOLFE, *A quasi-Newton method with memory for unconstrained function minimization*, J.Inst .Maths . Applics .15(1975), pp. 85–94.

[41] M.A.WOLFE AND C.VIAZMINSKY, *Supermemory descent methods for unconstrained minimization*, J. of Optimization theory and applications, 18(1976),

pp. 455-468

[42] Y. Yuan,*A review of trust region algorithms for optimization*, in ICM99: Proceedings of the Fourth International Congress on Industrial and Applied Mathematics, J. M. Ball and J. C. R. Hunt, eds., Oxford University Press,Oxford, 2000, pp. 271–282.

[43] Y. Yuan,*On the truncated conjugate gradient method*, Math. Prog., 87(2000), pp. 561–571.

致　谢

本文是在我的合作导师袁亚湘研究员的悉心指导和亲切关怀下完成的。两年的博士后科研工作，极大地提高了我的学术能力和学术视野。合作导师袁亚湘研究员渊博的知识、严谨求实的治学态度及其敏锐的学术眼光，将使我终身受益，在此谨表示深深的谢意。

感谢课题组的戴　虹副研究员，时贞军博士，文再文硕士，李改弟博士，王丽萍博士，李多全，周庆华，严涛，徐玲玲，夏勇，郑真真以及讨论班的其他同志。感谢他们给予我的许多关心和帮助，与他们一起讨论问题常使我受益匪浅。

感谢张林波研究员和白英工程师在计算机使用和 C 语言编程方面给予的指导。在两年的博士后工作期间，有许多老师如计算数学所的张时珍（已退休），张纪平，吴继萍，樊建荣等老师，人教处的尹永华等老师给予了我很多的关心和帮助。在此对他们表示衷心的感谢和深深的祝福。

感谢计算数学与科学工程计算研究所提供的舒适的学习生活环境和良好的学术氛围，以及科学与工程计算国家重点实验室优越的上机条件。计算数学所的博士后定期学术报告对我们有很大的帮助，周爱辉研究员，吴继萍，张文琼等老师为此付出许多劳动，在此表示感谢。

最后，我向我的父母及爱人李玉萍女士谨致特别的感谢。二年来，她们热情地支持我从事我的学业和工作，为承担家庭的生活重任付出了极大的辛苦。没有她们的支持我将很难顺利完成我的博士后工作。

在站期间完成文章目录

- ZHOUHONG WANG， YAXIANG YUAN AND ZAIWEN WEN, *A subspace trust region method for large scale unconstrained optimization*， 2003 年 ” 数值代数与优化国际会议 ”(International Conference on Numerical Linear Algebra and Optimization '2003) 论文集，SCIENCE PRESS， Beijing， 2004. pp. 265–274.

- 王周宏，关于无约束有限内存信赖域方法的研究，中国运筹学会第七届学术交流会论文集， Global-Link Publishing Company, Hong Kong, 2004. pp. 72–77.

- ZHOUHONG WANG， YAXIANG YUAN， *A subspace implementation of a quasi-Newton trust region method for unconstrained optimization*， to appear.