Conjugate Gradient Algorithms Closest to Self-Scaling Memoryless BFGS Method based on clustering the eigenvalues of the self-scaling memoryless BFGS iteration matrix or on minimizing the Byrd-Nocedal measure function with Different Wolfe Line Searches for Unconstrained Optimization

Neculai Andrei[1]
Center for Advanced Modeling and optimization,
Academy of Romanian Scientists
E-mail: nandrei@ici.ro

Technical Report No.2/2019
April 18, 2019
Bucharest, Romania

Abstract Three new procedures for computation the scaling parameter in the self-scaling memoryless Broyden-Fletcher-Goldfarb-Shanno search direction closest to the Perry and Shanno search direction are presented. The first two are based on clustering the eigenvalues of the self-scaling memoryless Broyden-Fletcher-Goldfarb-Shanno iteration matrix by using the determinant or the trace of this matrix. The third one is based on minimizing the measure function of Byrd and Nocedal (SIAM J Numer Anal 26:727-739, 1989). For all these three algorithms the sufficient descent condition is established. The stepsize is computed using three line search procedures based on: the standard Wolfe line search (SIAM Rev 11:226-235, 1969, SIAM Rev 13:185-188, 1971), the approximate Wolfe line search of Hager and Zhang (SIAM J Optim 16:170-192, 2005) or the improved Wolfe line search of Dai and Kou (SIAM J Optim 23:296-320, 2013). Under the improved Wolfe line search the global convergence of these algorithms is established. By using 80 unconstrained optimization test problems, with different structures and complexities, it is proved that the performances of the self-scaling memoryless algorithms with the approximate or the improved Wolfe line searches are better than the performances of the same algorithms with the standard Wolfe line search. For all procedures for the stepsize computation using the standard, the approximate, or the improved Wolfe line search, the performances of the self-scaling memoryless algorithms based on the determinant or on the trace of the iteration matrix are similar. Using the standard, the approximate or the improved Wolfe line search, the performances of the self-scaling memoryless algorithms based on the determinant or on the trace of the iteration matrix or on minimizing the measure function are better that those of CG-DESCENT with Wolfe or with the approximate Wolfe line search of Hager and Zhang (SIAM J Optim 16:170-192, 2005). For all three procedures for stepsize computation using the standard, the approximate or the improved Wolfe line search, the self-scaling memoryless algorithm based on minimizing the measure function of Byrd and Nocedal is top performer versus the same algorithms based

[1] *Dr. Neculai Andrei is full member of Academy of Romanian Scientists*.
The numerical results presented in this Technical Report are obtained with CG3x8.for package which is a modification of the CG-DESCENT (version 1.4) package implementing a conjugate gradient algorithm. CG3x8 implements three line searches: standard Wolfe, approximate Wolfe and improved Wolfe, with eight conjugate gradient methods: Hager and Zhang, Minim DETERMINANT (same as Dai and Kou), Minim TRACE, Minim measure function FI of Byrd and Nocedal, Hestenes and Stiefel, Dai and Yuan, Polak, Ribiere and Polyak, Combination trace & log(det).

on the determinant or on the trace of the iteration matrix.

Keywords Unconstrained optimization . Self-scaling . Quasi-Newton method . Standard Wolfe line search . Approximate Wolfe line search . Improved Wolfe line search

Mathematics Subject Classification (2000) 49M37. 90C30

1 Introduction

For solving large-scale unconstrained optimization problems the best very well known methods are the quasi-Newton Broyden-Fletcher-Goldfarb-Shanno (BFGS) [1-4] and the nonlinear conjugate gradient. The relationship between these two methods was given by Perry [5] and Shanno [6], who introduced the self-scaling memoryless BFGS method. For convex quadratic functions, if the line search is exact and if for the initial approximation to the Hessian matrix the identity matrix is used, then both the BFGS and the self-scaling memoryless BFGS will generate the same iterations as the conjugate gradient methods. For general nonlinear functions Shanno [6] proved that the conjugate gradient methods are precisely the BFGS quasi-Newton method, where the approximation to the inverse Hessian is restarted as the identity matrix at every step. Based on the self-scaling memoryless BFGS method by Perry and Shanno, Hager and Zhang [7, 8] and Dai and Kou [9] introduced two new conjugate gradient algorithms, called CG-DESCENT and CGOPT, respectively. Intensive numerical experiments proved that these two conjugate gradient methods are more efficient and more robust than the self-scaling memoryless BFGS method.

More exactly, if at every step the approximation to the inverse Hessian is restarted as a scaled identity matrix with a parameter, then the Perry-Shanno search direction is obtained which is dependent by the scaling parameter. Using an aggressive modification of the Perry-Shanno search direction by deleting its last term and selecting for the scaling parameter the value suggested by Oren and Spedicato [10], then the CG-DESCENT conjugate gradient is obtained. Some other conjugate gradient methods were suggested by considering other values for the scaling parameter, as those given by Oren [11], or Oren and Luenberger [12] or Al-Baali [13]. However, in this class of conjugate gradient algorithms, CG-DESCENT proved to be the best. Another approach, developed by Dai and Kou [9] was to seek the conjugate gradient direction that is closest to the Perry-Shanno search direction. Thus, Dai and Kou introduced a general family of conjugate gradient algorithms with a scaling parameter. If for this scaling parameter the value suggested by Oren and Luenberger is selected, than the CGOPT conjugate gradient algorithm is obtained. It is worth saying that the value of the scaling parameter suggested by Oren and Spedicatto and used to get CG-DESCENT conjugate gradient algorithm is obtained by minimizing the condition number of $H_k^{-1}H_{k+1}$, where H_k is the approximation to the inverse Hessian. On the other hand, the value of the scaling parameter given by Oren and Luenberger and used to get CGOPT conjugate gradient algorithm is obtained by reducing the condition number of the matrix $H_k^{1/2}\nabla^2 f(x_k)H_k^{1/2}$. Observe that these values for the scaling parameters are obtained by using the singular values of the corresponding matrices.

In this paper we present another approach to determine the value of the scaling parameter in the search direction which is closest to the Perry and Shanno's. In this approach we suggest three procedures for computing the scaling parameter based on the clustering the spectrum of the iteration matrix corresponding to the search direction, or based on the measure function of Byrd and Nocedal [14]. In the first procedure, the value of the scaling parameter is determined by using the determinant of the self-scaling memoryless BFGS iteration matrix. The second one is based

on the trace of the same matrix. Finally, in the third procedure the value of the scaling parameter is obtained by minimizing the measure function of Byrd and Nocedal. It is proved that the first procedure based on the determinant of the iteration matrix leads to the CGOPT conjugate gradient of Dai and Kou. The others are new.

As it is known, in conjugate gradient algorithms the procedure for the stepsize computation is crucial. It is common to see that in conjugate gradient algorithms the search directions tend to be poorly scaled and as a consequence the line search procedure must perform more function and gradient evaluations in order to obtain a suitable stepsize. In other words, in conjugate gradient algorithm the stepsizes differ from 1 in a very unpredictable way [15]. They can be larger or smaller than 1 depending on how the problem is scaled. This is in very sharp contrast to the Newton or quasi-Newton methods, including the limited memory methods, which accept the unit stepsize most of the time along the iterations, and therefore usually they require only few function evaluations per search direction. Subject to the stepsize computation, in order to get more efficient and robust conjugate gradient algorithms two new procedures based on the Wolfe conditions have been developed. The first is the approximate Wolfe line search introduced by Hager and Zhang in CG-DESCENT [7, 8]. The second one is the improved Wolfe line search given by Dai and Kou in CGOPT [9].

The structure of the paper is as follows. For completeness, in Section 2 we present the Dai-Kou family of self-scaling memoryless BFGS quasi-Newton methods [9]. Both the CG-DESCENT and the CGOPT algorithms are particularized from the search direction closest to the Perry and Shanno self-scaling memoryless BFGS search direction. The main contribution of this paper is given in Section 3, where three new procedures for computing the scaling parameter in the search direction closest to the Perry and Shanno's are developed. The first two are based on clustering the eigenvalues of the self-scaling memoryless BFGS iteration matrix, and the third one is based on minimizing the measure function of Byrd and Nocedal. In Section 4 some comments on the approximate Wolfe line search and on the improved Wolfe line search are described. Both for strongly convex functions or for general nonlinear functions, the convergence of the self-scaling memoryless BFGS algorithms with the improved Wolfe line search is proved under the classical assumptions. Section 5 is devoted to present the numerical results with these algorithms on a set of 80 large-scale unconstrained optimization test functions, of different structures and complexities, under the standard Wolfe line search, the approximate Wolfe line search by Hager and Zhang [7] and the improved Wolfe line search by Dai and Kou [9], respectively. The numerical comparisons among these algorithms showed that for all procedures for stepsize computation, the performances of the suggested algorithms based on clustering the eigenvalues of the self-scaling memoryless BFGS iteration matrix H_{k+1} using the determinant or the trace of H_{k+1} are similar. For the standard, the approximate or the improved Wolfe line searches, the self-scaling memoryless BFGS algorithms based on minimizing the measure function of Byrd and Nocedal is more efficient and more robust versus the algorithms based on the determinant or on the trace of H_{k+1}. Under the standard, the approximate or the improved Wolfe line search, the performances of the suggested algorithms based on the determinant or on the trace of the iteration matrix H_{k+1}, or based on minimizing the measure function defined by Byrd and Nocedal, are substantially better that those of CG-DESCENT with Wolfe or with the approximate Wolfe line search.

2 The Dai-Kou Family of Self-Scaling Memoryless Broyden-Fletcher-Goldfarb-Shanno Quasi-Newton Methods

Let us consider the unconstrained optimization problem

$$\min_{x\in\mathbb{R}^n} f(x), \tag{1}$$

where $f:\mathbb{R}^n \to \mathbb{R}$ is a real valued, twice differentiable function f of n variables with known gradient $g(x)=\nabla f(x)$. Suppose that f satisfies the following assumption.

Assumption 2.1: f is bounded below and its gradient g is Lipschitz continuous, namely there exists a constant $L>0$ such that $\|g(x)-g(y)\| \le L\|x-y\|$, for any $x,y\in\mathbb{R}^n$, where $\|.\|$ stands for the Euclidian norm.

As we know, given the initial point x_0, the iterations of the quasi-Newton methods are generated in the following form

$$x_{k+1} = x_k + \alpha_k d_k, \quad k=0,1,\ldots, \tag{2}$$

where $d_k \in \mathbb{R}^n$ is the search direction along which the values of function f are reduced, and $\alpha_k \in \mathbb{R}$ is the stepsize determined by a line search procedure. Often, for determination of the stepsize, the standard Wolfe line search conditions [16, 17] are used:

$$f(x_k+\alpha_k d_k) \le f(x_k) + \rho\alpha_k g_k^T d_k, \tag{3}$$

$$g(x_k+\alpha_k d_k)^T d_k \ge \sigma g_k^T d_k, \tag{4}$$

where $0<\rho<\sigma<1$. The search directions in the quasi-Newton methods are computed as

$$d_k = -\bar{H}_k g_k, \tag{5}$$

where $\bar{H}_k \in \mathbb{R}^{n\times n}$ is an approximation to the inverse Hessian of the minimizing function. At the iteration k, the approximation $\bar{H}_k$ to the inverse Hessian is updated to achieve $\bar{H}_{k+1}$ as a new approximation to the inverse Hessian in such a way that $\bar{H}_{k+1}$ satisfies a particular equation, namely the secant equation, which includes the second order information. The most used equation is the standard secant equation

$$\bar{H}_{k+1} y_k = s_k, \tag{6}$$

where $s_k = x_{k+1} - x_k$ and $y_k = g_{k+1} - g_k$.
Given the initial approximation $\bar{H}_0$ to the inverse Hessian, as an arbitrary symmetric and positive definite matrix, the most known quasi-Newton updating formula is the BFGS update

$$\bar{H}_{k+1} = \bar{H}_k - \frac{s_k y_k^T \bar{H}_k + \bar{H}_k y_k s_k^T}{y_k^T s_k} + \left(1+\frac{y_k^T \bar{H}_k y_k}{y_k^T s_k}\right)\frac{s_k s_k^T}{y_k^T s_k}. \tag{7}$$

The self-scaling memoryless BFGS method of Perry [5] and Shanno [6] is obtained by updating the scaled identity matrix $(1/\tau_k)I$ by the BFGS updating formula (7), i.e. by considering in (7) $\bar{H}_k = (1/\tau_k)I,$ where I is the $n \times n$ identity matrix and τ_k is a scaling parameter. Therefore, the search direction in the self-scaling memoryless BFGS method is computed as

$$d_{k+1} = -H_{k+1} g_{k+1}, \tag{8}$$

where

$$H_{k+1} = \frac{1}{\tau_k}\left(I - \frac{s_k y_k^T + y_k s_k^T}{y_k^T s_k} \right) + \left(1 + \frac{1}{\tau_k} \frac{\|y_k\|^2}{y_k^T s_k} \right) \frac{s_k s_k^T}{y_k^T s_k}, \tag{9}$$

and τ_k is the scaling parameter. Now, substituting (9) in (8), the self-scaling memoryless BFGS search direction of Perry and Shanno (with a multiplier difference) is obtained as

$$d_{k+1}^{PS} = -g_{k+1} + \left[\frac{g_{k+1}^T y_k}{y_k^T s_k} - \left(\tau_k + \frac{\|y_k\|^2}{y_k^T s_k} \right) \frac{g_{k+1}^T s_k}{y_k^T s_k} \right] s_k + \frac{g_{k+1}^T s_k}{y_k^T s_k} y_k. \tag{10}$$

Thus, subject to the parameter $\tau_k,$ a family of the Perry-Shanno self-scaling memoryless BFGS quasi-Newton methods is obtained. To this end the following particularizations of the search direction d_{k+1}^{PS} may be considered.

1) Having in view that $s_k = \alpha_k d_k,$ the deletion of the last term in (10) gives the following search direction

$$d_{k+1} = -g_{k+1} + \left[\frac{g_{k+1}^T y_k}{y_k^T d_k} - \left(\tau_k + \frac{\|y_k\|^2}{y_k^T s_k} \right) \frac{g_{k+1}^T s_k}{y_k^T d_k} \right] d_k. \tag{11}$$

Now, if τ_k is selected as the value given by Oren and Spedicato [10]

$$\tau_k^{OS} = \frac{\|y_k\|^2}{y_k^T s_k}, \tag{12}$$

then (11) reduces to the very well known conjugate gradient algorithm CG-DESCENT proposed by Hager and Zhang [7]:

$$d_{k+1} = -g_{k+1} + \beta_k^{CG-DESCENT} d_k, \tag{13}$$

where

$$\beta_k^{CG-DESCENT} = \frac{g_{k+1}^T y_k}{y_k^T d_k} - 2 \frac{\|y_k\|^2}{y_k^T s_k} \frac{g_{k+1}^T s_k}{y_k^T d_k}. \tag{14}$$

To establish the global convergence, for general nonlinear functions, the conjugate gradient parameter (14) is truncated as

$$\beta_k^{CG-DESCENT+} = \max\left\{ \frac{g_{k+1}^T y_k}{y_k^T d_k} - 2\frac{\|y_k\|^2}{y_k^T s_k}\frac{g_{k+1}^T s_k}{y_k^T d_k}, \frac{-1}{\|d_k\|\min\{\eta, \|g_k\|\}} \right\}, \tag{15}$$

where $\eta > 0$ is a constant ($\eta = 0.0001$) (see [7]).
The numerical experiments showed that CG-DESCENT is more efficient and more robust than the self-scaling memoryless BFGS method given by (10) [9].
Some other proposals for the parameter τ_k in (11) were given by Oren and Luenberger [12] as

$$\tau_k^{OL} = \frac{y_k^T s_k}{\|s_k\|^2} \tag{16}$$

and by Al-Baali [13]

$$\tau_k^{B1} = \min\left\{1, \frac{\|y_k\|^2}{y_k^T s_k}\right\} \quad \text{and} \quad \tau_k^{B2} = \min\left\{1, \frac{y_k^T s_k}{\|s_k\|^2}\right\}. \tag{17}$$

For a general nonlinear convex objective function, Nocedal and Yuan [18] proved the global convergence of the self-scaling BFGS method with τ_k given by (16) and with Wolfe line search. They also presented results indicating that the unscaled BFGS method in general is superior to the self-scaling BFGS with τ_k given by (16). Also, the proposals (17) for τ_k, analyzed by Al-Baali, ensure global convergence of the self-scaling BFGS method which is competitive with the unscaled BFGS method.

2) Observe that (10) is a three-term conjugate gradient algorithm. A more reasonable way to deal with the last term in (10) was suggested by Dai and Kou [9] who proposed to seek the search direction as a vector on the manifold $S_{k+1} = \{-g_{k+1} + \beta d_k : \beta \in \mathbb{R}\}$ that is closest to d_{k+1}^{PS}. The search direction in S_{k+1} closest to d_{k+1}^{PS} is obtained as solution of the following minimization problem

$$d_{k+1} = \arg\min\left\{\left\|d - d_{k+1}^{PS}\right\|_2 : d \in S_{k+1}\right\}: \tag{18}$$

which is

$$d_{k+1} = -g_{k+1} + \beta_k(\tau_k) d_k, \tag{19}$$

where

$$\beta_k(\tau_k) = \frac{g_{k+1}^T y_k}{y_k^T d_k} - \left(\tau_k + \frac{\|y_k\|^2}{y_k^T s_k} - \frac{s_k^T y_k}{\|s_k\|^2}\right)\frac{g_{k+1}^T s_k}{y_k^T d_k}. \tag{20}$$

In order to avoid the non-convergence of the algorithm, similarly to Gilbert and Nocedal [19], who proved the global convergence of the PRP methods for general nonlinear functions by restricting $\beta_k \geq 0$, (20) is truncated, being replaced by

$$\beta_k^+(\tau_k) = \max\left\{\beta_k(\tau_k), \eta\frac{g_{k+1}^T d_k}{\|d_k\|^2}\right\}, \tag{21}$$

where $\eta \in [0,1)$ is a parameter. ($\eta = 0.5$) Thus, the family of Dai-Kou self-scaling memoryless BFGS quasi-Newton methods is obtained. The following result shows that the search direction given by (19) and (20) satisfies the Dai and Liao conjugacy condition [20].

Proposition 2.1 *The search direction (19), where the parameter* $\beta_k(\tau_k)$ *is determined by (20) satisfies the Dai and Liao conjugacy condition* $d_{k+1}^T y_k = -t_k(g_{k+1}^T s_k)$, *where*

$$t_k = \tau_k + \frac{\|y_k\|^2}{y_k^T s_k} - \frac{s_k^T y_k}{\|s_k\|^2}$$

for any $k \geq 0$.

Proof From (19) and (20) by direct computation we get

$$d_{k+1}^T y_k = -\left(\tau_k + \frac{\|y_k\|^2}{y_k^T s_k} - \frac{s_k^T y_k}{\|s_k\|^2}\right)(g_{k+1}^T s_k) \equiv -t_k(g_{k+1}^T s_k). \qquad \blacksquare$$

Proposition 2.1 is a generalization of the result of Shanno [6] who proved that the conjugate gradient methods are precisely the BFGS quasi-Newton method, where the approximation to the inverse Hessian is restarted as the identity matrix at every step. In our case, at every step the approximation to the inverse Hessian is restarted as a scaled identity matrix. Therefore, (2), (19) and (20) is a conjugate gradient algorithm.

It is worth mentioning that if τ_k in (20) is selected as τ_k^{OL}, then the CGOPT conjugate gradient algorithm of Dai and Kou [9] is obtained, where in this case the search direction is computed as

$$d_{k+1} = -g_{k+1} + \beta_k^{CGOPT} d_k, \tag{22}$$

where

$$\beta_k^{CGOPT} = \frac{g_{k+1}^T y_k}{y_k^T d_k} - \frac{\|y_k\|^2}{y_k^T s_k}\frac{g_{k+1}^T s_k}{y_k^T d_k}. \tag{23}$$

Observe that the difference between the conjugate gradient parameters of CG-DESCENT given by (14) and of the CGOPT given by (23) is the absence of the constant factor 2 in the second term of the parameter from (23). Again, the numerical experiments showed that CGOPT performs more efficiently than the self-scaling memoryless BFGS method given by (10) [9]. Observe that if the line search is exact, i.e. $g_{k+1}^T s_k = 0$, then the second term in (20), (or in (14), or (23)) is missing and the search direction reduces to that of Hestenes and Stiefel [21].

Dai and Kou [9, Lemma 2.1] proved that if $y_k^T d_k > 0$, than the search direction given by (19) and (20) satisfies

$$g_{k+1}^T d_{k+1} \le -\min\left\{\tau_k \frac{\|s_k\|^2}{y_k^T s_k}, \frac{3}{4}\right\}\|g_{k+1}\|^2. \tag{24}$$

More general, if function f is continuously differentiable and bounded below, and its gradient g is Lipschitz continuous, then Dai and Kou [9, Lemma 2.2] proved that the search direction (19) where τ_k in (20) is chosen to be any of τ_k^{OS}, τ_k^{OL}, τ_k^{B1} or τ_k^{B2} and $y_k^T s_k > 0$, then $g_{k+1}^T d_{k+1} \le -c\|g_{k+1}\|^2$, for some positive constant $c > 0$.

Dai and Kou [9] implemented the algorithm (2), (19) and (20) endowed with two ingredients which improve its performances. The first ingredient is an improved Wolfe line search, which avoid the numerical drawback of the first Wolfe line search condition and guarantee the global convergence of the algorithm. The second one is an adaptive restart of the algorithm along the negative gradient based on how the minimizing function is close to some quadratic function. The numerical experiments with this family of self-scaling memoryless BFGS methods, given by Dai and Kou [9], where the parameter τ_k in (20) is chosen as τ_k^{OS}, τ_k^{OL}, τ_k^{B1} or τ_k^{B2}, proved that the selection τ_k^{OL} of τ_k is the most efficient one. With this selection of the parameter τ_k Dai and Kou [9] showed that the algorithm (2), (19) and (20) with improved Wolfe line search is more efficient and more robust than CG-DESCENT.

3 New Conjugate Gradient Algorithms based on Self-Scaling Memoryless Broyden-Fletcher-Goldfarb-Shanno Quasi-Newton Algorithms

The Dai-Kou family of the self-scaling memoryless BFGS quasi-Newton methods, given by (2) and (19) with (20), depends by the scaling parameter τ_k. In this section we present three different ways to choose the scaling parameter τ_k. For the very beginning observe that H_{k+1} given by (9) is symmetric and positive definite. Therefore it has n positive eigenvalues. If $y_k^T s_k > 0$, which always is satisfied when the setpsize is determined by the Wolfe line search, then there exists a set of mutually orthogonal unit vectors $\{u_k^i\}_{i=1}^{i=n-2}$ such that

$$s_k^T u_k^i = y_k^T u_k^i = 0, \quad i = 1,\ldots,n-2,$$

which leads to

$$H_{k+1} u_k^i = \frac{1}{\tau_k} u_k^i, \quad i = 1,\ldots,n-2.$$

Thus, the vectors u_k^i, $i = 1,\ldots,n-2$, are the eigenvectors of H_{k+1} correspondent to the eigenvalues $1/\tau_k$. Therefore, H_{k+1} given by (9) has $n-2$ eigenvalues all equal to $1/\tau_k$. Let λ_k^{n-1} and λ_k^n be the two remaining eigenvalues of H_{k+1}.

As we know, in a small neighborhood of the current point, the nonlinear objective function *f* in the unconstrained optimization problem (1) behaves like a quadratic one for which the results from the linear conjugate gradient can be applied. For faster convergence of linear conjugate gradient algorithms some approaches can be considered as follows: the presence of isolated smallest and/or largest eigenvalues of the matrix H_{k+1} as well as gaps inside the eigenvalues

spectrum [22], clustering of the eigenvalues about one point [23] or about several points [24], or preconditioning [25]. If the matrix has a number of certain distinct eigenvalues contained in m disjoint intervals of very small length, then the linear conjugate gradient method will produce a very small residual after m iterations [26]. This is an important property of linear conjugate gradient method and we try to use it in nonlinear case. Therefore, we consider the extension of the method of clustering the eigenvalues of the matrix defining the search direction from linear conjugate gradient algorithm to nonlinear case. For this we need to compute the determinant and the trace of the self-scaling memoryless BFGS matrix H_{k+1}. After some simple algebraic manipulation from (9) we get

$$\det(H_{k+1}) = \frac{1}{\tau_k^{n-1}} \frac{\|s_k\|^2}{y_k^T s_k} \tag{25}$$

and

$$tr(H_{k+1}) = \frac{n-2}{\tau_k} + \left(1 + \frac{1}{\tau_k} \frac{\|y_k\|^2}{y_k^T s_k}\right) \frac{\|s_k\|^2}{y_k^T s_k}. \tag{26}$$

The extension of the clustering the eigenvalues from the linear case to the nonlinear one can be achieved in the following two ways.

1) The first extension of the clustering the eigenvalues from the linear case to the nonlinear one is based on the determinant of the self-scaling memoryless BFGS matrix H_{k+1} given by (9). The idea of this variant of the self-scaling memoryless BFGS algorithm is to determine τ_k by clustering the eigenvalues of H_{k+1} in a point. Since H_{k+1} has $n-2$ eigenvalues all equal to $1/\tau_k$, then imposing that the remaining eigenvalues to have the same value, i.e. $\lambda_k^{n-1} = \lambda_k^n = 1/\tau_k$, from the equality

$$\det(H_{k+1}) = \frac{1}{\tau_k^n},$$

where $\det(H_{k+1})$ is given by (25), we get

$$\tau_k = \frac{y_k^T s_k}{\|s_k\|^2}. \tag{27}$$

From (23) we see that this choice of τ_k, based on determinant, is exactly the choice proposed by Dai and Kou in their CGOPT algorithm [9], using the Oren and Luenberger choice [12]: $\frac{y_k^T s_k}{s_k^T B_k s_k}$, with $B_k = H_k^{-1}$ for the BFGS method, where H_k is the identity matrix. Substituting (27) into (20) leads to the conjugate gradient parameter

$$\beta_k^{DE} = \frac{g_{k+1}^T y_k}{y_k^T d_k} - \frac{\|y_k\|^2}{y_k^T s_k} \frac{g_{k+1}^T s_k}{y_k^T d_k} \tag{28}$$

and from (21) the truncated form is obtained as

$$\beta_k^{DE+} = \max\left\{ \frac{g_{k+1}^T y_k}{y_k^T d_k} - \frac{\|y_k\|^2}{y_k^T s_k} \frac{g_{k+1}^T s_k}{y_k^T d_k}, \eta \frac{g_{k+1}^T d_k}{\|d_k\|^2} \right\}, \tag{29}$$

based on determinant. Observe that $\beta_k^{DE} = \beta_k^{CGOPT}$, i.e. the algorithm obtained by clustering the eigenvalues of the iteration matrix H_{k+1} is exactly the CGOPT algorithm of Dai and Kou. The formula (28) differs from (14) only with a constant coefficient in the second term of the Hager and Zhang method.

In the following we show that for strongly (uniformly) convex functions f the search directions (8) and (9), where τ_k is computed as in (27), satisfy the sufficient descent condition $g_k^T d_k \le -c\|g_k\|^2$ for any $k \ge 0$, where c is a positive constant. Recall that a differential function f is said to be strongly convex on a nonempty open convex set S if there exists a positive constant μ such that

$$(g(x) - g(y))^T (x - y) \ge \mu \|x - y\|^2, \text{ for any } x, y \in S,$$

where $g(x) = \nabla f(x)$.

Theorem 3.1 *Suppose that the Assumption 2.1 holds. For the method (2), (8) and (9), if f is a strongly convex function on the level set $S = \{x \in \mathbb{R}^n : f(x) \le f(x_0)\}$, and the stepsize α_k is determined by the Wolfe line search (3) and (4), then the search directions (8) and (9), where the parameter τ_k is computed as in (27), satisfy the sufficient descent condition $g_k^T d_k \le -c\|g_k\|^2$ for any $k \ge 0$, where c is a positive constant.*

Proof As we know H_{k+1} given by (9) has $n-2$ eigenvalues all equal to $1/\tau_k$, as well as λ_k^{n-1} and λ_k^n. Since $tr(H_{k+1})$ is equal to the summation of the eigenvalues of H_{k+1} and $\det(H_{k+1})$ is equal to the product of them, from (26) and (25) it is easy to see that

$$\lambda_k^{n-1} + \lambda_k^n = \left(1 + \frac{1}{\tau_k} \frac{\|y_k\|^2}{y_k^T s_k}\right) \frac{\|s_k\|^2}{y_k^T s_k} \tag{30}$$

and

$$\lambda_k^{n-1} \lambda_k^n = \frac{1}{\tau_k} \frac{\|s_k\|^2}{y_k^T s_k}. \tag{31}$$

Now, from the Assumption 2.1 we have $\|y_k\| \le L\|s_k\|$. On the other hand, from the strong convexity of function f on S we have $y_k^T s_k \ge \mu \|s_k\|^2$. Assume that $\lambda_k^n \le \lambda_k^{n-1}$. With these, from (31) and (30) we get

$$\lambda_k^n = \frac{1}{\tau_k}\frac{\|s_k\|^2}{y_k^T s_k}\frac{1}{\lambda_k^{n-1}} \geq \frac{1}{\tau_k}\frac{\|s_k\|^2}{y_k^T s_k}\frac{1}{(\lambda_k^{n-1}+\lambda_k^n)} = \frac{y_k^T s_k}{\tau_k(y_k^T s_k)+\|y_k\|^2}. \tag{32}$$

But, from (27)

$$\tau_k(y_k^T s_k) = \frac{(y_k^T s_k)^2}{\|s_k\|^2} \leq L^2\|s_k\|^2.$$

Therefore,

$$\lambda_k^n \geq \frac{\mu\|s_k\|^2}{L^2\|s_k\|^2 + L^2\|s_k\|^2} = \frac{\mu}{2L^2}. \tag{33}$$

Now, from (8) and (33), for all $k \geq 0,$ we have

$$d_{k+1}^T g_{k+1} = -g_{k+1}^T H_{k+1} g_{k+1} \leq -\lambda_k^n \|g_{k+1}\|^2 \leq -\frac{\mu}{2L^2}\|g_{k+1}\|^2,$$

i.e. the search direction (8), where τ_k is determined as in (27) satisfy the sufficient descent condition $g_k^T d_k \leq -c\|g_k\|^2$ with $c = \mu/(2L^2)$. ■

2) The second extension of the clustering the eigenvalues from the linear case to nonlinear one is based on the trace of the self-scaling memoryless BFGS matrix H_{k+1}. Again, the idea of this variant of the self-scaling memoryless BFGS algorithm is to determine τ_k by clustering the eigenvalues of H_{k+1} in a point, but this time using trace of H_{k+1}. Since H_{k+1} has $n-2$ eigenvalues, all equal to $1/\tau_k$, then imposing that the remaining eigenvalues to have the same value, i.e. $\lambda_k^{n-1} = \lambda_k^n = 1/\tau_k$, from the equality

$$tr(H_{k+1}) = \frac{n}{\tau_k}$$

where $tr(H_{k+1})$ is given by (26) we get

$$\tau_k = \left(2 - \frac{\|y_k\|^2\|s_k\|^2}{(y_k^T s_k)^2}\right)\frac{y_k^T s_k}{\|s_k\|^2}. \tag{34}$$

Now, substituting (34) into (20) leads to the conjugate gradient parameter

$$\beta_k^{TR} = \frac{g_{k+1}^T y_k}{y_k^T d_k} - \frac{y_k^T s_k}{\|s_k\|^2}\frac{g_{k+1}^T s_k}{y_k^T d_k}, \tag{35}$$

and from (21) the truncated form is obtained as

$$\beta_k^{TR+} = \max\left\{ \frac{g_{k+1}^T y_k}{y_k^T d_k} - \frac{y_k^T s_k}{\|s_k\|^2} \frac{g_{k+1}^T s_k}{y_k^T d_k}, \eta \frac{g_{k+1}^T d_k}{\|d_k\|^2} \right\}, \tag{36}$$

based on trace of H_{k+1}.

Theorem 3.2 *Suppose that the Assumption 2.1 holds. For the method (2), (8) and (9), if* f *is a strongly convex function on the level set* $S = \{x \in \mathbb{R}^n : f(x) \le f(x_0)\}$, *and the stepsize* α_k *is determined by the Wolfe line search (3) and (4), then the search directions (8) and (9), where the parameter* τ_k *is computed as in (34), satisfy the sufficient descent condition* $g_k^T d_k \le -c\|g_k\|^2$ *for any* $k \ge 0$, *where* c *is a positive constant.*

Proof The scaling parameter τ_k from (34) can be written as

$$\tau_k = 2\frac{y_k^T s_k}{\|s_k\|^2} - \frac{\|y_k\|^2}{y_k^T s_k}.$$

From (32) it is easy to see that

$$\lambda_k^n \ge \frac{\mu\|s_k\|^2}{3L^2\|s_k\|^2 + L^2\|s_k\|^2} = \frac{\mu}{4L^2}. \tag{37}$$

Now, from (8) and (37), for all $k \ge 0$, we have

$$d_{k+1}^T g_{k+1} = -g_{k+1}^T H_{k+1} g_{k+1} \le -\lambda_k^n \|g_{k+1}\|^2 \le -\frac{\mu}{4L^2}\|g_{k+1}\|^2,$$

i.e. the search directions (8), where τ_k is determined as in (34) satisfy the sufficient descent condition $g_k^T d_k \le -c\|g_k\|^2$ with $c = \mu/(4L^2)$. ■

3) Another possibility to determine a value for the scaling parameter τ_k in the self-scaling memoryless BFGS method, we consider in this paper, is to minimize a combination of the determinant and the trace of the iteration matrix H_{k+1} given by (9). Byrd and Nocedal [14] introduced such a combination of $\det(H_{k+1})$ and $tr(H_{k+1})$ as the function:

$$\varphi(H_{k+1}) = tr(H_{k+1}) - \ln(\det(H_{k+1})), \tag{38}$$

where $\ln(.)$ denotes the natural logarithm, known as the measure function. Since H_{k+1} is positive definite, it follows that $\varphi(H_{k+1})$ is well defined. Fletcher [27] observed that the BFGS formula can be derived by a variational argument using function φ. This is an elegant and efficient tool for analyzing the global behavior of quasi-Newton methods and now we intend to use it to generate new and efficient algorithms for unconstrained optimization. Observe that function φ works simultaneously with trace and determinant, thus simplifying the analysis of the quasi-Newton methods. In fact, this function is a measure of matrices involving all the eigenvalues of the iteration matrix, not only the smallest and the largest as it is traditionally used in the analysis

of the quasi-Newton methods based on the condition number of matrices (see [28–32]). Observe that this function is strictly convex on the set of symmetric and positive definite matrices and it is minimized by $H_{k+1} = I$. Besides, it becomes unbounded as H_{k+1} becomes singular or infinite and therefore it works as a barrier function that keeps H_{k+1} positive definite.

Therefore, the idea of this variant of the self-scaling memoryless BFGS algorithm is to determine τ_k by minimizing the measure function $\varphi(H_{k+1})$ of Byrd and Nocedal, defined in (38). From (25) and (26) we have

$$\varphi(H_{k+1}) = \frac{n-2}{\tau_k} + \frac{\|s_k\|^2}{y_k^T s_k} + \frac{1}{\tau_k}\frac{\|y_k\|^2\|s_k\|^2}{(y_k^T s_k)^2} + (n-1)\ln(\tau_k) - \ln\left(\frac{\|s_k\|^2}{y_k^T s_k}\right). \tag{39}$$

It is easy to see that the solution of the problem $\min \varphi(H_{k+1})$, where $\varphi(H_{k+1})$ is given by (39), is the solution of the equation

$$\frac{\partial \varphi(H_{k+1})}{\partial \tau_k} = -\frac{n-2}{\tau_k^2} - \frac{1}{\tau_k^2}\frac{\|y_k\|^2\|s_k\|^2}{(y_k^T s_k)^2} + (n-1)\frac{1}{\tau_k} = 0,$$

which is

$$\tau_k = \frac{n-2}{n-1} + \frac{1}{n-1}\frac{\|y_k\|^2\|s_k\|^2}{(y_k^T s_k)^2} > 0. \tag{40}$$

The measure function $\varphi(H_{k+1})$ of Byrd and Nocedal is a special combination of the determinant and of the trace of the iteration matrix H_{k+1}. Observe that the determinant in $\varphi(H_{k+1})$ is under the natural logarithm. Since H_{k+1} is positive definite, it follows that $tr(H_{k+1}) > 0$. It is quite possible that along the iterations $\ln(\det(H_{k+1})) < 0$, this being more harmful for minimization of $\varphi(H_{k+1})$. Therefore a variant of the algorithm based on minimizing the measure function $\varphi(H_{k+1})$ is to update the value of the parameter τ_k using (40) only when $\det(H_{k+1}) > 1$, otherwise τ_k remains to be updated by (34).

Hence, to minimize the measure function $\varphi(H_{k+1})$ we consider the following procedure. From (25) compute $\det(H_{k+1})$ with τ_k given by (34). With this value of $\det(H_{k+1})$ compute:

$$\bar{\tau}_k = \begin{cases} \left(2 - \dfrac{\|y_k\|^2\|s_k\|^2}{(y_k^T s_k)^2}\right)\dfrac{y_k^T s_k}{\|s_k\|^2}, & \text{if} \quad \det(H_{k+1}) \le 1, \\ \dfrac{n-2}{n-1} + \dfrac{1}{n-1}\dfrac{\|y_k\|^2\|s_k\|^2}{(y_k^T s_k)^2} & \text{if} \quad \det(H_{k+1}) > 1. \end{cases} \tag{41}$$

Now, considering $\tau_k = \bar{\tau}_k$ into (20) we get

$$\beta_k^{FI} = \frac{g_{k+1}^T y_k}{y_k^T d_k} - \left(\bar{\tau}_k + \frac{\|y_k\|^2}{y_k^T s_k} - \frac{s_k^T y_k}{\|s_k\|^2}\right)\frac{g_{k+1}^T s_k}{y_k^T d_k} \tag{42}$$

and its truncated value

$$\beta_k^{FI+} = \max\left\{ \frac{g_{k+1}^T y_k}{y_k^T d_k} - \left(\bar{\tau}_k + \frac{\|y_k\|^2}{y_k^T s_k} - \frac{s_k^T y_k}{\|s_k\|^2} \right) \frac{g_{k+1}^T s_k}{y_k^T d_k}, \eta \frac{g_{k+1}^T d_k}{\|d_k\|^2} \right\}, \tag{43}$$

based on minimizing the measure function of Byrd and Nocedal. Besides, based on the insights gained from the example given by Powell [33], we constrain β_k^{FI+} parameter to be positive, i.e.

$$\beta_k^{FI+} = \max\{\beta_k^{FI+}, 0\}. \tag{44}$$

Theorem 3.3 *Suppose that the Assumption 2.1 holds. For the method (2), (8) and (9), if* f *is a strongly convex function on the level set* $S = \{x \in \mathbb{R}^n : f(x) \le f(x_0)\}$, *and the stepsize* α_k *is determined by the Wolfe line search (3) and (4), then the search directions (8) and (9), where the parameter* τ_k *is computed as in (40), satisfy the sufficient descent condition* $g_k^T d_k \le -c\|g_k\|^2$ *for any* $k \ge 0$, *where* c *is a positive constant.*

Proof Having in view that $\|y_k\| \le L\|s_k\|$ and $y_k^T s_k \ge \mu\|s_k\|^2$, following the same procedure as in previous theorems, from (40) the quantity $\tau_k(y_k^T s_k)$ in (32) can be estimated as

$$\begin{aligned} \tau_k(y_k^T s_k) &= \frac{n-2}{n-1}(y_k^T s_k) + \frac{1}{n-1}\frac{\|y_k\|^2\|s_k\|^2}{y_k^T s_k} \le y_k^T s_k + \frac{\|y_k\|^2\|s_k\|^2}{y_k^T s_k} \\ &\le \|y_k\|\|s_k\| + \frac{\|y_k\|^2\|s_k\|^2}{\mu\|s_k\|^2} \le \left(L + \frac{L^2}{\mu}\right)\|s_k\|^2. \end{aligned} \tag{45}$$

Therefore, from (32), using (45) we get

$$\lambda_k^n \ge \frac{y_k^T s_k}{\tau_k(y_k^T s_k) + \|y_k\|^2} \ge \frac{\mu^2}{L^2 + \mu(L + L^2)}. \tag{46}$$

Now, from (8) and (46), for all $k \ge 0$, we have

$$d_{k+1}^T g_{k+1} = -g_{k+1}^T H_{k+1} g_{k+1} \le -\lambda_k^n \|g_{k+1}\|^2 \le -\frac{\mu^2}{L^2 + \mu(L + L^2)}\|g_{k+1}\|^2,$$

i.e. the search directions (8), where τ_k is determined as in (40) satisfy the sufficient descent condition $g_k^T d_k \le -c\|g_k\|^2$ with $c = \mu^2 / [L^2 + \mu(L + L^2)]$. ■

With these developments the following general self-scaling memoryless BFGS quasi-Newton algorithm may be presented.

Algorithm CGSSML (Conjugate Gradient Self-Scaling MemoryLess BFGS algorithm)

1. Initialization. Choose an initial point $x_0 \in \mathbb{R}^n$. Choose the constants σ, ρ with $0 < \rho < \sigma < 1$ and $\varepsilon > 0$ sufficiently small. Compute $g_0 = \nabla f(x_0)$. Set $d_0 = -g_0$ and $k = 0$
2. Test a criterion for stopping the iterations. For example, if $\|g_k\|_\infty < \varepsilon$, then stop the iterations
3. Compute the stepsize $\alpha_k > 0$ using the Wolfe line search conditions, or some modifications of them
4. Update the variables $x_{k+1} = x_k + \alpha_k d_k$ and compute f_{k+1} and g_{k+1}
5. Compute the scaling parameter τ_k using clustering the eigenvalues of the iteration matrix, or by minimizing the measure function of Byrd and Nocedal
6. Compute the parameter β_k according the values of parameter τ_k
7. Update the search direction $d_{k+1} = -g_{k+1} + \beta_k d_k$
8. Restart criterion. If $\left|g_{k+1}^T g_k\right| > 0.2\|g_{k+1}\|^2$ then set $d_{k+1} = -g_{k+1}$
9. Set $k = k+1$ and go to step 2

For computing the stepsize α_k in step 3 of the algorithm, the Wolfe line search (3) and (4) or the approximate Wolfe line search of Hager and Zhang [7, 8], or the improved Wolfe line search of Dai and Kou [9] may be implemented, as it is described in the next section. Observe that in step 5 the parameter τ_k may be computed using the clustering the eigenvalues of H_{k+1} by the determinant of H_{k+1} (27), or by the trace of H_{k+1} (34), or by minimizing the measure function of Byrd and Nocedal (41). In our algorithm, when the Powell restart condition is satisfied (step 8), then we restart the algorithm with the negative gradient $-g_{k+1}$. Some other restarting procedures may be implemented in CGSSML, like $d_{k+1}^T g_{k+1} \geq 10^{-3}\|d_{k+1}\|\|g_{k+1}\|$ of Birgin and Martínez [34], or the adaptive restarting strategy of Dai and Kou [9], but we are interesting to see the performances of CGSSML implementing the Powell restarting technique.

4 Line Search in CGSSML and Convergence Analysis

As it is known the stepsize is crucial in the efficiency of any line search algorithm like CGSSML. Usually, the Wolfe line search (3) and (4) are implemented in line search algorithms. However, in order to improve the performances of the line search algorithms the approximate Wolfe and the improved Wolfe line searches were introduced.
Hager and Zhang [7, 8] introduced the *approximate Wolfe line search*:

$$\sigma d_k^T g_k \leq d_k^T g_{k+1} \leq (2\rho - 1) d_k^T g_k, \tag{47}$$

where $0 < \rho < 1/2$ and $\rho < \sigma < 1$. This approximate line search is implemented in the CG-DESCENT algorithm. The first inequality in (47) is the same as (4). When f is quadratic, the second inequality in (47) is equivalent to (3). As shown by Hager and Zhang [7], the first Wolfe condition (3) limits the accuracy of a conjugate gradient method to the order of the square root of the machine precision, while with the approximate Wolfe line search we can achieve accuracy to the order of the machine precision. In practical computations, the first Wolfe condition (3) may never be satisfied because of the numerical errors, even for tinny values for ρ. In order to avoid

the numerical drawback of the Wolfe line search, Hager and Zhang [7] introduced a combination of the original Wolfe conditions and the approximate Wolfe conditions (47). Their line search is working very well in numerical computations, but cannot guarantee the global convergence of the algorithm in theory. Details on the global convergence of the CG-DESCENT algorithm with the approximate Wolfe line search are given by Hager and Zhang [7, 8].

In order to overcome the deficiencies of the approximate Wolfe line search, Dai and Kou [9] introduced the so called *improved Wolfe line search*: given a constant parameter $\varepsilon > 0$, a positive sequence $\{\eta_k\}$ satisfying $\sum_{k\geq 1}\eta_k < \infty$ as well as the parameters ρ and σ satisfying $0 < \rho < \sigma < 1$, Dai and Kou proposed the following modified Wolfe condition

$$f(x_k + \alpha d_k) \leq f(x_k) + \min\left\{\varepsilon\left|g_k^T d_k\right|, \rho\alpha g_k^T d_k + \eta_k\right\}. \tag{48}$$

The line search satisfying (48) and (4) is called the improved Wolfe line search. If f is continuously differentiable and bounded below, the gradient g is Lipschitz continuous and d_k is a descent direction (i.e. $g_k^T d_k < 0$), then there must exist a suitable stepsize satisfying (4) and (48), since they are weaker than the standard Wolfe conditions.

Under the improved Wolfe line search the search direction satisfies the Zoutendijk condition [35]. This condition has a crucial role in proving the convergence of the CGSSML algorithm with improved Wolfe line search.

Proposition 4.1 *Suppose that the Assumption 2.1 holds. Consider the method (2) where the search direction d_k satisfies the descent condition $g_k^T d_k < 0$ and the stepsize α_k satisfies the improved Wolfe line search conditions (4) and (48). Then*

$$\sum_{k\geq 1}\frac{(g_k^T d_k)^2}{\|d_k\|^2} < \infty. \tag{49}$$

Proof Like in [9] from the Lipschitz continuity and (4) we can write

$$L\alpha_k\|d_k\|^2 \geq (g_{k+1} - g_k)^T d_k \geq (\sigma - 1)g_k^T d_k.$$

Therefore

$$\alpha_k \geq \frac{\sigma - 1}{L}\frac{g_k^T d_k}{\|d_k\|^2}. \tag{50}$$

From (48) we have

$$f_{k+1} \leq f_k + \min\left\{\varepsilon\left|g_k^T d_k\right|, \rho\alpha g_k^T d_k + \eta_k\right\} \leq f_k + \rho\alpha g_k^T d_k + \eta_k.$$

Now, from (50) we get

$$f_k - f_{k+1} + \eta_k \geq c\frac{(g_k^T d_k)^2}{\|d_k\|^2}, \tag{51}$$

wher $c = \rho(1-\sigma)/L$. Since f is bounded below, summing (51) over k and having in view that the sequence $\{\eta_k\}$ satisfies $\sum_{k\geq 1}\eta_k < \infty$ it follows that (49) holds. ■

For strongly convex functions, the following theorem prove the global convergence of the algorithm (2), (19) and (20), where the scaling parameter τ_k is chosen as in (27), (34) or (40), under the improved Wolfe line search.

Theorem 4.1 *Suppose that the Assumption 2.1 holds. Consider the algorithm (2) in which the search direction is defined by (19) and (20), where τ_k is chosen to be as in (27), (34) or (40) and the stepsize α_k is determined by the improved Wolfe line search (4) and (48). If the function f is strongly convex, then the algorithm CGSSML is global convergent, i.e.* $\lim_{k\to\infty}\|g_k\| = 0$.

Proof From the Assumption 2.1 and the strong convexity of function f it follows that $\|y_k\| \leq L\|s_k\|$ and $y_k^T s_k \geq \mu\|s_k\|^2$. Therefore, for any τ_k given by (27), (34) and (40), there exists a positive constant c_τ such that $|\tau_k| \leq c_\tau$. From (19) and (20) we have

$$\begin{aligned}
\|d_{k+1}\| &\leq \|g_{k+1}\| + \frac{|g_{k+1}^T y_k|}{|y_k^T d_k|}\|d_k\| + \left|\tau_k + \frac{\|y_k\|^2}{y_k^T s_k} + \frac{s_k^T y_k}{\|s_k\|^2}\right| \frac{|g_{k+1}^T s_k|}{|y_k^T d_k|}\|d_k\| \\
&\leq \|g_{k+1}\| + \frac{\|g_{k+1}\| L\|s_k\|}{\mu\|s_k\|^2}\|s_k\| + \left(c_\tau + \frac{L^2\|s_k\|^2}{\mu\|s_k\|^2} + \frac{L\|s_k\|^2}{\|s_k\|^2}\right)\frac{\|g_{k+1}\|\|s_k\|}{\mu\|s_k\|^2}\|s_k\| \\
&\leq \left(1 + \frac{L^2 + 2\mu L + \mu c_\tau}{\mu^2}\right)\|g_{k+1}\|. \qquad (52)
\end{aligned}$$

On the other hand, since from the Theorems 3.1-3.3 for any τ_k given by (27), (34) or (40) the search direction (19) and (20) satisfies the sufficient descent condition, it follows that

$$\sum_{k\geq 1}\frac{\|g_k\|^4}{\|d_k\|^2} < \infty. \qquad (53)$$

By (52) and (53) we get

$$\sum_{k\geq 1}\|g_k\|^2 < \infty,$$

which implies that $\lim_{k\to\infty}\|g_k\| = 0$. ■

For general nonlinear functions, the global convergence of the algorithm (2) with (19) and (20), where the scaling parameter τ_k is chosen as in (27), (34) or (40) under the improved Wolfe line search follows the methodology given by Gilbert and Nocedal [19]. Dai and Kou [9] proved that if function f satisfies the Assumption 2.1 and there exists $\gamma > 0$ such that $\|g_k\| \geq \gamma$ for any $k \geq 1$, then for the family of conjugate gradient algorithms given by (2), in which the search

direction d_{k+1} is computed as in (19) and (20), and the stepsize α_k is determined by the improved Wolfe line search (4) and (48), then $d_k \neq 0$ and

$$\sum_{k\geq 2}\|u_k - u_{k-1}\|^2 < \infty, \tag{54}$$

where $u_k = d_k / \|d_k\|$.

This result, similar to Lemma 4.1 in [19], is used for proving the global convergence of the CGSSML algorithm with improved Wolfe line search. For this in the following proposition we prove that $\beta_k(\tau_k)$ in (20) has Property (*) defined in [19] (see also [36]).

Proposition 4.2 *Suppose that the Assumption 2.1 holds. Consider the family of conjugate gradient algorithms given by (2) in which the search direction d_{k+1} is computed as in (19) and (20) and the stepsize α_k is determined by the improved Wolfe line search (4) and (48). If the sequence $\{x_k\}$ generated by the algorithm CGSSML is bounded and if τ_k is chosen as in (27), (34) or (40), then $\beta_k(\tau_k)$ in (20) has Property (*).*

Proof The proof follows by contradiction, like in [9]. Suppose that $\|g_k\| \geq \gamma$ for any $k \geq 1$. From continuity of the gradient and the boundedness of $\{x_k\}$ it follows that there exists a positive constant $\bar{\gamma}$ such that

$$\|x_k\| \leq \bar{\gamma}, \quad \|g_k\| \leq \bar{\gamma}, \text{ for any } k \geq 1. \tag{55}$$

From (4) it follows that

$$g_{k+1}^T d_k \geq \sigma g_k^T d_k. \tag{56}$$

From Theorems 3.1 – 3.3 it follows that for any values of τ_k given by (27), or (34), or (40) we have $g_k^T d_k \leq -c\|g_k\|^2$, where c is a constant. Therefore, from (56) we get

$$d_k^T y_k = d_k^T g_{k+1} - d_k^T g_k \geq -(1-\sigma) d_k^T g_k \geq c(1-\sigma)\gamma^2. \tag{57}$$

Now, from (56) and since $g_k^T d_k < 0$, it follows that

$$\frac{\sigma}{\sigma - 1} \leq \frac{g_{k+1}^T d_k}{d_k^T y_k} \leq 1. \tag{58}$$

As it was proved in Theorems 3.1 – 3.3, it is easy to see that for any values of τ_k given by (27), or (34), or (40) there exists a positive constant c_τ such that

$$\left|\tau_k (y_k^T s_k)\right| \leq c_\tau \|s_k\|^2, \text{ for any } k \geq 1. \tag{59}$$

It is easy to see that $\beta_k(\tau_k)$ from (20) can be written as

$$\beta_k(\tau_k) = \frac{g_{k+1}^T y_k}{d_k^T y_k} - \left(1 - \frac{(d_k^T y_k)^2}{\|d_k\|^2 \|y_k\|^2}\right) \frac{\|y_k\|^2}{d_k^T y_k} \frac{g_{k+1}^T d_k}{d_k^T y_k} - \frac{\tau_k (y_k^T s_k)}{d_k^T y_k} \frac{g_{k+1}^T d_k}{d_k^T y_k}. \tag{60}$$

Observe that $\|y_k\| \le L\|s_k\|$ and $0 \le (d_k^T y_k)^2 \le \|d_k\|^2 \|y_k\|^2$, for any $k \ge 1$. Since by (55), $\|s_k\| = \|x_{k+1} - x_k\| \le \|x_{k+1}\| + \|x_k\| \le 2\bar{\gamma}$, using (57), (59) and (60) we get that there exists a constant $c_\beta > 0$ such that for any $k \ge 1$,

$$|\beta_k(\tau_k)| \le c_\beta \|s_k\|. \tag{61}$$

Now, like in [19] define $b = 2c_\beta \bar{\gamma}$ and $\lambda = 1/(2c_\beta^2 \bar{\gamma})$. From (61) and (55) it follows that for all $k \ge 1$ we have that

$$|\beta_k(\tau_k)| \le b, \tag{62}$$

and

$$\|s_k\| \le \lambda \quad \Rightarrow \quad |\beta_k(\tau_k)| \le \frac{1}{b}. \tag{63}$$

Therefore, (62) and (63) show that $\beta_k(\tau_k)$ defined by (20) has Property (*) (see [19]) ■

Theorem 4.2 *Suppose that the Assumption 2.1 holds. Consider the algorithm (2) in which the search direction is defined by (19) and (20), where* τ_k *is chosen to be as in (27), (34) or (40) and the stepsize* α_k *is determined by the improved Wolfe line search (4) and (48). If the sequence* $\{x_k\}$ *generated by the algorithm CGSSML is bounded, then the algorithm is global convergent, i.e.* $\liminf_{k\to\infty} \|g_k\| = 0.$

Proof By contradiction suppose that $\|g_k\| \ge \gamma$ for any $k \ge 1$. Since $-g_k^T d_k \ge c\|g_k\|^2$ for some positive constant $c > 0$ and for any $k \ge 1$, from Zoutendijk condition (49) it follows that

$$\|d_k\| \to +\infty. \tag{64}$$

From the continuity of the gradient, it follows that there exists a positive constant $\bar{\gamma}$ such that $\|g_k\| \le \bar{\gamma}$, for any $k \ge 1$. By (21), (64) means that $\beta_k(\tau_k)$ can only be less than $\eta \frac{g_{k+1}^T d_k}{\|d_k\|^2}$ for finite times, since otherwise, we have that

$$\|d_{k+1}\| = \left\| -g_{k+1} + \eta \frac{g_{k+1}^T d_k}{\|d_k\|^2} d_k \right\| \le (1+\eta)\|g_{k+1}\| \le (1+\eta)\bar{\gamma}$$

for infinite k 's and therefore we get a contradiction with (64). Hence, we can suppose that along the iterations $\beta_k^+(\tau_k) = \beta_k(\tau_k)$ for sufficiently large k. With this, from (54) and Lemma 4.2 in [19] and the boundedness of the sequence $\{x_k\}$, we get a contradiction similarly to the proof of Theorem 4.3 in [19]. This contradiction shows that $\liminf_{k\to\infty} \|g_k\| = 0.$ ■

5 Numerical Results and Comparisons

In this section we report some numerical results of the CGSSML algorithm for solving large-scale unconstrained optimization problems. Algorithm CGSSML was implemented by modifying the CG-DESCENT code (Fortran version 1.4) of Hager and Zhang [8] in order to incorporate the self-scaling memoryless BFGS algorithms in which the conjugate gradient parameter β_k in the search direction is computed by clustering the eigenvalues of the iteration matrix H_{k+1} or by minimizing the measure function of Byrd and Nocedal, presented in Section 3, and respectively with the standard Wolfe or with the approximate Wolfe or with the improved Wolfe line searches, discussed in Section 4.

Remark: Notice that, the algorithm CGSSML can be implemented by modifying the most recent C version 6.8 of CG-DESCENT code of Hager and Zhang [37], i.e. the preconditioned CG-DESCENT, where a limited memory conjugate gradient algorithm is used: L-CG-DESCENT. However, in this paper we do not implement CGSSML into the frame of L-CG-DESCENT because our interest here is to see the performances of β_k^{DE+}, β_k^{TR+} and β_k^{FI+} subject to different line search conditions, for solving large-scale unconstrained optimization problems, without any ingredients like: preconditioning, limited memory or adaptive restart of the algorithm. ■

The algorithms compared in this section are as follows: DESW, DEAW and DEIW, i.e. CGSSML algorithm with β_k^{DE+} given by (29) and with the standard Wolfe line search (3) and (4), with the approximate Wolfe line search of Hager and Zhang given by (47), with the improved Wolfe line search of Dai and Kou (48) and (4), respectively. TRSW, TRAW and TRIW, i.e. CGSSML algorithm with β_k^{TR+} given by (36) and with the standard Wolfe line search (3) and (4), with the approximate Wolfe line search of Hager and Zhang given by (47), with the improved Wolfe line search of Dai and Kou (48) and (4), respectively. FISW, FIAW and FIIW, i.e. CGSSML algorithm with β_k^{FI+} given by (44) and with the standard Wolfe line search (3) and (4), with the approximate Wolfe line search of Hager and Zhang given by (47), with the improved Wolfe line search of Dai and Kou (48) and (4), respectively. The code is compiled with f77 (default compiler settings) on a Workstation Intel Pentium 4 with 1.8 GHz. We selected a number of 80 large-scale unconstrained optimization test functions in generalized or extended form [38]. In this collection, some problems are quadratic and some of them are highly nonlinear. The problems are presented in extended (separable) or generalized (chained) form. The Hessian for the problems in extended form has a block-diagonal structure. On the other hand, the Hessian for the problems in generalized form has a banded structure with small bandwidth, often being tri- or penta- diagonal. For some other optimization problems from this set, the corresponding Hessian has a sparse structure or it is a dense (full) matrix. The vast majority of the optimization problems included in our collection described in [38] is taken from CUTEr collection [39]. For each test function, we have taken 10 numerical experiments with the number of variables $n = 1000, 2000, \ldots, 10,000$.
The parameters in the standard Wolfe line searches are $\rho = 0.0001$ and $\sigma = 0.8$. All the algorithms use the same stopping criterion $\|g_k\|_\infty \le 10^{-6}$, where $\|.\|_\infty$ is the maximum absolute component of a vector, or when the number of iterations exceeds 2000 iterations. The rest of parameters are the same defined in CG-DESCENT by Hager and Zhang [8] and Algorithm 4.1 of Dai and Kou [9]. In all algorithms, we considered in our numerical experiments, the Powell restart criterion, described in step 8 of the CGSSML algorithm, is used.

The algorithms which we compare in these numerical experiments find local solutions. Therefore, the comparisons of the algorithms are given in the following context. Let f_i^{ALG1} and f_i^{ALG2} be the optimal value found by ALG1 and ALG2 for problem $i = 1,\ldots,80,$ respectively. We say that, in the particular problem $i,$ the performance of ALG1 was better than the performance of ALG2 if:

$$\left| f_i^{ALG1} - f_i^{ALG2} \right| < 10^{-3} \tag{65}$$

and if the number of iterations (#iter), or the number of function-gradient evaluations (#fg), or the CPU time of ALG1 was less than the number of iterations, or the number of function-gradient evaluations, or the CPU time corresponding to ALG2, respectively. The performances of the algorithms are displayed by the Dolan and Moré performance profiles [40].

It is worth emphasizing that in our numerical experiments we compare algorithms included in CGSSML versus CG-DESCENT version 1.4. The idea was to see the performances of the algorithms using β_k^{DE+} given by (29), β_k^{TR+} given by (36), β_k^{FI+} given by (44) and $\beta_k^{CG-DESCENT+}$ given by (15) without any other ingredients included in the limited memory conjugate gradient algorithm proposed by Hager and Zhang [37], or in the CGOPT by Dai and Kou [9]. Our interests were to see the power the conjugate gradient parameters β_k^{DE+}, β_k^{TR+}, β_k^{FI+}, and $\beta_k^{CG-DESCENT+}$ with different line searches for solving large-scale unconstrained optimization problems.

In the first set of numerical experiments we compare the performance of CGSSML algorithms with standard Wolfe line search, namely DESW versus TRSW, DESW versus FISW and TRSW versus FISW for solving the set of problems considered in this numerical study. Figure 1 shows the Dolan and Moré CPU performance profiles of these algorithms. When comparing DESW versus TRSW subject to the CPU time metric we see that DESW is top performer. Comparing DESW versus TRSW (see Fig. 1), subject to the number of iterations, we see that DESW was better in 250 problems (i.e. it achieved the minimum number of iterations in 250 problems). TRSW was better in 143 problems and they achieved the same number of iterations in 370 problems, etc. Out of 800 problems, only for 763 problems does the criterion (65) holds. Observe that DESW and TRSW have similar performances, DESW being slightly more efficient and more robust. It seems that from the viewpoint of clustering of the eigenvalues of H_{k+1} using the determinant or the trace of the iteration matrix leads to algorithms with similar performances. From Figure 1 we see that FISW is top performer versus DESW and versus TRSW. This is because the FISW algorithm is based on an *ad hoc* procedure for minimizing a special combination of the determinant and of the trace of the iteration matrix H_{k+1}.

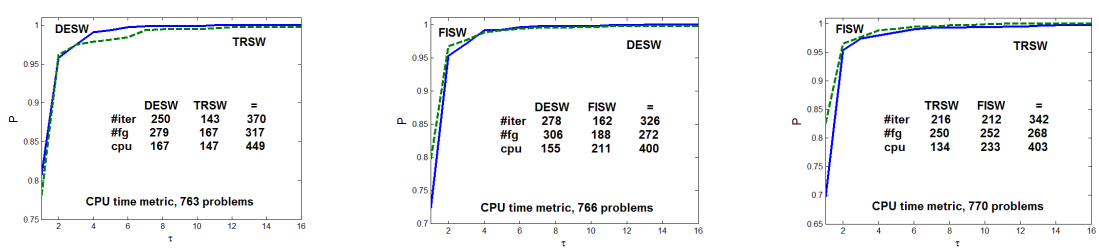

Fig. 1 Performance profiles of DESW versus TRSW, of DESW versus FISW and of TRSW versus FISW

In the second set of numerical experiments we compare DESW, TRSW and FISW versus CG-DESCENT (version 1.4) with truncated conjugate gradient parameter (15) and with standard Wolfe line search (3) and (4). CG-DESCENT was devised in order to ensure sufficient descent,

independent of the accuracy of the line search. In CG-DESCENT the search direction (13), where the conjugate gradient parameter is computed as in (14), satisfies the sufficient descent condition $g_k^T d_k \le -(7/8)\|g_k\|^2$, provided that $y_k^T d_k \ne 0$. The search directions in CG-DESCENT do not satisfy the conjugacy condition. When iterates jam the expression $\|y_k\|^2 (g_{k+1}^T s_k)/(y_k^T s_k)^2$ in (14) becomes negligible. If the minimizing function f is quadratic and the line search is exact, then CG-DESCENT reduces to the Hestenes and Stiefel algorithm [21]. Figure 2 presents the Dolan and Moré performance profiles of these algorithms. Form Figure 2 we see that DESW, TRSW and FISW are top performers versus CG-DESCENT and the differences are significant. Since all these algorithms use the same line search based on Wolfe conditions (3) and (4), it follows that DESW, TRSW and FISW generates a better search direction. Notice that the difference between DESW and CG-DESCENT is only in a constant coefficient of the second term of the Hager and Zhang method. Besides, the truncation mechanisms in these algorithms are different and this explains the differences between these algorithms.

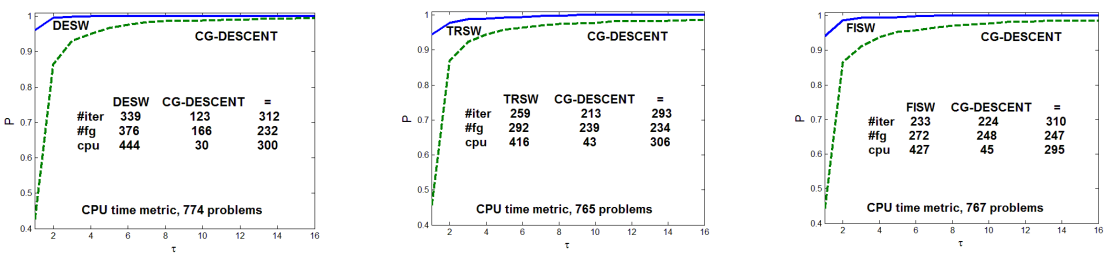

Fig. 2 Performance profiles of DESW, TRSW and FISW versus CG-DESCENT

In the third set of numerical experiments we compare the performance of CGSSML algorithms with approximate Wolfe line search, namely DEAW versus TRAW, DEAW versus FIAW and TRAW versus FIAW for solving the set of problems we considered in this numerical study. Figure 3 presents the Dolan and Moré performance profiles of these algorithms. From Figure 3 we see that both DEAW and TRAW have similar performances. However, FIAW is top performer versus both DEAW and TRAW.

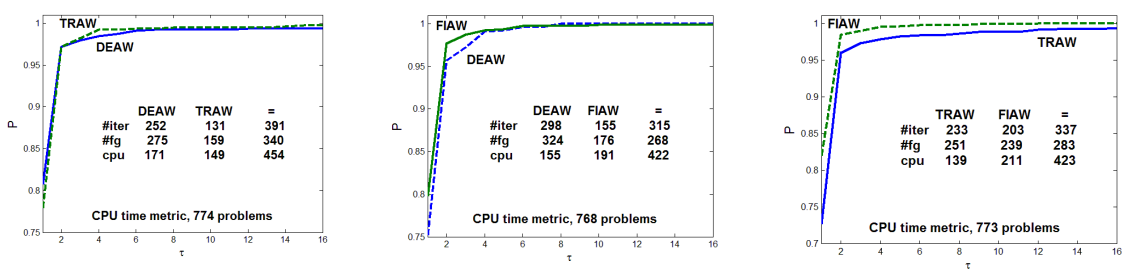

Fig. 3 Performance profiles of DEAW versus TRAW, of DEAW versus FIAW and of TRAW versus FIAW

In the fourth set of numerical experiment we compare DEAW, TRAW and FIAW versus CG-DESCENT with truncated conjugate gradient parameter (15) and with approximate Wolfe line search (47) (CG-DESCENTa). The CG-DESCENTa algorithm of Hager and Zhang [7, 8] implements advanced features of line search including: an approximation to the Wolfe line search

condition that can be evaluated with a greater accuracy, a special secant procedure that leads to a rapid and accurate reduction in the width of the interval bracketing the stepsize, and a quadratic step that retains the n -step quadratic convergence property of the algorithm [41]. Figure 4 shows the performances of these algorithms. Observe that all DEAW, TRAW and FIAW are top performers versus CG-DESCENTa. The greatest difference is between FIAW and CG-DESCENTa.

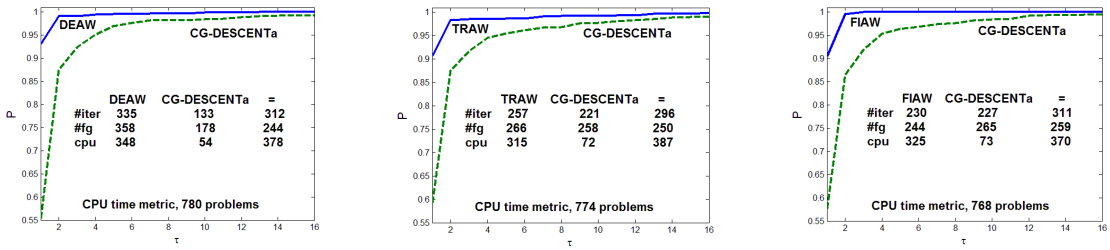

Fig. 4 Performance profiles of DEAW, TRAW and FIAW versus CG-DESCENTa

In the fifth set of numerical experiments we compare the performance of CGSSML algorithms with improved Wolfe line search, namely DEIW versus TRIW, DEIW versus FIIW and TRIW versus FIIW for solving the set of problems considered in this numerical study. Figure 5 presents the performances of these algorithms. Both DEIW and TRIW have similar performances. Observe that FIIW is slightly top performer versus DEIW and TRIW. However, they have similar performances, at least for this set of unconstrained optimization problems considered in this paper.

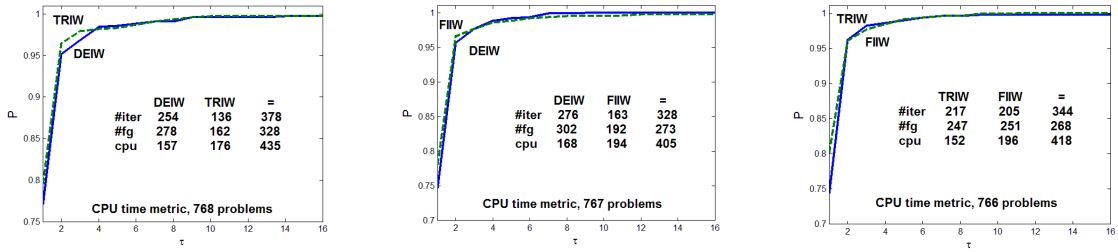

Fig. 5 Performance profiles of DEIW versus TRIW, of DEIW versus FIIW and of TRIW versus FIIW

In the sixth set of numerical experiments we compare DEIW, TRIW and FIIW versus CG-DESCENT with truncated conjugate gradient parameter (15) and with Wolfe line search (3) and (4). Figure 6 shows the performances of these algorithms. We see that DEIW, TRIW and FIIW are more efficient and more robust than CG-DESCENT with Wolfe line search. We see that using the determinant, the trace or the measure function of Byrd and Nocedal, we get conjugate gradient algorithms more efficient and more robust. All these methods are based on clustering the eigenvalues of the iteration matrix H_{k+1} given by (9) with corresponding values for τ_k.

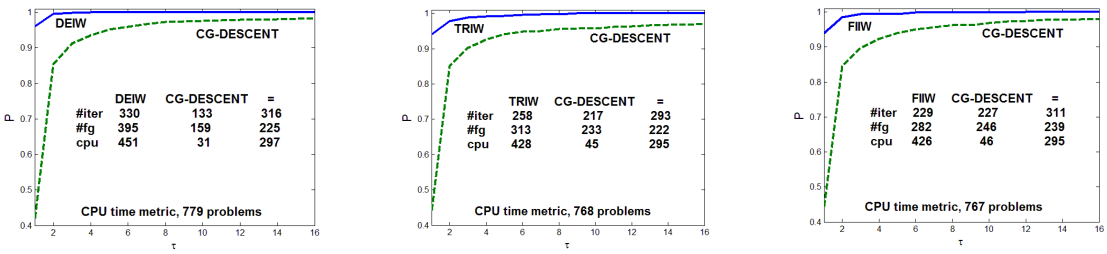

Fig. 6 Performance profiles of DEIW, TRIW and FIIW versus CG-DESCENT

In the next set of numerical experiments we compare the performance of the CGSSML algorithms based on determinant or on trace or on minimizing the measure function of Byrd and Nocedal with different line searches. Figure 7 presents the performances of DESW versus DEAW, of DESW versus DEIW and of DEAW versus DEIW. We see that both the CGSSML algorithms based on determinant with the approximate Wolfe and with the improved Wolfe line search are more robust than the algorithm with standard Wolfe line search. The CGSSML algorithms based on determinant with approximate Wolfe and with the improved Wolfe line searches have similar performances, subject to the CPU time metric.

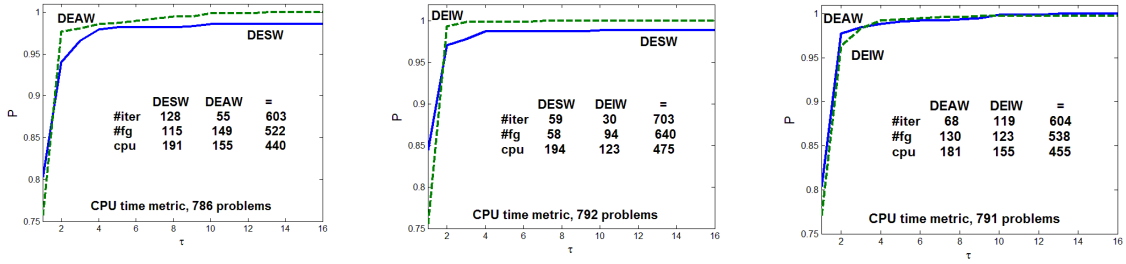

Fig. 7 Performance profiles of DESW versus DEAW, of DESW versus DEIW and of DEAW versus DEIW

Figure 8 shows the performances of TRSW versus TRAW, of TRSW versus TRIW and of TRAW versus TRIW. We see that both the CGSSML algorithms based on trace with the approximate Wolfe and with the improved Wolfe line search are more robust than the algorithm with standard Wolfe line search. Again the CGSSML algorithms based on trace with the approximate Wolfe and with the improved Wolfe line searches have similar performances, TRIW being slightly more efficient. In other words, the improved Wolfe line search is more efficient versus the approximate Wolfe line search, at least for this set of optimization problems.

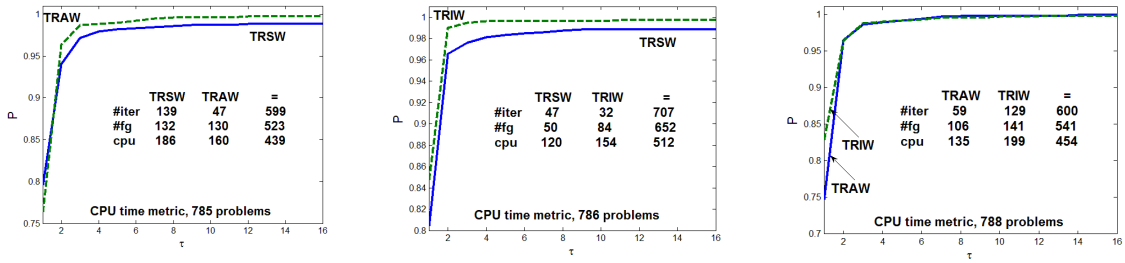

Fig. 8 Performance profiles of TRSW versus TRAW, of TRSW versus TRIW and of TRAW versus TRIW

Figure 9 shows the performances of FISW versus FIAW, of FISW versus FIIW and of FIAW versus FIIW. We see that the CGSSML algorithms based on minimizing the measure function of Byrd and Nocedal endowed with approximate Wolfe line search or with improved Wolfe line search are more robust than the algorithms with standard Wolfe line search. Besides, we have computational evidence that the algorithms minimizing the measure function with approximate Wolfe line search and with improved Wolfe line search have similar performances, FIAW being slightly more efficient.

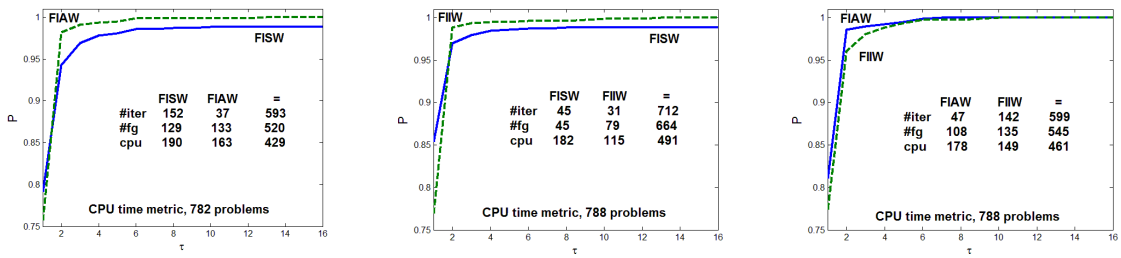

Fig. 9 Performance profiles of FISW versus FIAW, of FISW versus FIIW and of FIAW versus FIIW

Some remarks are in order.

1) Both the approximate Wolfe line search and the improved Wolfe line search are important ingredients for the efficiency and robustness of the self-scaling memoryless BFGS algorithms with clustering the eigenvalues. The performances of the CGSSML algorithms with approximate or improved line searches are better than the performances of the same algorithms with the standard Wolfe line search.

2) No matter how the stepsize is computed, using the standard, the approximate or the improved Wolfe line search, the performances of the CGSSML algorithms based on the determinant or on the trace of the iteration matrix H_{k+1}, or based on minimizing the measure function $\varphi(H_{k+1})$ defined by Byrd and Nocedal, are better that those of CG-DESCENT with Wolfe or with the approximate Wolfe line search.

3) For all procedures for stepsize computation using the standard, the approximate, or the improved Wolfe line search, the performances of the CGSSML algorithms based on the determinant or on the trace of the iteration matrix H_{k+1} are similar.

4) For the standard, the approximate or the improved Wolfe line searches, the self-scaling memoryless BFGS algorithms based on minimizing the measure function of Byrd and Nocedal is more efficient and more robust versus the algorithms based on the determinant or on the trace of H_{k+1}.

Both the CG-DESCENT and CGOPT are self-scaling memoryless BFGS algorithms obtained by using the singular values of some associated matrices. In our approach we followed the idea to cluster the eigenvalues of the iteration matrix, based on the determinant or on the trace of this matrix, or to minimize the measure function associated to all the eigenvalues of the iteration matrix. We conclude that the self-scaling memoryless BFGS algorithms based on minimizing the measure function of Byrd and Nocedal, which is a combination of the determinant and of the trace of the iteration matrix, are top performers versus the same algorithms based on the determinant or on the trace of the iteration matrix.

6 Conclusions

In this paper we have presented new procedures for selecting the parameter in the family of conjugate gradient algorithms of Dai and Kou [9], in which the search direction is closest to the Perry and Shanno self-scaling memoryless BFGS search direction. Three procedures have been developed. Two of them are based on clustering the eigenvalues of the self-scaling memoryless BFGS matrix H_{k+1} using the determinant or using the trace of H_{k+1}. The third one is based on minimizing the measure function of Byrd and Nocedal. The algorithm based on clustering the eigenvalues of H_{k+1} using the determinant is the same as CGOPT established by Dai and Kou [9]. The sufficient descent condition was established for all three members of the family. For stepsize computation, all these algorithms are endowed with the standard, the approximate and the improved Wolfe line search. In our numerical experiment we implemented the Powell restarting criterion. Our intensive numerical experiments showed that the algorithms using the clustering the eigenvalues of H_{k+1} based on determinant or on the trace of this matrix have similar performances. All the self-scaling memoryless BFGS algorithms endowed with the standard, or the approximate, or the improved Wolfe line search, are top performers versus the CG-DESCENT algorithm of Hager and Zhang. Besides, the self-scaling memoryless BFGS algorithm based on minimizing the measure function of Byrd and Nocedal is superior to the self-scaling memoryless BFGS algorithms based on the determinant or on the trace of the iteration matrix H_{k+1}, including here the CGOPT algorithm by Dai and Kou.

As an extension of the ideas presented in this paper, instead of minimizing the measure function $\varphi(H_{k+1})$ of Byrd and Nocedal, to determine a value for the parameter τ_k in (20) we may consider minimizing another measures function given by Dennis and Wolkowicz [42]

$$\omega(H_{k+1}) = \frac{tr(H_{k+1})}{n\left(\det(H_{k+1})\right)^{1/n}}. \tag{66}$$

Another possibility is to consider the self-scaling memoryless Broyden family of methods which include two parameters. We mention the possibility to implement in step 8 of CGSSML the adaptive restart criterion of Dai and Kou [9]. Finally, another extension is to implement CGSSML by modifying the most recent C version 6.8 of CG-DESCENT code of Hager and Zhang [37]

References

1. Broyden, C.G.: The convergence of a class of double-rank minimization algorithms. I. General considerations. Journal of the Institute of Mathematics and Its Applications, **6**, 76-90 (1970)
2. Fletcher, R.: A new approach to variable metric algorithms. The Computer Journal, **13**, 317-322 (1970)
3. Goldfarb, D.: A family of variable metric method derived by variation mean. Mathematics of Computation, **23**, 23-26 (1970)
4. Shanno, D.F.: Conditioning of quasi-Newton methods for function minimization. Mathematics of Computation, **24**, 647-656 (1970)
5. Perry, A.: A class of conjugate gradient algorithms with two step variable metric memory. Discussion paper 269, Center for Mathematical Studies in Economics and Management Science. Northwestern University, Il, USA. (1977)

6. Shanno, D.F.: Conjugate gradient methods with inexact searches, Mathematics of Operations Research **3**, 244-256 (1978)
7. Hager, W.W., & Zhang, H.: A new conjugate gradient method with guaranteed descent and an efficient line search. SIAM Journal on Optimization, **16**, 170-192 (2005)
8. Hager, W.W., & Zhang, H.: Algorithm 851: CG-DESCENT, a conjugate gradient method with guaranteed descent. ACM Transactions on Mathematical Software, **32**(1), 113-137 (2006)
9. Dai, Y.H., & Kou, C.X.: A nonlinear conjugate gradient algorithm with an optimal property and an improved Wolfe line search. SIAM Journal on Optimization, **23**(1), 296-320 (2013)
10. Oren, S.S., & Spedicato, E.: Optimal conditioning of self-scaling variable metric algorithm. Mathematical Programming, **10**, 70-90 (1976)
11. Oren, S.S.: Self-scaling variable metric (SSVM) algorithms. Part II: Implementation and experiments. Management Science, **20**(5), 863-874 (1974)
12. Oren, S.S., & Luenberger, D.G.: Self-scaling variable metric (SSVM) algorithms, part I: criteria and sufficient conditions for scaling a class of algorithms. Management Science, **20**, 845-862 (1974)
13. Al-Baali, M.: Numerical experience with a class of self-scaling quasi-Newton algorithms, Journal of Optimization Theory and Applications, **96**(3), 533-553 (1998)
14. Byrd, R., & Nocedal, J.: A tool for the analysis of quasi-Newton methods with application to unconstrainedminimization. SIAM J. Numer. Anal. **26**, 727–739 (1989)
15. Andrei, N.: *Critica Raţiunii Algoritmilor de Optimizare fără Restricţii.* [*Criticism of the Unconstrained Optimization Algorithms Reasoning*]. Editura Academiei Române, Bucureşti (2009)
16. Wolfe, P.: Convergence conditions for ascent methods. SIAM Review, **11**, 226-235 (1969)
17. Wolfe, P.: Convergence conditions for ascent methods. II: Some corrections. SIAM Review, **13**, 185-188(1971)
18. Nocedal, J., & Yuan, Y.X.: Analysis of self-scaling quasi-Newton method. Mathematical Programming, **61**, 19-37 (1993)
19. Gilbert, J.C., & Nocedal, J.: Global convergence properties of conjugate gradient methods for optimization, SIAM Journal on Optimization, **2**, 21-42 (1992)
20. Dai, Y.H., & Liao, L.Z.: New conjugate conditions and related nonlinear conjugate gradient methods, Applied Mathematics & Optimization, **43**, 87-101 (2001)
21. Hestenes, M.R., & Stiefel, E.: Methods of conjugate gradients for solving linear systems. Journal of Research of the National Bureau of Standards, **49**, 409-436 (1952)
22. Axelsson, O., & Lindskog, G.: On the rate of convergence of the preconditioned conjugate gradient methods, Numer. Math. **48**, 499–523 (1986)
23. Winther, R.: Some superlinear convergence results for the conjugate gradient method, SIAM J. Numer. Anal. **17**, 14–17 (1980)
24. Kratzer, D., Parter, S.V., & Steuerwalt, M.: Block splittings for the conjugate gradient method, Comput. Fluid. **11**, 255–279 (1983)
25. Kaporin, I.E.: New convergence results and preconditioning strategies for the conjugate gradient methods, Numer. Linear Algebr. **1**, 179–210 (1994)
26. Luenberger, D.G.: Introduction to linear and nonlinear programming. Addison-Wesley Publishing Company, Reading, Second edition, (1984)
27. Fletcher, R.: A new variational result for quasi-Newton formulae. SIAM J. Optim. **1**, 18–21 (1991)
28. Andrei, N.: Eigenvalues versus singular values study in conjugate gradient algorithms for large-scale unconstrained optimization. Optim. Methods. Softw. **32**(3), 534–551 (2017)
29. Andrei N.: A new three-term conjugate gradient algorithm for unconstrained optimization. Numer. Algorithms **68**, 305-321 (2015)
30. Andrei, N.: An adaptive conjugate gradient algorithm for large-scale unconstrained optimization. J. Comput. Appl. Math. **292**, 83–91 (2016)

31. Babaie-Kafaki, S.: On optimality of the parameters of self-scaling memoryless quasi-Newton updating formulae. J. Optim. Theory Appl. **167**(1), 91–101 (2015)
32. Babaie-Kafaki, S.: A modified scaling parameter for the memoryless BFGS updating formula. Numer Algorithms. **72**(2), 425–433 (2016)
33. Powell, M.J.D.: Nonconvex minimization calculations and the conjugate gradient method. In Griffiths, D.F. (Ed.) Numerical Analysis (Dundee, 1983), Lecture Notes in Mathematics, vol. 1066, 122-141 (1984)
34. Birgin, E., & Martínez, J.M.: A spectral conjugate gradient method for unconstrained optimization. Applied Mathematics & Optimization, **43**(2), 117-128 (2001)
35. Zoutendijk, G.: Nonlinear programming, computational methods. In J. Abadie (Ed.) Integer and Nonlinear Programming. North-Holland, Amsterdam, 38-86 (1970)
36. Dai, Y.H.: Convergence Analysis of Nonlinear Conjugate Gradient Methods. In Wang, Y., Yagola, A.G., & Yang, C., (Eds.) *Optimization and Regularization for Computational Inverse Problems and Applications*, Chapter 8, Higher Education Press, Beijing and Springer-Verlag, Berlin, Heidelberg, 157-181 (2010)
37. Hager, W.W., & Zhang, H.: The limited memory conjugate gradient method. SIAM Journal on Optimization **23**(4), 2150-2168 (2013)
38. Andrei, N.: UOP - A collection of 80 unconstrained optimization test problems. Technical Report No. 7/2018, November 17, Research Institute for Informatics, Bucharest, Romania. (2018)
39. Bongartz, I., Conn, A.R., Gould, N.I.M., & Toint, P.L.: CUTE: constrained and unconstrained testing environments. ACM TOMS **21**, 123-160 (1995)
40. Dolan, E.D., & Moré, J.J.: Benchmarking optimization software with performance profiles. Mathematical Programming, **91**, 201-213 (2002)
41. Hager, W.W.: A derivative-based bracketing scheme for univariate minimization and the conjugate gradient method. Computers Math. Applic. **18**(9), 779-795 (1988)
42. Dennis, J.E., & Wolkowicz, H.: Sizing and least-change secant methods, SIAM Journal on Numerical Analysis, **30**(5), 1291-1314 (1993)

April 18, 2019

-----oooOooo-----

APPENDIX (Fortran program CG3x8.FOR)

```
c-------------------------------------------------------------------------------
c                              January 17, 2019
c
c                                Main program.
c
c                        All subroutines are included.
c
c The program is a modification of the CG-DESCENT of Hager and Zhang (2005)
c to include different formulae for parameter beta computation under the
c three line search conditions: standard Wolfe, Approximate Wolfe and
c Improved Wolfe.
c
c-------------------------------------------------------------------------------
c The program is that of Hager and Zhang (CG-DESCENT), where
c the formula for beta computation is modified as:
c
c  ibeta     Algorithm
c -------------------------
c    1       Hager and Zhang (2005)
c    2       Minim DETERMINANT (Andrei, Thechnical Report No.2/2019)
c    3       Minim TRACE (Andrei, Thechnical Report No.2/2019)
c    4       Minim Fi - measure function of Byrd and Nocedal
c            (Andrei, Thechnical Report No.2/2019)
c    5       Hestenes - Stiefel
c    6       Dai - Yuan
c    7       Polak-Ribiere-Polyak
c    8       Minim of combination of DETERMINANT and TRACE (Andrei, Thechnical
c            Report No.2/2019)
c
c
c The algorithms are described in:
c
c   N. Andrei, A Conjugate Gradient Algorithms Closest to Self-Scaling
c   Memoryless BFGS Method based on minimizing the Byrd-Nocedal measure
c   function with Different Wolfe Line Searches for Unconstrained Optimization.
c   Technical Report No.2/2019,
c   April 18, 2019.
c   28 pages.
c
c  The  program  implements  optimization  of  80  unconstrained  optimization  test
problems
c described in:
c
c  N. Andrei, UOP - A collection of 80 unconstrained optimization test
C  problems.
c  Technical Report No. 7/2018, November 17,
c  Research Institute for Informatics, Bucharest, Romania. (2018)
c
c The functions are implemented in Fortran.
c The subroutine CG_VALUE contains the Fortran expressions of the minimizing
C functions.
c The subroutine CG_GRAD contains the Fortran expressions of the gradient of
C the minimizing functions.
c
c The subroutine INIPOINT contains the initial point for starting the
C optimization.
c
c The parameters are defined in the subroutine CG_DESCENT. (See subroutine
C CG_INIT.)
c
c
c
```

```
c  Dr. Neculai Andrei
c  Center for Advanced Modeling and optimization,
c  Academy of Romanian Scientists
c  E-mail: nandrei@ici.ro

c-----------------------------------------------------------------------------
      parameter (maxsize = 100000)

      real*8 x (maxsize), d (maxsize), g (maxsize),
     &       xtemp (maxsize), gtemp (maxsize), gnorm, f, timp

       real*8 tol

       logical AWolfe,IWolfe,SWolfe

      integer n, status, iter, nfunc, ngrad,ibeta
      integer*4 gh,gm,gs,gc, ght,gmt,gst,gct, timpexp
      integer*2 iyear, imonth, iday
      character*70 numef, fnumef(200)

      external cg_value, cg_grad

c Input file

      open(unit=7,file='fun80.txt',status='old')

c Output files

      open(unit=8,file='fisw.out',status='unknown')
      open(unit=9,file='fisw.rez',status='unknown')

c-----------------------------------------Select the line search procedure

       SWolfe = .true.
       AWolfe = .false.
       IWolfe = .false.

*-------------------------------------------------------------------
c
c---------------------------------------------- Select beta parameter

c ibeta = 1:  Hager and Zhang
c ibeta = 2:  Minim DETERMINANT  (same as Dai and Kou)
c ibeta = 3:  Minim TRACE
c ibeta = 4:  Minim measure function FI (Byrd-Nocedal)
c ibeta = 5:  Hestenes and Stiefel
c ibeta = 6:  Dai and Yuan
c ibeta = 7:  Polak, Ribiere and Polyak
c ibeta = 8:  Combination trace & log(det)

       ibeta = 8

c---------------------------------------------- Range of numerical experiments
       nexpini = 1
               nexptot = 80

c---------------------------------------------- Stop rule

c                              StopRule = F: |g|_infty <= tol
      tol = 0.000001d0

c-----------------------------------------------------------------------------
```

```
      write(8,90)
90    format(1x,52('*'))
      write(8,91)
91    format(1x,'* FCG - Fast Conjugate Gradient Project           *')
      write(8,92)
92    format(1x,'*                                                 *')
     call getdat(iyear, imonth, iday)
     write(8,1) imonth, iday, iyear
1    format(1x,'* Date: Month:',i2,'  Day:',i2,'   Year: ',i4,15x,'*')

c-----------------------------------------------------------------------------
-
            if(SWolfe) write(8,111)
111   format(1x,'* Standard Wolfe line search                      *')
            if(IWolfe) write(8,112)
112   format(1x,'* Improved Wolfe line search                      *')
            if(AWolfe) write(8,113)
113   format(1x,'* Approximate Wolfe line search                   *')
c-----------------------------------------------------------------------------
--

            if(ibeta .eq. 1) write(8,211)
211   format(1x,'* Hager - Zhang conjugate gradient algorithm      *')
            if(ibeta .eq. 2) write(8,212)
212   format(1x,'* MINIM DETERMINANT conjugate gradient algorithm  *')
            if(ibeta .eq. 3) write(8,213)
213   format(1x,'* MINIM TRACE conjugate gradient algorithm        *')
            if(ibeta .eq. 4) write(8,214)
214   format(1x,'* MINIM FI conjugate gradient algorithm           *')
            if(ibeta .eq. 5) write(8,215)
215   format(1x,'* Hestenes-Stiefel conjugate gradient algorithm   *')
            if(ibeta .eq. 6) write(8,216)
216   format(1x,'* Dai-Yuan conjugate gradient algorithm           *')
            if(ibeta .eq. 7) write(8,217)
217   format(1x,'* Polak-Ribiere-Polyak conjugate gradient algorithm*')
            if(ibeta .eq. 8) write(8,218)
218   format(1x,'* Min TRACE & DET conjugate gradient algorithm    *')

      write(8,90)

             do i=1,80
               read(7,21) numef
               fnumef(i)=numef
21             format(a70)
             end do

*---------------------------------------------- Start experiments

     do nexp = nexpini, nexptot

          itert = 0
          nfunct= 0
          ngradt= 0
          timp  = 0

          numef = fnumef(nexp)

      call printbeta(ibeta, SWolfe,AWolfe,IWolfe, numef,nexp)

     write(8,19)
19   format(9x,'n',5x,'iter',4x,'nfunc',4x,'ngrad',2x,'time(c)',
    *       15x,'f',18x,'gnorm',9x,'s')
     write(8,20)
```

```
20    format(1x,94('-'))

          do  n = 1000, 10000, 1000

c Call INIPOINT   (initial guess)

          call inipoint (n,x, nexp)

c Call CG_DESCENT for optimization

       call gettim(gh,gm,gs,gc)

      call cg_descent (tol, x, n, cg_value, cg_grad,
     &           status, gnorm,
     &           f,iter, nfunc, ngrad, d, g, xtemp, gtemp,
     &           SWolfe,IWolfe,AWolfe,ibeta,nexp)

       call gettim(ght,gmt,gst,gct)
       call exetim(gh,gm,gs,gc, ght,gmt,gst,gct)
*
       timpexp = ght*360000 + gmt*6000 + gst*100 + gct

          itert  = itert  + iter
          nfunct = nfunct + nfunc
          ngradt = ngradt + ngrad
          timp = timp + float(timpexp)

*------------------------------------------------------------------ *.out

      write(8,51) n, iter, nfunc, ngrad, timpexp, f, gnorm,status
51    format(3x,i7,3x,i6,3x,i6,3x,i6,3x,i6,5x,e20.13,2x,e20.13,1x,i1)

      write(*,851) n, iter, nfunc, ngrad, timpexp, f, gnorm
851   format(1x,i6,2x,i4,2x,i5,2x,i5,2x,i5,2x,e20.13,2x,e20.13)


*------------------------------------------------------------------ *.rez
          if(n .eq. 1000) then
            write(9,611)nexp, n,iter, nfunc, timpexp,f,gnorm
611         format(i2,i6,2x,i6,2x,i6,2x,i6,2x,e20.13,2x,e20.13)
          else
            write(9,61)       n,iter, nfunc, timpexp,f,gnorm
61          format(2x,i6,2x,i6,2x,i6,2x,i6,2x,e20.13,2x,e20.13)
          end if
*-----------------------------------------------------------------

          end do
c-------------------------------------- End do n

          write(8,20)
          write(8,71)  itert, nfunct, ngradt, timp/100.d0
          write(*,71)  itert, nfunct, ngradt, timp/100.d0
71        format(6x,'TOTAL',2x,i6,3x,i6,3x,i6,2x,f7.2,'(seconds)')

      end do
c---------------------------------------------End do nexp

      write(9,1)imonth, iday, iyear

      stop
      end

c
```

```
c ------------------------------------------ Last line  Main program



*-----------------------------------------------------------------
*  Date created       : May 30, 1995
*  Date last modified : May 30, 1995
*
*  Subroutine for execution time computation.
*
*-----------------------------------------------------------------
*
       subroutine exetim(tih,tim,tis,tic,tfh,tfm,tfs,tfc)
*
         integer*4 tih,tim,tis,tic
         integer*4 tfh,tfm,tfs,tfc
*
         integer*4 ti,tf
         integer*4 ch,cm,cs
         data ch,cm,cs/360000,6000,100/
*
         ti=tih*ch+tim*cm+tis*cs+tic
         tf=tfh*ch+tfm*cm+tfs*cs+tfc
         tf=tf-ti
         tfh=tf/ch
         tf=tf-tfh*ch
         tfm=tf/cm
         tf=tf-tfm*cm
         tfs=tf/cs
         tfc=tf-tfs*cs
*
         return
       end
*----------------------------------------------------- Last line exetim




       subroutine printbeta(ibeta, SWolfe,AWolfe,IWolfe,numef,nexp)
       integer ibeta,nexp
              character*70 numef
       logical SWolfe, AWolfe, IWolfe

c-------------------------------------------------------------------ibeta=1
              if(ibeta .eq. 1) then
       if(SWolfe) then
         write(8,11) nexp, numef
         write(*,11) nexp, numef
11       format(/,2x,i3,2x,'Hager-Zhang. Standard WLS. Function:',a40,/)
       end if
       if(IWolfe) then
         write(8,12) nexp, numef
         write(*,12) nexp, numef
12       format(/,2x,i3,2x,'Hager-Zhang. Improved WLS. Function:',a40,/)
       end if
       if(AWolfe) then
         write(8,13) nexp, numef
         write(*,13) nexp, numef
13    format(/,2x,i3,2x,'Hager-Zhang. Approximate WLS. Function:',a40,/)
       end if
          end if
```

```
c-------------------------------------------------------------------ibeta=2
      if(ibeta .eq. 2) then
      if(SWolfe) then
        write(8,21) nexp, numef
        write(*,21) nexp, numef
21      format(/,2x,i3,2x,'min DET. Standard WLS. Function:',a40,/)
      end if
      if(IWolfe) then
        write(8,22) nexp, numef
        write(*,22) nexp, numef
22      format(/,2x,i3,2x,'min DET. Improved WLS. Function:',a40,/)
      end if
      if(AWolfe) then
        write(8,23) nexp, numef
        write(*,23) nexp, numef
23    format(/,2x,i3,2x,'min DET. Approximate WLS. Function:',a40,/)
      end if
         end if


c-------------------------------------------------------------------ibeta=3
      if(ibeta .eq. 3) then
      if(SWolfe) then
        write(8,31) nexp, numef
        write(*,31) nexp, numef
31      format(/,2x,i3,2x,'min TRACE. Standard WLS.',
     *                    ' Function:',a40,/)
      end if
      if(IWolfe) then
        write(8,32) nexp, numef
        write(*,32) nexp, numef
32      format(/,2x,i3,2x,'min TRACE. Improved WLS.',
     *                    ' Function:',a40,/)
      end if
      if(AWolfe) then
        write(8,33) nexp, numef
        write(*,33) nexp, numef
33    format(/,2x,i3,2x,'min TRACE. Approximate WLS.',
     *                  ' Function:',a40,/)
      end if
         end if


c-------------------------------------------------------------------ibeta=4
      if(ibeta .eq. 4) then
      if(SWolfe) then
        write(8,41) nexp, numef
        write(*,41) nexp, numef
41      format(/,2x,i3,2x,'Min FI (Byrd-Nocedal). Standard WLS.',
     *                    ' Function:',a40,/)
      end if
      if(IWolfe) then
        write(8,42) nexp, numef
        write(*,42) nexp, numef
42      format(/,2x,i3,2x,'Min FI (Byrd-Nocedal). Improved WLS.',
     *                    ' Function:',a40,/)
      end if
      if(AWolfe) then
        write(8,43) nexp, numef
        write(*,43) nexp, numef
43    format(/,2x,i3,2x,'Min FI (Byrd-Nocedal). Approximate WLS.',
```

```
     *                  ' Function:',a40,/)
      end if
         end if


c-------------------------------------------------------------------ibeta=5
      if(ibeta .eq. 5) then
      if(SWolfe) then
        write(8,51) nexp, numef
        write(*,51) nexp, numef
51      format(/,2x,i3,2x,'Hestenes-Stiefel. Standard WLS.',
     *                     ' Function:',a40,/)
      end if
      if(IWolfe) then
        write(8,52) nexp, numef
        write(*,52) nexp, numef
52      format(/,2x,i3,2x,'Hestenes-Stiefel. Improved WLS.',
     *                     ' Function:',a40,/)
      end if
      if(AWolfe) then
        write(8,53) nexp, numef
        write(*,53) nexp, numef
53    format(/,2x,i3,2x,'Hestenes-Stiefel. Approximate WLS.',
     *                  ' Function:',a40,/)
      end if
         end if


c-------------------------------------------------------------------ibeta=6
      if(ibeta .eq. 6) then
      if(SWolfe) then
        write(8,61) nexp, numef
        write(*,61) nexp, numef
61      format(/,2x,i3,2x,'Dai-Yuan. Standard WLS.',
     *                     ' Function:',a40,/)
      end if
      if(IWolfe) then
        write(8,62) nexp, numef
        write(*,62) nexp, numef
62      format(/,2x,i3,2x,'Dai-Yuan. Improved WLS.',
     *                     ' Function:',a40,/)
      end if
      if(AWolfe) then
        write(8,63) nexp, numef
        write(*,63) nexp, numef
63    format(/,2x,i3,2x,'Dai-Yuan. Approximate WLS.',
     *                  ' Function:',a40,/)
      end if
         end if

c-------------------------------------------------------------------ibeta=7
      if(ibeta .eq. 7) then
      if(SWolfe) then
        write(8,71) nexp, numef
        write(*,71) nexp, numef
71      format(/,2x,i3,2x,'Polak-Ribiere-Polyak. Standard WLS.',
     *                     ' Function:',a40,/)
      end if
      if(IWolfe) then
        write(8,72) nexp, numef
        write(*,72) nexp, numef
72      format(/,2x,i3,2x,'Polak-Ribiere-Polyak. Improved WLS.',
     *                     ' Function:',a40,/)
```

```
      end if
      if(AWolfe) then
        write(8,73) nexp, numef
        write(*,73) nexp, numef
73    format(/,2x,i3,2x,'Polak-Ribiere-Polyak. Approximate WLS.',
     *                  ' Function:',a40,/)
      end if
         end if


c-------------------------------------------------------------------ibeta=8
      if(ibeta .eq. 8) then
      if(SWolfe) then
        write(8,81) nexp, numef
        write(*,81) nexp, numef
81      format(/,2x,i3,2x,'TRACE & DETERMINANT (det<1). Standard WLS.',
     *                    ' Function:',a40,/)
      end if
      if(IWolfe) then
        write(8,82) nexp, numef
        write(*,82) nexp, numef
82      format(/,2x,i3,2x,'TRACE & DETERMINANT (det<1). Improved WLS.',
     *                    ' Function:',a40,/)
      end if
      if(AWolfe) then
        write(8,83) nexp, numef
        write(*,83) nexp, numef
83    format(/,2x,i3,2x,'TRACE & DETERMINANT (det<1). Approximate WLS.',
     *                  ' Function:',a40,/)
      end if
         end if

      return
      end
c ---------------------------------------------------- Last line Printbeta




c     | -------------------------------------------------------------------
c     |      A conjugate gradient method with guaranteed descent
c     |                                 January 15, 2004
c     |           William W. Hager    and   Hongchao Zhang
c     |          hager@math.ufl.edu       hzhang@math.ufl.edu
c     |                   Department of Mathematics
c     |                     University of Florida
c     |                 Gainesville, Florida 32611 USA
c     |                      352-392-0281 x 244
c     |
c     |                 copyright by William W. Hager
c     |
c     |   http://www.math.ufl.edu/~hager/papers/cg_descent.ps
c     |   http://www.math.ufl.edu/~hager/papers/cg_compare.ps
c     |   http://www.math.ufl.edu/~hager/papers/cg_manual.ps
c     |
c     |    INPUT:
c     |
c     |(double) grad_tol-- StopRule = F: |g|_infty <= grad_tol[default]
c     |                    StopRule = T: |g|_infty <= grad_tol(1+|f|)
c     |
c     |(double) x       --starting guess (length n)
c     |
```

```
c     |(int)    dim     --problem dimension (also denoted n)
c     |
c     |         cg_value--name of cost evaluation subroutine
c     |                  (external in main program, cg_value(f, x, n)
c     |                   puts value of cost function at x in f
c     |                   f is double precision scalar and x is
c     |                   double precision array of length n)
c     |
c     |         cg_grad --name gradient evaluation subroutine
c     |                  (external in main program, cg_grad (g, x, n)
c     |                   puts gradient at x in g, g and x are
c     |                   double precision arrays of length n)
c     |
c     |(double) gnorm   --if the parameter Step in cg.parm is .true.,
c     |                   then gnorm contains the initial step used at
c     |                   iteration 0 in the line search
c     |
c     |(double) d       --direction (work array, length n)
c     |
c     |(double) g       --gradient (work array, length n)
c     |
c     |(double) xtemp   --work array (work array, length n)
c     |
c     |(double) gtemp   --work array (work array, length n)
c     |
c     |    OUTPUT:
c     |
c     |(int)    status  -- 0 (convergence tolerance satisfied)
c     |                    1 (change in func <= feps*|f|)
c     |                    2 (total iterations exceeded maxit)
c     |                    3 (slope always negative in line search)
c     |                    4 (number secant iterations exceed nsecant)
c     |                    5 (search direction not a descent direction)
c     |                    6 (line search fails in initial interval)
c     |                    7 (line search fails during bisection)
c     |                    8 (line search fails during interval update)
c     |
c     |(double) gnorm   --max abs component of gradient
c     |
c     |(double) f       --function value at solution
c     |
c     |(double) x       --solution (length n)
c     |
c     |(int)    iter    --number of iterations
c     |
c     |(int)    nfunc   --number of function evaluations
c     |
c     |(int)    ngrad   --number of gradient evaluations
c     |
c     |Note: The file cg.parm must be placed in the directory where
c     |      the code is run
c     | ---------------------------------------------------------------------
c


      subroutine cg_descent (grad_tol, x, dim, cg_value, cg_grad,
     &          status, gnorm, f, iter, nfunc, ngrad,
     &          d, g, xtemp, gtemp, SWolfe,IWolfe,AWolfe,ibeta,nexp)

      double precision x (1), d (1), g (1), xtemp (1), gtemp (1),
     &                delta, sigma, epsilon, theta, gamma, rho, tol,
     &                eta, fpert, f0, wolfe_hi, wolfe_lo,
     &                awolfe_hi, QuadCutOff, zero, feps,
```

```
     &                 psi0, psi1, psi2,
     &                 grad_tol, delta2, eta_sq,   gtgprev,z1,z2,
     &                 f, ftemp, gnorm, xnorm, gnorm2, dnorm2, denom,
     &                 t, t1, t2, t3, t4, dphi, dphi0, alpha, talpha,
     &                 yk, yk2, ykgk, dkyk, beta,dkdk,etta, gkgk,tauu,
     &                 a8, y(100000),trh,deth,bn

         integer       nrestart, nexpand, nsecant, maxit,
     &                 n, n5, n6, nf, ng, info,
     &                 iter, status, nfunc, ngrad,ibeta,
     &                 i, j, i1, i2, i3, i4, dim


         logical QuadOK, QuadStep, PrintLevel, PrintFinal,
     &           StopRule, ERule, AWolfe, Step, cg_tol,IWolfe,SWolfe

         external      cg_value, cg_grad

      common /cgparms/delta, sigma, epsilon, theta, gamma, rho, tol,
     &                eta, fpert, f0, wolfe_hi, wolfe_lo,
     &                awolfe_hi, QuadCutOff, zero, feps,
     &                psi0, psi1, psi2,
     &                n, n5, n6, nf, ng, info,
     &                nrestart, nexpand, nsecant, maxit,
     &                QuadStep, PrintLevel, PrintFinal, StopRule,
     &                ERule, Step, QuadOK

c initialize the parameters

      call cg_init (grad_tol, dim)

         delta2 = 2*delta - 1
         eta_sq = eta*eta
         iter = 0

          z1 = 1.d0/(float(n)-1.d0)                        ! For FI only
          z2 = (float(n)-2.d0)/(float(n)-1.d0)             !


c initial function and gradient evaluations, initial direction

      call cg_value (f, x, n, nexp)
         nf = nf + 1
      call cg_grad (g, x, n, nexp)            !g = gradient(x0)  d(0)=-g(0)
         ng = ng + 1
         f0 = f + f
         gnorm = zero
         xnorm = zero
         gnorm2 = zero
         do i = 1, n5
             xnorm = dmax1 (xnorm, dabs (x (i)))
             t = g (i)
             d (i) = -t
             gnorm = dmax1 (gnorm, dabs(t))
             gnorm2 = gnorm2 + t*t
         enddo
         do i = n6, n, 5
             xnorm = dmax1 (xnorm, dabs (x (i)))
             t = g (i)
             gnorm = dmax1 (gnorm, dabs (t))
             d (i)   = -t
             j = i + 1
             t1 = g (j)
```

```
            d (j) = -t1
            gnorm = dmax1 (gnorm, dabs (t1))
            xnorm = dmax1 (xnorm, dabs (x (j)))
            j = i + 2
            t2 = g (j)
            d (j) = -t2
            gnorm = dmax1 (gnorm, dabs (t2))
            xnorm = dmax1 (xnorm, dabs (x (j)))
            j = i + 3
            t3 = g (j)
            d (j) = -t3
            gnorm = dmax1 (gnorm, dabs (t3))
            xnorm = dmax1 (xnorm, dabs (x (j)))
            j = i + 4
            t4 = g (j)
            d (j) = -t4
            gnorm = dmax1 (gnorm, dabs (t4))
            xnorm = dmax1 (xnorm, dabs (x (j)))
            gnorm2 = gnorm2 + t*t + t1*t1 + t2*t2 + t3*t3 + t4*t4
        enddo

c  gnorm2 = g(k)T*g(k)  (=gtg)

        if ( PrintLevel ) then
            write (*, 10) iter, f, gnorm
   10       format ('iter: ', i5, ' f= ', e14.6, ' gnorm= ', e14.6)
        endif

        if ( cg_tol (f, gnorm) ) goto 100

        dphi0 = -gnorm2         ! dphi0 = d(0)T*g(0) = g(0)T*g(0)
        if ( Step ) then
            alpha = gnorm
        else
            alpha = psi0*xnorm/gnorm
            if ( xnorm .eq. zero ) then
                if ( f .ne. zero ) then
                    alpha = psi0*dabs (f)/gnorm2
                else
                    alpha = 1.d0
                endif
            endif
        endif

c start the conjugate gradient iteration

c
c   alpha starts as old step, ends as initial step for next iteration
c   f is function value for alpha = 0
c   QuadOK = .true. means that a quadratic step was taken
c
c=====================================================================
c
        do iter = 1, maxit

            etta = 1.d0/float(iter)/float(iter)
c            etta is used in improved Wofe I

            QuadOK = .false.
            alpha = psi2*alpha

            if ( QuadStep ) then
                if ( f .ne. zero ) then
```

```
                    t = dabs ((f-f0)/f)
                else
                    t = 1.d0
                endif
                if ( t .gt. QuadCutOff ) then
                    talpha = psi1*alpha
                 call cg_step (xtemp, x, d, talpha)
                 call cg_value (ftemp, xtemp, n, nexp)
                    nf = nf + 1
                    if ( ftemp .lt. f ) then
                       denom = 2.0d0*(((ftemp-f)/talpha)-dphi0)
                       if ( denom .gt. zero ) then
                           QuadOK = .true.
                           alpha = -dphi0*talpha/denom
                       endif
                    endif
                endif
            endif
            f0 = f

            if ( PrintLevel ) then
                write (*, 20) QuadOK, alpha, f0, dphi0
   20           format ('QuadOK:', l2, ' initial a:',
     &                e14.6,' f0:', e14.6, ' dphi', e14.6)
            endif

c Parameters in Wolfe and approximiate Wolfe conditions, and in update

            if ( ERule ) then
                fpert = f + epsilon
            else
                fpert = f + epsilon*dabs (f)
            endif

            wolfe_hi = delta*dphi0
            wolfe_lo = sigma*dphi0
            awolfe_hi = delta2*dphi0

         call cg_line (alpha, f, dphi, dphi0, x, xtemp, d, gtemp,
     &          cg_value, cg_grad,SWolfe,IWolfe,AWolfe,etta,nexp)

      do i=1,n
        y(i) = gtemp(i) - g(i)
      end do

      gnorm8 = cg_dot(g,g)

c
c ---------------------------------------------------------------------------

            if ( info .gt. 0 ) goto 100

c
c Test for convergence to within machine epsilon
c (set feps to zero to remove this test)
c
            if ( -alpha*dphi0 .le. feps*dabs (f) ) then
                info = 1
                goto 100
            endif

c compute beta, yk2, gnorm, gnorm2, dnorm2, update x and g,
```

```
          gtgprev = cg_dot(g,g)     ! For PR only


c-----------------------------------------------------------------------
            if ( mod (iter, nrestart) .ne. 0 ) then   ! Powell restart test

                gnorm = zero
                dnorm2 = zero
                yk2 = zero
                ykgk = zero
                do i = 1, n5
                    x (i) = xtemp (i)
                    t = gtemp (i)
                    yk = t - g (i)
                    yk2 = yk2 + yk**2
                    ykgk = ykgk + yk*t
                    g (i) = t
                    gnorm = dmax1 (gnorm, dabs (t))
                    dnorm2 = dnorm2 + d (i)**2
                enddo
                do i = n6, n, 5
                    x (i) = xtemp (i)
                    t = gtemp (i)
                    yk = t - g (i)
                    yk2 = yk2 + yk**2
                    ykgk = ykgk + yk*t
                    i1 = i + 1
                    x (i1) = xtemp (i1)
                    t1 = gtemp (i1)
                    i2 = i + 2
                    x (i2) = xtemp (i2)
                    t2 = gtemp (i2)
                    i3 = i + 3
                    x (i3) = xtemp (i3)
                    t3 = gtemp (i3)
                    i4 = i + 4
                    x (i4) = xtemp (i4)
                    t4 = gtemp (i4)
                    yk2 = yk2 + (t1-g (i1))**2 + (t2-g (i2))**2
     &                     + (t3-g (i3))**2 + (t4-g (i4))**2
                    ykgk = ykgk + (t1-g (i1))*t1 + (t2-g (i2))*t2
     &                       + (t3-g (i3))*t3 + (t4-g (i4))*t4
                    g (i) = t
                    gnorm = dmax1 (gnorm, dabs (t))
                    g (i1) = t1
                    gnorm = dmax1 (gnorm, dabs (t1))
                    g (i2) = t2
                    gnorm = dmax1 (gnorm, dabs (t2))
                    g (i3) = t3
                    gnorm = dmax1 (gnorm, dabs (t3))
                    g (i4) = t4
                    gnorm = dmax1 (gnorm, dabs (t4))
                    dnorm2 = dnorm2 + d (i)**2 + d (i1)**2 + d (i2)**2
     &                                  + d (i3)**2 + d (i4)**2
                enddo

                if ( cg_tol (f, gnorm) ) goto 100
                dkyk = dphi - dphi0
```

```
*                                    *
*                                   ***
*                                  *****
*                                   ***
*                                    *


c                              BETA computation:
c                             ====================
c
c
c  HAGER - ZHANG
       if (ibeta .eq. 1) then
                  beta = (ykgk - 2.d0*dphi*yk2/dkyk)/dkyk
                  beta = dmax1 (beta,
     &                -1.d0/dsqrt (dmin1 (eta_sq, gnorm2)*dnorm2))
       end if
c-----------------------------------------------------------------------


c Minim DETERMINANT  (same as CGOPT by Dai and Kou)
       if (ibeta .eq. 2) then
                   dkdk = cg_dot(d,d)
            tauu = dkyk/dkdk
                   beta = ykgk/dkyk -(tauu+yk2/dkyk-dkyk/dkdk)*dphi/dkyk
                   beta = dmax1(beta , 0.5d0*dphi/dkdk)
       end if
c-----------------------------------------------------------------------



c  Minim TRACE
       if (ibeta .eq. 3) then
                   dkdk = cg_dot(d,d)
                   a8 = (yk2*dkdk)/(dkyk*dkyk)
                   tauu = (2.d0 - a8)*dkyk/dkdk                        ! minTR

                   beta = ykgk/dkyk - tauu*dphi/dkyk -
     &                       yk2*dphi/(dkyk*dkyk) + dphi/dkdk
                   beta = dmax1( beta , 0.5d0*dphi/dkdk )
       end if
c-----------------------------------------------------------------------



c Minim measure function FI (Byrd-Nocedal)
       if (ibeta .eq. 4) then
                   dkdk = cg_dot(d,d)
                   a8 = (yk2*dkdk)/(dkyk*dkyk)
                   tauu = z2 + z1*a8
c                                                   min fi (Byrd-Nocedal)
                   beta = ykgk/dkyk - tauu*dphi/dkyk -
     &                      yk2*dphi/(dkyk*dkyk) + dphi/dkdk
                   beta = dmax1( beta , 0.5d0*dphi/dkdk )
       end if
c-----------------------------------------------------------------------


c HESTENES-STIEFEL
       if (ibeta .eq. 5) then
                   beta = ykgk/dkyk
                   beta = dmax1(0.d0 , beta)
       end if
```

```
c-----------------------------------------------------------------------


c  DAI-YUAN
       if(ibeta .eq. 6) then
                 gkgk = cg_dot(g,g)
                 beta = gkgk/dkyk
         beta = dmax1(beta , 0.d0)
       end if
c-----------------------------------------------------------------------


c  POLAK-RIBIERE-POLYAK
       if(ibeta .eq. 7) then
                  beta = ykgk/gtgprev
                  beta = dmax1(beta , 0.d0)
       end if
c-----------------------------------------------------------------------


c  Minim TRACE & LOG(DET)
       if (ibeta .eq. 8) then
                  dkdk = cg_dot(d,d)
           a8 = (yk2*dkdk)/(dkyk*dkyk)
            tauu = z2 + z1*a8

c Compute log(Det)

          deth = dkdk/( dkyk*(tauu**(n-1)) )
          if(deth .le. 1.d0) then
             tauu = dkyk/dkdk                          ! min DET
c           tauu = (2.d0 - a8)*dkyk/dkdk               ! min TRACE

            beta = ykgk/dkyk - tauu*dphi/dkyk -
     &              yk2*dphi/(dkyk*dkyk) + dphi/dkdk
            beta = dmax1( beta , 0.5d0*dphi/dkdk )
          else
            tauu = z2 + z1*a8                           ! min fi (Byrd-Nocedal)
            beta = ykgk/dkyk - tauu*dphi/dkyk -
     &              yk2*dphi/(dkyk*dkyk) + dphi/dkdk
            beta = dmax1( beta , 0.5d0*dphi/dkdk )
          end if

          if(beta .lt. 0.d0) then
             beta = 0.d0
                  end if

       end if
c-------------------------------------------------------------------------------
c                              End BETA computation
c                              ====================



c     update search direction: (Normal direction)

                 gnorm2 = zero

                 do i = 1, n5
                     t = g (i)
                     d (i) = -t + beta*d (i)
                     gnorm2 = gnorm2 + t*t
                 enddo
```

```
                do i = n6, n, 5
                    d (i)  = -g (i)  + beta*d (i)
                    i1 = i + 1
                    d (i1) = -g (i1) + beta*d (i1)
                    i2 = i + 2
                    d (i2) = -g (i2) + beta*d (i2)
                    i3 = i + 3
                    d (i3) = -g (i3) + beta*d (i3)
                    i4 = i + 4
                    d (i4) = -g (i4) + beta*d (i4)

                    gnorm2 = gnorm2 + g (i)**2 + g (i1)**2 + g (i2)**2
     &                              + g (i3)**2 + g (i4)**2
                enddo

                dphi0 = -gnorm2 + beta*dphi

            else

c     search direction d = -g (Restart)

                if ( PrintLevel ) then
                    write (*, *) "RESTART CG"
                endif
                gnorm = zero
                gnorm2 = zero
                do i = 1, n5
                    x (i) = xtemp (i)
                    t = gtemp (i)
                    g (i) = t
                    d (i) = -t
                    gnorm = dmax1 (gnorm, dabs(t))
                    gnorm2 = gnorm2 + t*t
                enddo
                do i = n6, n, 5
                    x (i) = xtemp (i)
                    t = gtemp (i)
                    g (i) = t
                    d (i) = -t
                    gnorm = dmax1 (gnorm, dabs(t))
                    j = i + 1
                    x (j) = xtemp (j)
                    t1 = gtemp (j)
                    g (j) = t1
                    d (j) = -t1
                    gnorm = dmax1 (gnorm, dabs(t1))
                    j = i + 2
                    x (j) = xtemp (j)
                    t2 = gtemp (j)
                    g (j) = t2
                    d (j) = -t2
                    gnorm = dmax1 (gnorm, dabs(t2))
                    j = i + 3
                    x (j) = xtemp (j)
                    t3 = gtemp (j)
                    g (j) = t3
                    d (j) = -t3
                    gnorm = dmax1 (gnorm, dabs(t3))
                    j = i + 4
                    x (j) = xtemp (j)
                    t4 = gtemp (j)
                    g (j) = t4
                    d (j) = -t4
```

```
                    gnorm = dmax1 (gnorm, dabs(t4))
                  gnorm2 = gnorm2 + t*t + t1*t1 + t2*t2 + t3*t3 + t4*t4
                enddo
                if ( cg_tol (f, gnorm) ) goto 100
                dphi0 = -gnorm2
            endif
            if ( PrintLevel ) then
                write (*, 10) iter, f, gnorm
            endif
            if ( dphi0 .gt. zero ) then
               info = 5
               goto 100
            endif
        enddo
c***

        info = 2
100     nfunc = nf
        ngrad = ng
        status = info
        if ( info .gt. 2 ) then
            gnorm = zero
            do i = 1, n
                x (i) = xtemp (i)
                g (i) = gtemp (i)
                gnorm = dmax1 (gnorm, dabs(g (i)))
            enddo
        endif
        if ( PrintFinal ) then
            write (6, *)
     &      'nexp=',nexp,'  n=', n, '   Termination status:', status
            if ( status .eq. 0 ) then
                write (6, 200)
            else if ( status .eq. 1 ) then
                write (6, 210)
            else if ( status .eq. 2 ) then
                write (6, 220) maxit
                write (6, 300)
                write (6, 400) grad_tol
            else if ( status .eq. 3 ) then
                write (6, 230)
                write (6, 300)
                write (6, 430)
                write (6, 410)
            else if ( status .eq. 4 ) then
                write (6, 240)
                write (6, 400) grad_tol
            else if ( status .eq. 5 ) then
                write (6, 250)
            else if ( status .eq. 6 ) then
                write (6, 260)
                write (6, 300)
                write (6, 400) grad_tol
                write (6, 410)
                write (6, 420)
            else if ( status .eq. 7 ) then
                write (6, 260)
                write (6, 400) grad_tol
            else if ( status .eq. 8 ) then
                write (6, 260)
                write (6, 300)
                write (6, 400) grad_tol
                write (6, 410)
```

```
                write (6, 420)
            endif
            write (6, 500) gnorm
            write (6, *) 'function value:', f
            write (6, *) 'cg iterations:', iter
            write (6, *) 'function evaluations:', nfunc
            write (6, *) 'gradient evaluations:', ngrad
         endif
         return
200   format (' Convergence tolerance for gradient satisfied')
210   format (' Terminating since change in function value <= feps*|f|')
220   format (' Total number of iterations exceed max allow:', i10)
230   format (' Slope always negative in line search')
240   format (' Line search fails, too many secant steps')
250   format (' Search direction not a descent direction')
260   format (' Line search fails')
300   format (' Possible causes of this error message:')
400   format ('   - your tolerance (grad_tol = ', d10.4,
     &        ') is too strict')
410   format ('   - your gradient routine has an error')
420   format ('   - the parameter epsilon in cg.parm is too small')
430   format ('   - your cost function has an error')
500   format (' absolute largest component of gradient: ', d10.4)
         end

c     PARAMETERS:
c
c     delta - range (0, .5), used in the Wolfe conditions
c     sigma - range [delta, 1), used in the Wolfe conditions
c     epsilon- range [0, infty), determines when to test approximate Wolfe
c     theta - range (0,1), used in interval update rules
c     gamma - range (0,1), determines when to perform bisection step
c     rho   - range (1, infty), growth factor when finding initial interval
c     eta   - range (0, infty), used in lower bound for beta
c     psi0  - range (0, 1), factor used in very initial starting guess
c     psi1  - range (0, 1), factor previous step multiplied by in QuadStep
c       psi2   - range (1, infty), factor previous step is multipled by for
startup
c     QuadCutOff - lower bound on rel change in f before QuadStep
c     restart_fac - range (0, infty) restart cg when iter = n*restart
c     maxit_fac - range (0, infty) terminate in maxit = maxit_fac*n iterations
c     feps - stop when -alpha*dphi0 (est. change in value) <= feps*|f|
c            (feps = 0 removes this test, example: feps = eps*1.e-5
c             where eps is machine epsilon)
c     tol   - range (0, infty), convergence tolerance
c     nexpand - range [0, infty), number of grow/shrink allowed in bracket
c     nsecant - range [0, infty), maximum number of secant steps
c     QuadStep- .true. (use quadratic step) .false. (no quadratic step)
c     PrintLevel- .false. (no printout) .true. (print intermediate results)
c     PrintFinal- .false. (no printout) .true. (print messages, final error)
c     StopRule - .false. (max abs grad <= tol) .true. (... <= tol*(1+|f|))
c     ERule - .false. (eps_k = epsilon|f|) .true. (eps_k = epsilon)
c     AWolfe - .false. (use standard Wolfe only)
c            - .true. (use approximate + standard Wolfe)
c     Step - .false. (program computing starting step at iteration 0)
c                - .true. (user provides starting step in gnorm argument of
cg_descent
c     info  - same as status
c
c     DEFAULT PARAMETER VALUES:
c
c         delta : 0.1
c         sigma : 0.9
```

```
c         epsilon: 1.e-6
c         theta : 0.5
c         gamma : 0.66
c         rho   : 5.0
c         restart: 1.0
c         eta   : 0.01
c         psi0  : 0.01
c         psi1  : 0.1
c         psi2  : 2.0
c         QuadCutOff: 1.d-12
c         tol   : grad_tol
c         maxit : 500*n
c         nrestart: n
c         nexpand: 50
c         nsecant: 50
c         QuadStep: .true.
c         PrintLevel: .false.
c         PrintFinal: .true.
c         StopRule: .false.
c         Step: .false.
c         info  : 0
c         feps  : 0.0
c

c                            CG_INIT
c-----------------------------------------------------------------------

         subroutine cg_init (grad_tol, dim)

         double precision delta, sigma, epsilon, theta, gamma, rho, tol,
     &                eta, fpert, f0, wolfe_hi, wolfe_lo,
     &                awolfe_hi, QuadCutOff, zero, feps,
     &                psi0, psi1, psi2,
     &                grad_tol, restart_fac, maxit_fac

         integer      nrestart, nexpand, nsecant, maxit,
     &                n, n5, n6, nf, ng, info,
     &                dim

         logical      QuadOK, QuadStep, PrintLevel, PrintFinal,
     &                StopRule, ERule, AWolfe, IWolfe,SWolfe,Step

      common /cgparms/delta, sigma, epsilon, theta, gamma, rho, tol,
     &               eta, fpert, f0, wolfe_hi, wolfe_lo,
     &               awolfe_hi, QuadCutOff, zero, feps,
     &               psi0, psi1, psi2,
     &               n, n5, n6, nf, ng, info,
     &               nrestart, nexpand, nsecant, maxit,
     &               QuadStep, PrintLevel, PrintFinal, StopRule,
     &               ERule, Step, QuadOK

         n = dim
         tol = grad_tol
c
         delta = 0.1
         sigma = 0.9
         epsilon= 1.e-6
         theta = 0.5
         gamma = 0.66
         rho   = 5.0
         restart= 1.0
         restart_fac=1.d0
         eta   = 0.01
```

```
         psi0  = 0.01
         psi1  = 0.1
         psi2  = 2.0
         QuadCutOff= 1.d-12
         tol   = grad_tol
c         maxit = 500*n
         maxit = 2000
         maxit_fac = 15
         nexpand= 50
         nsecant= 50
         QuadStep= .true.
         PrintLevel= .false.
         PrintFinal= .false.
         StopRule= .false.
         ERule= .false.
         Step= .false.
         feps  = 0.0
c
c--------
c
         nrestart = n*restart_fac
c         maxit = n*maxit_fac
         maxit=2000
         zero = 0.d0
         info = 0
         n5 = mod (n, 5)
         n6 = n5 + 1
         nf = 0
         ng = 0
         return
         end


c                                CG_WOLFE
c-----------------------------------------------------------------------


c check whether the Wolfe or the approximate Wolfe conditions
c are satisfied

      logical function cg_Wolfe(alpha,f,dphi,SWolfe,IWolfe,AWolfe,etta)

         double precision delta, sigma, epsilon, theta, gamma, rho, tol,
     &                eta, fpert, f0, wolfe_hi, wolfe_lo,
     &                awolfe_hi, QuadCutOff, zero, feps,
     &                psi0, psi1, psi2,
     &                alpha, f, dphi,etta

         integer          nrestart, nexpand, nsecant, maxit,
     &                n, n5, n6, nf, ng, info

         logical          QuadOK, QuadStep, PrintLevel, PrintFinal,
     &                StopRule, ERule, Step,SWolfe,IWolfe,AWolfe

      common /cgparms/delta, sigma, epsilon, theta, gamma, rho, tol,
     &               eta, fpert, f0, wolfe_hi, wolfe_lo,
     &               awolfe_hi, QuadCutOff, zero, feps,
     &               psi0, psi1, psi2,
     &               n, n5, n6, nf, ng, info,
     &               nrestart, nexpand, nsecant, maxit,
     &               QuadStep, PrintLevel, PrintFinal, StopRule,
     &               ERule, Step, QuadOK
```

```
      if ( dphi .ge. wolfe_lo ) then

c test original Wolfe conditions

            if(SWolfe) then
             if ( f-f0 .le. alpha*wolfe_hi ) then    ! Wolfe line search
                cg_Wolfe = .true.
                return
              end if
            end if

c test improved Wolfe

       if(IWolfe) then
       if(f-f0 .le. dmin1(epsilon*dabs(wolfe_hi/delta),
     &    0.0001d0*alpha*wolfe_hi/delta+etta)) then   ! Improved Wolfe
             cg_Wolfe = .true.
             return
       end if
     end if

c test approximate Wolfe conditions

           if ( AWolfe ) then   ! Approximate Wolfe line search

              if ( (f .le. fpert) .and. (dphi .le. awolfe_hi) ) then
                cg_Wolfe = .true.
                return
              endif
           endif
      endif

      cg_Wolfe = .false.

          return
         end


c                                 CG_TOL
c-----------------------------------------------------------------------


c check for convergence of the cg iterations

         logical function cg_tol (f, gnorm)

         double precision delta, sigma, epsilon, theta, gamma, rho, tol,
     &                eta, fpert, f0, wolfe_hi, wolfe_lo,
     &                awolfe_hi, QuadCutOff, zero, feps,
     &                psi0, psi1, psi2,
     &                f, gnorm

         integer          nrestart, nexpand, nsecant, maxit,
     &                n, n5, n6, nf, ng, info

         logical          QuadOK, QuadStep, PrintLevel, PrintFinal,
     &                StopRule, ERule, AWolfe, Step

      common /cgparms/delta, sigma, epsilon, theta, gamma, rho, tol,
     &               eta, fpert, f0, wolfe_hi, wolfe_lo,
     &               awolfe_hi, QuadCutOff, zero, feps,
     &               psi0, psi1, psi2,
```

```
     &                n, n5, n6, nf, ng, info,
     &                nrestart, nexpand, nsecant, maxit,
     &                QuadStep, PrintLevel, PrintFinal, StopRule,
     &                ERule, Step, QuadOK

         if ( StopRule ) then
             if ( gnorm .le. tol*(1.0 + dabs (f)) ) then
              cg_tol = .true.
                 return
             endif
         else
             if ( gnorm .le. tol ) then
              cg_tol = .true.
                 return
             endif
         endif

         cg_tol = .false.

         return
         end


c                                 CG_DOT
c-----------------------------------------------------------------------
c

c compute dot product of x and y

         double precision function cg_dot (x, y)

         double precision delta, sigma, epsilon, theta, gamma, rho, tol,
     &                eta, fpert, f0, wolfe_hi, wolfe_lo,
     &                awolfe_hi, QuadCutOff, zero, feps,
     &                psi0, psi1, psi2,
     &                x (1), y(1), t

         integer          nrestart, nexpand, nsecant, maxit,
     &                n, n5, n6, nf, ng, info, i

         logical          QuadOK, QuadStep, PrintLevel, PrintFinal,
     &                StopRule, ERule, AWolfe, Step

      common /cgparms/delta, sigma, epsilon, theta, gamma, rho, tol,
     &                eta, fpert, f0, wolfe_hi, wolfe_lo,
     &                awolfe_hi, QuadCutOff, zero, feps,
     &                psi0, psi1, psi2,
     &                n, n5, n6, nf, ng, info,
     &                nrestart, nexpand, nsecant, maxit,
     &                QuadStep, PrintLevel, PrintFinal, StopRule,
     &                ERule, Step, QuadOK

         t = zero
         do i = 1, n5
             t = t + x (i)*y (i)
         enddo
         do i = n6, n, 5
             t = t + x (i)*y(i) + x (i+1)*y (i+1) + x (i+2)*y (i+2)
     &                      + x (i+3)*y (i+3) + x (i+4)*y (i+4)
         enddo
      cg_dot = t
         return
         end
```

```
c                               CG_STEP
c-----------------------------------------------------------------------


c
c  compute xtemp = x + alpha d
c
         subroutine cg_step (xtemp, x, d, alpha)

         double precision delta, sigma, epsilon, theta, gamma, rho, tol,
     &                eta, fpert, f0, wolfe_hi, wolfe_lo,
     &                awolfe_hi, QuadCutOff, zero, feps,
     &                psi0, psi1, psi2,
     &                xtemp (1), x (1), d (1), alpha

         integer          nrestart, nexpand, nsecant, maxit,
     &                n, n5, n6, nf, ng, info, i, j

         logical          QuadOK, QuadStep, PrintLevel, PrintFinal,
     &                StopRule, ERule, AWolfe, Step

      common /cgparms/delta, sigma, epsilon, theta, gamma, rho, tol,
     &               eta, fpert, f0, wolfe_hi, wolfe_lo,
     &               awolfe_hi, QuadCutOff, zero, feps,
     &               psi0, psi1, psi2,
     &               n, n5, n6, nf, ng, info,
     &               nrestart, nexpand, nsecant, maxit,
     &               QuadStep, PrintLevel, PrintFinal, StopRule,
     &               ERule, Step, QuadOK

         do i = 1, n5
             xtemp (i) = x(i) + alpha*d(i)
         enddo
         do i = n6, n, 5
             xtemp (i) = x (i) + alpha*d (i)
             j = i + 1
             xtemp (j) = x (j) + alpha*d (j)
             j = i + 2
             xtemp (j) = x (j) + alpha*d (j)
             j = i + 3
             xtemp (j) = x (j) + alpha*d (j)
             j = i + 4
             xtemp (j) = x (j) + alpha*d (j)
         enddo

          return
         end


c                                 CG_LINE
c-----------------------------------------------------------------------


      subroutine cg_line (alpha, phi, dphi, dphi0, x, xtemp, d, gtemp,
     &              cg_value, cg_grad,SWolfe,IWolfe,AWolfe,etta,nexp)

         double precision delta, sigma, epsilon, theta, gamma, rho, tol,
     &                eta, fpert, f0, wolfe_hi, wolfe_lo,
     &                awolfe_hi, QuadCutOff, zero, feps,
     &                psi0, psi1, psi2,
     &                x (1), xtemp (1), d (1), gtemp (1),
```

```
     &                 a, dphia, b, dphib, alpha, phi, dphi, c,
     &                 a0, da0, b0, db0, width, fquad, dphi0,
     &                 cg_dot,etta

         integer          nrestart, nexpand, nsecant, maxit,
     &                 n, n5, n6, nf, ng, info,
     &                 ngrow, nshrink, cg_update, iter, flag

         logical          QuadOK, QuadStep, PrintLevel, PrintFinal,
     &                 StopRule, ERule, Step, cg_Wolfe,
     &                 SWolfe,IWolfe,AWolfe

         external      cg_value, cg_grad

      common /cgparms/delta, sigma, epsilon, theta, gamma, rho, tol,
     &                eta, fpert, f0, wolfe_hi, wolfe_lo,
     &                awolfe_hi, QuadCutOff, zero, feps,
     &                psi0, psi1, psi2,
     &                n, n5, n6, nf, ng, info,
     &                nrestart, nexpand, nsecant, maxit,
     &                QuadStep, PrintLevel, PrintFinal, StopRule,
     &                ERule, Step, QuadOK

      call cg_step (xtemp, x, d, alpha)
      call cg_grad (gtemp, xtemp, n, nexp)
         ng = ng + 1
         dphi = cg_dot (gtemp, d)
c
c Find initial interval [a,b] such that dphia < 0, dphib >= 0,
c        and phia <= phi0 + tolf*dabs (phi0)
c
         a = zero
         dphia = dphi0
         ngrow = 0
         nshrink = 0
         do while ( dphi .lt. zero )
          call cg_value (phi, xtemp, n, nexp)
             nf = nf + 1
c
c if quadstep in effect and quadratic conditions hold, check wolfe condition
c
             if ( QuadOK ) then
                 if ( ngrow .eq. 0 ) fquad = dmin1 (phi, f0)
                 if ( phi .le. fquad ) then
                     if ( PrintLevel ) then
                         write (*, 10) alpha, phi, fquad
   10                    format ('alpha:', e14.6, ' phi:', e14.6,
     &                        'fquad:', e14.6)
                     endif
      if(cg_Wolfe(alpha, phi, dphi,SWolfe,IWolfe,AWolfe,etta)) return
                 endif
             endif
             if ( phi .le. fpert ) then
                 a = alpha
                 dphia = dphi
             else
c
c contraction phase
c
                 b = alpha
                 do while ( .true. )
                     alpha = .5d0*(a+b)
                     nshrink = nshrink + 1
```

```
                    if ( nshrink .gt. nexpand ) then
                        info = 6
                        return
                    endif
                 call cg_step (xtemp, x, d, alpha)
                 call cg_grad (gtemp, xtemp, n, nexp)
                    ng = ng + 1
                    dphi = cg_dot (gtemp, d)
                    if ( dphi .ge. zero ) goto 100
                 call cg_value (phi, xtemp, n, nexp)
                    nf = nf + 1
                    if ( PrintLevel ) then
                        write (6, 20) a, b, alpha, phi, dphi
   20                   format ('contract, a:', e14.6,
     &                        ' b:', e14.6, ' alpha:', e14.6,
     &                        ' phi:', e14.6, ' dphi:', e14.6)
                    endif
                    if ( QuadOK .and. (phi .le. fquad) ) then
      if(cg_Wolfe (alpha, phi, dphi,SWolfe,IWolfe,AWolfe,etta)) return
                    endif
                    if ( phi .le. fpert ) then
                        a = alpha
                        dphia = dphi
                    else
                        b = alpha
                    endif
                 enddo
             endif
c
c expansion phase
c
             ngrow = ngrow + 1
             if ( ngrow .gt. nexpand ) then
                 info = 3
                 return
             endif
             alpha = rho*alpha
          call cg_step (xtemp, x, d, alpha)
          call cg_grad (gtemp, xtemp, n, nexp)
             ng = ng + 1
             dphi = cg_dot (gtemp, d)
             if ( PrintLevel ) then
                 write (*, 30) a, alpha, phi, dphi
   30            format ('expand,   a:', e14.6, ' alpha:', e14.6,
     &                 ' phi:', e14.6, ' dphi:', e14.6)
                 write (6, *) "expand, alpha:", alpha, "dphi:", dphi
             endif
         enddo
100   continue
         b = alpha
         dphib = dphi
         if ( QuadOK ) then
          call cg_value (phi, xtemp, n, nexp)
             nf = nf + 1
             if ( ngrow + nshrink .eq. 0 ) fquad = dmin1 (phi, f0)
             if ( phi .le. fquad ) then
      if(cg_Wolfe(alpha, phi, dphi,SWolfe,IWolfe,AWolfe,etta)) return
             endif
         endif
         do iter = 1, nsecant
             if ( PrintLevel ) then
                 write (*, 40) a, b, dphia, dphib
   40            format ('secant, a:', e14.6, ' b:', e14.6,
```

```
     &                 ' da:', e14.6, ' db:', e14.6)
            endif
            width = gamma*(b - a)
            if ( -dphia .le. dphib ) then
                alpha = a - (a-b)*(dphia/(dphia-dphib))
            else
                alpha = b - (a-b)*(dphib/(dphia-dphib))
            endif
            c = alpha
            a0 = a
            b0 = b
            da0 = dphia
            db0 = dphib
            flag = cg_update (a, dphia, b, dphib, alpha, phi,
     &              dphi, x, xtemp, d, gtemp, cg_value, cg_grad,
     &              SWolfe,IWolfe,AWolfe,etta,nexp)
            if ( flag .gt. 0 ) then
                return
            else if ( flag .eq. 0 ) then
                if ( c .eq. a ) then
                    if ( dphi .gt. da0 ) then
                        alpha = c - (c-a0)*(dphi/(dphi-da0))
                    else
                        alpha = a
                    endif
                else
                    if ( dphi .lt. db0 ) then
                        alpha = c - (c-b0)*(dphi/(dphi-db0))
                    else
                        alpha = b
                    endif
                endif
                if ( (alpha .gt. a) .and. (alpha .lt. b) ) then
                    if ( PrintLevel ) write (*, *) "2nd secant"
                    flag = cg_update (a, dphia, b, dphib, alpha, phi,
     &             dphi, x, xtemp, d, gtemp, cg_value, cg_grad,
     &             SWolfe,IWolfe,AWolfe,etta,nexp)
                    if ( flag .gt. 0 ) return
                endif
            endif
c
c    bisection iteration
c
            if ( (b-a) .ge. width ) then
                alpha = .5d0*(b+a)
                if ( PrintLevel ) write (*, *) "bisection"
                flag = cg_update (a, dphia, b, dphib, alpha, phi,
     &                dphi, x, xtemp, d, gtemp, cg_value, cg_grad,
     &                SWolfe,IWolfe,AWolfe,etta,nexp)
                if ( flag .gt. 0 ) return
            else
                if ( b .le. a ) then
                    info = 7
                    return
                endif
            endif
        end do
        info = 4
        return
        end


c                           CG_UPDATE
```

```
c-----------------------------------------------------------------------

c
c update returns 1 if Wolfe condition is satisfied or too many iterations
c        returns  0 if the interval updated successfully
c        returns -1 if search done
c
         integer function cg_update (a, dphia, b, dphib, alpha, phi,
     &               dphi, x, xtemp, d, gtemp, cg_value, cg_grad,
     &               SWolfe,IWolfe,AWolfe,etta,nexp)

         double precision delta, sigma, epsilon, theta, gamma, rho, tol,
     &                eta, fpert, f0, wolfe_hi, wolfe_lo,
     &                awolfe_hi, QuadCutOff, zero, feps,
     &                psi0, psi1, psi2,
     &                a, dphia, b, dphib, alpha, phi, dphi,
     &                x (1), xtemp (1), d (1), gtemp (1),
     &                cg_dot,etta

         integer          nrestart, nexpand, nsecant, maxit,
     &                n, n5, n6, nf, ng, info,
     &                nshrink

         logical          QuadOK, QuadStep, PrintLevel, PrintFinal,
     &                StopRule, ERule, Step, cg_Wolfe,
     &                SWolfe,IWolfe,AWolfe

         external      cg_value, cg_grad

      common /cgparms/delta, sigma, epsilon, theta, gamma, rho, tol,
     &               eta, fpert, f0, wolfe_hi, wolfe_lo,
     &               awolfe_hi, QuadCutOff, zero, feps,
     &               psi0, psi1, psi2,
     &               n, n5, n6, nf, ng, info,
     &               nrestart, nexpand, nsecant, maxit,
     &               QuadStep, PrintLevel, PrintFinal, StopRule,
     &               ERule, Step, QuadOK

      call cg_step (xtemp, x, d, alpha)
      call cg_value (phi, xtemp, n, nexp)
         nf = nf + 1
      call cg_grad (gtemp, xtemp, n, nexp)
         ng = ng + 1
         dphi = cg_dot (gtemp, d)
         if ( PrintLevel ) then
             write (*, 10) alpha, phi, dphi
   10        format ('update alpha:', e14.6, ' phi:', e14.6,
     &            ' dphi:', e14.6)
         endif
      cg_update = 0
      if(cg_Wolfe (alpha, phi, dphi,SWolfe,IWolfe,AWolfe,etta)) then
          cg_update = 1
             goto 110
         endif
         if ( dphi .ge. zero ) then
             b = alpha
             dphib = dphi
             goto 110
         else
             if ( phi .le. fpert ) then
                 a = alpha
                 dphia = dphi
```

```
                goto 110
            endif
        endif
        nshrink = 0
        b = alpha
        do while ( .true. )
            alpha = .5d0*(a+b)
            nshrink = nshrink + 1
            if ( nshrink .gt. nexpand ) then
                info = 8
             cg_update = 1
                goto 110
            endif
         call cg_step (xtemp, x, d, alpha)
         call cg_grad (gtemp, xtemp, n, nexp)
            ng = ng + 1
            dphi = cg_dot (gtemp, d)
         call cg_value (phi, xtemp, n, nexp)
            nf = nf + 1
            if ( PrintLevel ) then
                write (6, 20) a, alpha, phi, dphi
   20           format ('contract, a:', e14.6, ' alpha:', e14.6,
     &                 ' phi:', e14.6, ' dphi:', e14.6)
            endif
      if(cg_Wolfe(alpha, phi, dphi,SWolfe,IWolfe,AWolfe,etta)) then
             cg_update = 1
                goto 110
       endif
            if ( dphi .ge. zero ) then
                b = alpha
                dphib = dphi
                goto 100
            endif
            if ( phi .le. fpert ) then
                if ( PrintLevel ) then
                    write (6, *) "update a:", alpha, "dphia:", dphi
                endif
                a = alpha
                dphia = dphi
            else
                b = alpha
            endif
        enddo
100   continue
      cg_update = -1
110   continue
        if ( PrintLevel ) then
            write (*, 200) a, b, dphia, dphib, cg_update
200         format ('UP a:', e14.6, ' b:', e14.6,
     &             ' da:', e14.6, ' db:', e14.6, ' up:', i2)
        endif
        return
        end
c
c*********************************************************************
c Last Line of CG_DESCENT of Hager and Zhang (modified as above)
```

```
*                                                   November 26, 2018
*
* .-----------------------------------------------------------------.
* |                     Subroutine inipoint
* |                     ===================
* |
* | Subroutine for initial point specification.
* |This is a user subroutine:
* |
* | The calling sequence is:
* |  call inipoint(n,x,nexp)
* |where:
* |  n    (integer) the number of variables,
* |  x    (double)  array with the initial point.
* |  nexp (integer) parameter specifying the number of the
* |                 problem considered in a train of experiments.
* |
* |                                                 Dr.  Neculai Andrei
* .-------------------------------------------------------------------.


*****************************************************************
      subroutine inipoint(n,x, nexp)

C     This subroutine computes the initial point

      real*8 x(n)

      go to ( 1, 2, 3, 4, 5, 6, 7, 8, 9,10,
     *       11,12,13,14,15,16,17,18,19,20,
     *       21,22,23,24,25,26,27,28,29,30,
     *       31,32,33,34,35,36,37,38,39,40,
     *       41,42,43,44,45,46,47,48,49,50,
     *       51,52,53,54,55,56,57,58,59,60,
     *       61,62,63,64,65,66,67,68,69,70,
     *       71,72,73,74,75,76,77,78,79,80) nexp

1     continue
c                             Freudenstein & Roth - FREUROTH (CUTE)
        i=1
91      x(i)  =   0.5d0
        x(i+1)=  -2.d0
        i=i+2
        if(i.le.n) go to 91
      return

2     continue
c                             Extended White & Holst
        i=1
92      x(i)  =  -1.2d0
        x(i+1)=   1.d0
        i=i+2
        if(i.le.n) go to 92
      return

3     continue
c                           TR-WHITEHOLST
*                   Tridiagonal White & Holst (c=4)
        i=1
93      x(i)  = -1.2d0
        x(i+1)=  1.0d0
        i=i+2
```

```
        if(i.le.n) go to 93
       return

4     continue
c                               Extended Beale
        i=1
94      x(i)  =   1.d0
        x(i+1)=   0.8d0
        i=i+2
        if(i.le.n) go to 94
      return

5     continue
c                               Extended Powell
        i=1
95      x(i)  =  3.d0
        x(i+1)= -1.d0
        x(i+2)=  0.d0
        x(i+3)=  1.d0
        i=i+4
        if(i.le.n) go to 95
      return

6     continue
c                               Extended Maratos
        i=1
96     x(i)  =  0.1d0
        x(i+1)=  0.1d0
        i=i+2
        if(i.le.n) go to 96
      return

7     continue
c                               Extended Cliff
        i=1
97      x(i)  =   0.001d0
        x(i+1)=  -0.001d0
        i=i+2
        if(i.le.n) go to 97
      return

8     continue
c                               Extended Wood   WOODS (CUTE)
        i=1
98      x(i)  =  -3.d0
        x(i+1)=  -1.d0
        x(i+2)=  -3.d0
        x(i+3)=  -1.d0
        i=i+4
        if(i.le.n) go to 98
      return

9     continue
c                               Extended Hiebert
      do i=1,n
        x(i) = 5.001d0
      end do
      return

10    continue
c                               Extended Rosenbrock  SROSENBR (CUTE)
        i=1
910     x(i)  =  -1.2d0
```

```
        x(i+1)=   1.d0
        i=i+2
        if(i.le.n) go to 910
      return

11    continue
c                              Generalized Risenbrock GENROSNB (CUTE)
        i=1
911     x(i)  = -1.2d0
        x(i+1)=  1.1d0
        i=i+2
        if(i.le.n) go to 911
      return

12    continue
c                             Extended Himmelblau
      do i=1,n
        x(i) = 1.d0
      end do
      return

13    continue
c                              HIMMELBG (CUTE)
      do i=1,n
        x(i) = 1.5d0
      end do
      return

14    continue
c                              HIMMELBH (CUTE)
      do i=1,n
        x(i) = 0.8d0
      end do
      return

15    continue
c                             Extended Trigonometric  ET1
      do i=1,n
        x(i) = 0.2d0
      end do
      return

16    continue
c                             Extended Trigonometric  ET2
      do i=1,n
        x(i) = 0.2d0
      end do
      return

17    continue
c                             Extended Block Diagonal BD1
      do i=1,n
        x(i) = 1.0d0
      end do
      return

18    continue
c                             Extended Tridiagonal 1
      do i=1,n
        x(i) = 2.d0
      end do
      return
```

```
19    continue
c                               Extended Three Expo Terms
      do i=1,n
        x(i) = 0.1d0
      end do
      return

20    continue
c                               Generalized Tridiagonal 1
      do i=1,n
        x(i) = 2.d0
      end do
      return

21    continue
c                               Generalized Tridiagonal 2
      do i=1,n
        x(i) = -1.d0
      end do
      return

22     continue
c                               Tridiagonal Double Borded TR-DB1
      do i=1,n
        x(i) = -1.d0
      end do
      return

23    continue
c                               Broyden Pentadiagonal (CUTE)
      do i=1,n
        x(i) = -1.d0
      end do
      return

24    continue
c                               Extended PSC1
        i=1
924     x(i)  =  3.d0
        x(i+1)=  0.1d0
        i=i+2
        if(i.le.n) go to 924
      return

25    continue
c                               Perturbed Quadratic PQ1
      do i=1,n
        x(i) = 1.d0
      end do
      return

26    continue
c                               Perturbed Quadratic  PQ2
      do i=1,n
        x(i) = 0.5d0
      end do
      return
27    continue
c                               Almost Perturbed Quadratic
      do i=1,n
        x(i) = 0.5d0
      end do
      return
```

```
28    continue
c                               Almost Perturbed Quartic
      do i=1,n
        x(i) = 0.5d0
      end do
      return

29     continue
c                               Extended Penalty Function  U52
      do i=1,n
       x(i) = float(i)/100.d0
      end do
      return

30    continue
c                               TR-Summ of quadratics
      do i=1,n
        x(i) = 1.d0
      end do
      return

31    continue
c                               Quadratic Diagonal Perturbed
      do i=1,n
        x(i) = 0.5d0
      end do
      return


32    continue
*                                FH1
*   Full Hessian FH1 (Summ of Quadratics, Quadratic inside)
*
      do i=1,n
        x(i) = float(i)/float(n)
      end do
      return

33     continue
*                                FH2
*    Full Hessian FH2 (Quadratic, perturbed with sin/cos)
*
      do i=1,n
        x(i) = 1.d0
      end do
      return



34     continue
*                                FH3
*    Full Hessian FH3 (Quartic, perturbed with sin/cos)
*
      do i=1,n
        x(i) = 1.d0
      end do
      return

35    continue
c                               Diagonal Full Borded
      do i=1,n
        x(i) = 0.001d0
```

```
      end do
      return


36    continue
c                           D-DBAUP3
*                 Diagonal Double Borded Arrow Up
        i=1
936     x(i)  =  0.4d0
        x(i+1)=  1.0d0
        i=i+2
        if(i.le.n) go to 936
      return

37    continue
c                             QP1 Extended Quadratic Penalty
      do i=1,n
        x(i) = 1.d0
      end do
      return

38    continue
c                             QP2 Extended Quadratic Penalty
      do i=1,n
        x(i) = 2.d0
      end do
      return

39    continue
c                            QP3 Extended Quadratic Penalty
      do i=1,n
        x(i) = 1.d0
      end do
      return

40     continue
c                             STAIRCASE S1
       do i=1,n
         x(i) = 1.d0
       end do
       return

41     continue
c                             STAIRCASE S2
       do i=1,n
         x(i) = 1.d0
       end do
       return

42     continue
c                             STAIRCASE S3
       do i=1,n
         x(i) = 2.d0
       end do
       return

43    continue
c                           NONDQUAR
*              Tridiagonal Double Borded Arrow-Down
*
        i=1
943     x(i)  =  1.d0
        x(i+1)= -1.d0
```

```
        i=i+2
        if(i.le.n) go to 943
      return


44     continue
c                           TRIDIA
*                        Tridiagonal
*
       do i=1,n
         x(i) = 1.d0
       end do
       return

45     continue
c                           ARWHEAD
*                Diagonal Bouble borded Arrow Down
*
       do i=1,n
         x(i) = 1.d0
       end do
       return

46    continue
c                           NONDIA (CUTE)
*                   Diagonal Bouble borded Arrow Up
      do i=1,n
        x(i) = -0.01d0
      end do
      return

47    continue
c                            BDQRTIC (CUTE)
      do i=1,n
        x(i) = 1.d0
      end do
      return

48    continue
c                            DQDRTIC (CUTE)
      do i=1,n
        x(i) = 3.d0
      end do
      return

49    continue
c                            EG2 (CUTE)
      do i=1,n
        x(i) = 0.001d0
      end do
      return

50    continue
c                            EG3
      do i=1,n
        x(i) = 0.02d0
      end do
      return

51    continue
c                            EDENSCH (CUTE)
      do i=1,n
        x(i) = 0.d0
```

```
      end do
      return

52    continue
c                                FLETCHCR (CUTE)
      do i=1,n
        x(i) = 0.5d0
      end do
      return

53    continue
c                               ENGVAL1 (CUTE)
      do i=1,n
        x(i) = 2.d0
      end do
      return

54    continue
c                               DENSCHNA (CUTE)
      do i=1,n
        x(i) = 1.d0
      end do
      return

55    continue
c                               DENSCHNB (CUTE)
      do i=1,n
        x(i) = 10.d0
      end do
      return

56    continue
c                               DENSCHNC  (CUTE)
      do i=1,n
        x(i) = 1.d0
      end do
      return

57    continue
c                               DENSCHNF (CUTE)
        i=1
957     x(i)  =  100.d0
        x(i+1)= -100.d0
        i=i+2
        if(i.le.n) go to 957
      return

58    continue
c                               SINQUAD  (CUTE)
      do i=1,n
        x(i) = 0.00000d0
      end do
      return

59    continue
c                                DIXON3DQ (CUTE)
      do i=1,n
        x(i) = -0.1d0
      end do
      return

60    continue
c                                BIGGSB1 (CUTE)
```

```
      do i=1,n
        x(i) = 0.1d0
      end do
      return

61    continue
c                                PRODsin (m=n-1)
      do i=1,n
        x(i) = 0.000001d0
      end do
      return


62    continue
c                                PROD1 (m=n)
      do i=1,n
        x(i) = 1.d0
      end do
      return

63    continue
c                                PRODcos (m=n-1)
      do i=1,n
        x(i) = 0.1d0
      end do
       x(1)=1.d0
      return

64    continue
c                                PROD2 (m=1)
      do i=1,n
        x(i) = 0.00001d0
      end do
       x(n)=1.d0
      return

65    continue
c                                DIXMAANA (CUTE)
      do i=1,n
        x(i) = 2.d0
      end do
      return

66    continue
c                                DIXMAANB (CUTE)
      do i=1,n
        x(i) = 2.d0
      end do
      return

67    continue
c                                DIXMAANC (CUTE)
      do i=1,n
        x(i) = 2.d0
      end do
      return

68    continue
c                                DIXMAAND (CUTE)
      do i=1,n
        x(i) = 2.d0
      end do
      return
```

```
69    continue
c                               DIXMAANL (CUTE)
      do i=1,n
        x(i) = 1.d0
      end do
      return

70    continue
c                               ARGLINB (m=5)
        i=1
970     x(i)  =  0.01d0
        x(i+1)=  0.001d0
        i=i+2
        if(i.le.n) go to 970
      return

71    continue
c                               VARDIM (CUTE)
      do i=1,n
        x(i) = 1.d0-float(i)/float(n)/100000.d0
c        x(i)=1.d0
      end do
      return

72    continue
c                               DIAG-AUP1
      do i=1,n
        x(i) = 4.d0
      end do
      return

73    continue
c                               ENGVAL8
      do i=1,n
        x(i) = 2.d0
      end do
      return

74     continue
c                                QUARTIC (CUTE)
      do i=1,n
        x(i) = 2.d0
      end do
      return

75     continue
c                                LIARWHD
      do i=1,n
        x(i) = 4.d0
      end do
      return

76     continue
c                                NONSCOMP
      do i=1,n
        x(i) = 3.d0
      end do
      return


77     continue
c                                Linear perturbed
```

```
      do i=1,n
        x(i) = 2.d0
      end do
      return

78     continue
c                               CUBE
        i=1
978     x(i)  = -1.2d0
        x(i+1)=  1.1d0
        i=i+2
        if(i.le.n) go to 978
      return

79     continue
c                               HARKERP
      do i=1,n
        x(i) = 2.d0
      end do
      return

80     continue
c                               QUARTICM
      do i=1,n
        x(i) = 2.d0
      end do
      return

       end
c-------------------------------------------------------------Last line INIPOINT




*****************************************************************
*                              Date created:  November 26, 2018.
*
*
*       FUNCTIONS FOR UNCONSTRAINED OPTIMIZATION
*      ==========================================
*
*                                                Dr.  Neculai Andrei
*****************************************************************
*
      subroutine cg_value(f, x, n, nexp)
      real*8 x(n), f

      real*8 t1,t2,t3,t4, c, d
      real*8 s, temp(1000000), temp1, tsum, sum
      real*8 u(1000000), v(1000000), t(1000000)
      real*8 u1, v1, c1, c2
      real*8 alpha, beta, gamma, delta

      integer k1, k2, k3, k4

*
      go to ( 1, 2, 3, 4, 5, 6, 7, 8, 9,10,
     *       11,12,13,14,15,16,17,18,19,20,
     *       21,22,23,24,25,26,27,28,29,30,
     *       31,32,33,34,35,36,37,38,39,40,
     *       41,42,43,44,45,46,47,48,49,50,
     *       51,52,53,54,55,56,57,58,59,60,
```

```
     *       61,62,63,64,65,66,67,68,69,70,
     *       71,72,73,74,75,76,77,78,79,80) nexp

cF1                                 FREUROTH (CUTE)
*                           Extended Freudenstein & Roth
*
*                       Initial Point: [0.5, -2, ...,0.5, -2].
*

1     continue
      f = 0.d0

      do i=1,n/2
        t1=-13.d0+x(2*i-1)+5.d0*x(2*i)*x(2*i)-x(2*i)**3-2.d0*x(2*i)
        t2=-29.d0+x(2*i-1)+x(2*i)**3+x(2*i)**2-14.d0*x(2*i)

        f = f + t1*t1 + t2*t2
      end do

      return


cF2                     Extended White & Holst function
*
*           Initial point: [-1.2, 1, -1.2, 1, ........., -1.2, 1]

2     continue
      c=1.d0

      f=0.d0
      do i=1,n/2
        f = f + c*(x(2*i)-x(2*i-1)**3)**2 + (1.d0-x(2*i-1))**2
      end do
      return


cF3                           TR-WHITEHOLST
*                     Tridiagonal. White-Holst (c=4)
*             Initial point x =[-1.2,  1, ..., -1.2, 1]
*
3     continue
      c = 4.d0
      f = 0.d0

      do i=1,n-1
        f = f + c*(x(i+1)-x(i)**3)**2 + (1.d0-x(i))**2
      end do
      return

cF4                   Extended Beale Function  BEALE (CUTE)
*
*                    Initial Point: [1, 0.8, ...., 1, 0.8]
*
4     continue
*
      f=0.d0

      do i=1,n/2
        t1=1.5d0  -x(2*i-1)+x(2*i-1)*x(2*i)
        t2=2.25d0 -x(2*i-1)+x(2*i-1)*x(2*i)*x(2*i)
        t3=2.625d0-x(2*i-1)+x(2*i-1)*x(2*i)*x(2*i)*x(2*i)

        f = f + t1*t1 + t2*t2 + t3*t3
```

```
      end do
      return

cF5                              Extended Powell
*
*                      Initial Point: [3, -1, 0, 1, ......].
*
5     continue

      f=0.d0

      do i=1,n/4
        t1= x(4*i-3) + 10.d0*x(4*i-2)
        t2= x(4*i-1) - x(4*i)
        t3= x(4*i-2) - 2.d0*x(4*i-1)
        t4= x(4*i-3) - x(4*i)

        f = f + t1*t1 + 5.d0*t2*t2 + t3**4 + 10.d0*t4**4
      end do
      return


cF6                           Extended Maratos Function
*
*                      Initial Point: [1.1, 0.1, ...,1.1, 0.1].
*
6     continue
      c = 1.d0
      f = 0.d0

      do i=1,n/2
        t1 = x(2*i-1)**2 + x(2*i)**2 - 1.d0

        f = f + (x(2*i-1) + c*t1*t1)
      end do
      return


cF7                              Extended CLIFF (CUTE)
*
*                       Initial Point:  [0, -0.1, ......, 0, -0.1].
*
7     continue

      f=0.d0

      do i=1,n/2
       temp1 = (x(2*i-1)-3.d0)/100.d0

      f = f+temp1*temp1-(x(2*i-1)-x(2*i))+dexp(2.d0*(x(2*i-1)-x(2*i)))
      end do
      return

cF8                           Extended Wood Function
*                                   WOODS (CUTE)
*
*                         Initial Point: [-3,-1,-3,-1,......]
8     continue

      f=0.d0

      do i=1,n/4
        f = f + 100.d0*(x(4*i-3)**2-x(4*i-2))**2
```

```
     *        +        (x(4*i-3)-1.d0)**2
     *        +  90.d0*(x(4*i-1)**2-x(4*i))**2
     *        +        (1.d0-x(4*i-1))**2
     *        +  10.1d0*(x(4*i-2)-1.d0)**2
     *        +  10.1d0*(x(4*i)  -1.d0)**2
     *        +  19.8d0*(x(4*i-2)-1.d0)*(x(4*i)-1.d0)
      end do
      return


cF9                          Extended Hiebert Function
*
*                             Initial Point: [0,0,...0].
9     continue

      c1 = 10.d0
      c2 = 500.d0

      f = 0.d0

      do i=1,n/2
        f = f + (x(2*i-1)-c1)**2 + (x(2*i-1)*x(2*i)-c2)**2
      end do
      return


cF10                          SROSENBR (CUTE)
*                     Extended Rosenbrock function
*
* Initial point: [-1.2, 1, -1.2, 1, ........., -1.2, 1]

10    continue
      c=1000.d0

      f=0.d0
      do i=1,n/2
        f = f + c*(x(2*i)-x(2*i-1)**2)**2 + (1.d0-x(2*i-1))**2
      end do
      return


cF11                          GENROSNB (CUTE)
*                          Generalized Rosenbrock
*                    Initial Point: [-1.2, 1, ...  -1.2, 1]

11    continue

      f = (x(1)-1.d0)**2
      do i=2,n
        f = f + 100.d0*(x(i)-x(i-1)**2)**2
      end do
      return


cF12                             HIMMELBC (CUTE)
*                         Extended Himmelblau Function
*
*                           Initial Point: [1, 1, ....., 1]

12    continue

      f=0.d0
```

```
      do i=1,n/2
        u1 = x(2*i-1)**2 + x(2*i)    - 11.d0
        v1 = x(2*i-1)    + x(2*i)**2 - 7.d0

        f = f + u1*u1 + v1*v1
      end do
      return


cF13                          HIMMELBG (CUTE)
*
*                     Initial Point: [1.5,1.5,...,1.5]

13    continue

      f=0.d0
      do i=1,n/2
        f = f + (2.d0*x(2*i-1)**2+3.d0*x(2*i)**2)*
     *          (dexp(-x(2*i-1)-x(2*i)))
      end do
      return


cF14                          HIMMELBH (CUTE)
*
*                     Initial Point: [1.5,1.5,...,1.5]

14    continue

      f= 0.d0
      do i=1,n/2
        f=f+(-3.d0*x(2*i-1)-2.d0*x(2*i)+2.d0+x(2*i-1)**3 + x(2*i)**2)
      end do
      return


cF15                            Extended Trigonometric ET1
*
*                          Initial Point: [0.2, 0.2, ....,0.2].

15    continue
      s= float(n)
      do i=1,n
        s = s - dcos(x(i))
      end do

      do i=1,n
        temp(i) = s + float(i)*(1.d0-dcos(x(i))) - dsin(x(i))
      end do

      f = 0.d0
      do i=1,n
        f = f + temp(i)**2
      end do
      return


cF16                            Extended Trigonometric ET2
*
*                          Initial Point: [0.2, 0.2, ....,0.2].

16    continue
      s= float(n)
```

```
      do i=1,n
        s = s - dsin(x(i))
      end do

      do i=1,n
        temp(i) = s + float(i)*(1.d0-dsin(x(i))) - dsin(x(i))
      end do

      f = 0.d0
      do i=1,n
        f = f + temp(i)**2
      end do
      return


cF17                  Extended Block Diagonal BD1 Function
*
*                     Initial Point: [0.1, 0.1, ..., 0.1].
*
17    continue

      f = 0.d0

      do i=1,n/2
        t1 = x(2*i-1)**2 + x(2*i)**2 - 2.d0
        t2 = dexp(x(2*i-1)) - x(2*i)

        f = f + t1*t1 + t2*t2
      end do
      return


cF18                   Extended Tridiagonal-1 Function
*
*                        Initial Point: [2,2,...,2]
18    continue

      f=0.d0
      do i=1,n/2
        u(i) = x(2*i-1) + x(2*i) - 3.d0
        v(i) = x(2*i-1) - x(2*i) + 1.d0
      end do
      do i=1,n/2
        f = f + u(i)**2 + v(i)**4
      end do
      return


cF19                         Extended Three Exponential Terms
*
*                           Intial Point: [0.1,0.1,.....,0.1].
*
19    continue
      f=0.d0

      do i=1,n/2
        t1= x(2*i-1) + 3.d0*x(2*i) - 0.1d0
        t2= x(2*i-1) - 3.d0*x(2*i) - 0.1d0
        t3=-x(2*i-1) - 0.1d0

        f = f + dexp(t1) + dexp(t2) + dexp(t3)
      end do
      return
```

```
cF20                 Generalized Tridiagonal-1 Function
*
*                        Initial Point: [2,2,...,2]
20    continue

      f=0.d0
      do i=1,n-1
        u(i) = x(i) + x(i+1) - 3.d0
        v(i) = x(i) - x(i+1) + 1.d0
      end do

      do i=1,n-1
        f = f + u(i)**2 + v(i)**4
      end do
      return


cF21                       Generalized Tridiagonal-2
*                                Penta Diagonal
*                    Initial point: [-1, -1, ........., -1., -1]
21    continue
      f = 0.d0
      u(1) = 5.d0*x(1)-3.d0*x(1)**2-x(1)**3-3.d0*x(2)+1.d0

      do i=2,n-1
        u(i)=5.d0*x(i)-3.d0*x(i)**2-x(i)**3-x(i-1)-3.d0*x(i+1)+1.d0
      end do

      u(n)=5.d0*x(n)-3.d0*x(n)**2-x(n)**3-x(n-1)+1.d0

      do i=1,n
        f = f + u(i)**2
        v(i) = 5.d0 -6.d0*x(i) -3.d0*x(i)**2
      end do
      return


cF22                         TR-DB1
*                   Tridiagonal Double Borded
*               Initial Point:  [-1, -1, ...., -1]
*
22    continue
       f = (x(1)-1.d0)**2

       do i=1,n-1
         temp(i) = x(1) - 0.5d0*x(i)**2 - 0.5d0*x(i+1)**2
         f = f + temp(i)*temp(i)
      end do
      return


cF23                       Broyden Pentadiagonal
*
*                  Initial point x0=[-1., -1., ..., -1.].
*
23    continue
*

      temp(1) = 3.d0*x(1) - 2.d0*x(1)*x(1)
      do i=2,n-1
        temp(i) = 3.d0*x(i)-2.d0*x(i)*x(i)-x(i-1)-2.d0*x(i+1)+1.d0
```

```
      end do

      temp(n) = 3.d0*x(n)-2.d0*x(n)*x(n)-x(n-1)+1.d0

      f = 0.d0

      do i=1,n
        f = f + temp(i)*temp(i)
      end do
      return


cF24                        Extended PSC1 Function
*
*                    Initial point: [3, 0.1, ..., 3, 0.1]

24    continue

      f = 0.d0
      do i=1,n/2
        f = f + (x(2*i-1)**2 +x(2*i)**2 +x(2*i-1)*x(2*i))**2
     *        + (dsin(x(2*i-1)))**2 + (dcos(x(2*i)))**2
      end do
      return


cF25                         Perturbed Quadratic function PQ1
*
*                        Initial Point:  [1, 1, ......,1].
*
25    continue

      temp1 = 0.d0
      do i=1,n
        temp1 = temp1 + x(i)
      end do

      f = temp1*temp1/100.d0

      do i=1,n
        f = f + float(i)*x(i)**2
      end do

      return


cF26                        Perturbed Quadratic function PQ2
*
*                      Initial Point:  [0.5, 0.5, ......, 0.5].
*
26    continue

      temp1 = 0.d0
      do i=1,n
        temp1 = temp1 + float(i)*x(i)
      end do

      f = temp1*temp1

      do i=1,n
        f = f + float(i)*x(i)**2
      end do
      return
```

```
cF27                    Almost Perturbed Quadratic
*
*                  Initial point x0=[0.5, 0.5, ...,0.5].
*
27    continue
*
      f = ((x(1)+x(n))**2)/100.d0

      do i=1,n
        f = f + float(i)*x(i)*x(i)
      end do
      return


cF28                    Almost Perturbed Quartic
*
*                  Initial point x0=[0.5, 0.5, ...,0.5].
*
28    continue
*

      f = ((x(1)+x(n))**2)/100.d0

      do i=1,n
        f = f + float(i)*x(i)**4
      end do
      return


cF29                    Extended Penalty Function  U52 (MatrixRom)
*
*                           Intial Point: [1,2,3,.....,n].
*
29    continue
      temp1=0.d0
      do i=1,n
        temp1 = temp1 + x(i)**2
      end do
*
      f = (temp1 - 0.25d0)**2

      do i=1,n-1
        f = f + (x(i)-1.d0)**2
      end do
      return


cF30                              TR-Summ of quadratics Function
*
*                                Initial Point: [1, 1, ....., 1]
*
30    continue

      c = 100000.d0
      f = 0.d0

      do i=1,n-1
        f = f + x(i)*x(i) + c*(x(i+1)+x(i)*x(i))**2
      end do
      return
```

```
cF31                    Quadratic Diagonal Perturbed Function
*
*                      Initial Point:  [0.5, 0.5, ......, 0.5].
*
31    continue

      temp1 = 0.d0
      do i=1,n
        temp1 = temp1 + x(i)
      end do

      f = temp1*temp1

      do i=1,n
        f = f + (float(i)/100.d0) * x(i)**2
      end do
      return


cF32                          FH1 (m=50)
*        Full Hessian FH1 (Summ of Quadratics, Quadratic inside)
*
32    continue
      m=50
      f=0.d0

      do i=1,m
        u(i)=0.d0
        do j=1,n
          u(i) = u(i) + float(i)*float(j)*x(j)*x(j)
        end do

        f = f + (u(i)-1.d0)**2
      end do
      return


cF33                         FH2
*      Full Hessian FH2 (Quadratic, perturbed with sin/cos)
*           Initial Point:  [1, 1, ......, 1].
*
33    continue
       s = 0.d0
       do i=1,n
         s = s + x(i)
      end do

       f = s*s

       do i=1,n
         f = f + float(i)*(dsin(x(i)) + dcos(x(i)))/1000.d0
      end do
     return


cF34                         FH3
*      Full Hessian FH3 (Quartic, perturbed with sin/cos)
*           Initial Point:  [1, 1, ......, 1].
*
34    continue
       s = 0.d0
       do i=1,n
```

```
         s = s + x(i)**2
      end do

       f = s*s

       do i=1,n
         f = f + float(i)*(dsin(x(i)) + dcos(x(i)))/1000.d0
      end do
      return


cF35                        Diagonal Full Borded
*
*                Initial point: [0.1, 0.1, ..........., 0.1]

35    continue
      f=(x(1)-1.d0)**4 + (x(n)**2-x(1)**2)**2

      do i=1,n-2
        temp(i) = sin(x(i+1)-x(n)) - x(1)**2 - x(i+1)**2
        f = f + temp(i)*temp(i)
      end do
      return


cF36                          D-DBAUP3
*                 Diagonal Double Borded Arrow Up
*               Initial point: x0=[4, 0, ....,4,0]
*
36    continue
      f=0.d0

      do i=1,n
        f = f + 4.d0*(x(i)*x(i) - x(1))**2 + (x(i)-1.d0)**2
      end do
      return


cF37                   QP1 Extended Quadratic Penalty
*
*                     Initial Point:  [1, 1, ......,1].
*
37    continue

      t1=0.d0
      do i=1,n
        t1 = t1 + x(i)*x(i)
      end do
      t1 = t1 - 0.5d0

       f = 0.d0
      do i=1,n-1
        f = f + (x(i)*x(i) - 2.d0)**2
      end do

      f = f + t1*t1
      return


cF38                  QP2 Extended Quadratic Penalty Function
*
*                      Initial Point:  [1, 1, ......,1].
*
```

```
38    continue

      t1=0.d0
      do i=1,n
        t1 = t1 + x(i)*x(i)
      end do
      t1 = t1 - 100.d0

      f = 0.d0
      do i=1,n-1
        f = f + (x(i)*x(i) - dsin(x(i)))**2
      end do

      f = f + t1*t1
      return


cF39                   QP3 Extended Quadratic Penalty
*
*                     Initial Point: [1., 1., ....,1.].

39    continue

      t1=0.d0
      do i=1,n
        t1 = t1 + x(i)*x(i)
      end do
      t1 = t1 - 0.25d0

       f = t1*t1

      do i=1,n-1
        f = f - (x(i)*x(i) - 1.d0)**2
      end do
      return


cF40                            STAIRCASE S1
*
*                      Initial point x0=[1,1,...,1].
*
40    continue

       f=0.d0
      do i=1,n-1
        f = f + (x(i)+x(i+1)-float(i))**2
      end do
      return


cF41                            STAIRCASE S2
*
*                      Initial point x0=[1,1,...,1].
*
41    continue
       f = 0.d0
       do i=2,n
         f = f + (x(i-1)+x(i)-float(i))**2
       end do
       return


cF42                            STAIRCASE S3
```

```
*
*                      Initial point x0=[2,2,...,2].
*
42    continue
       f = 0.d0
       do i=2,n
         f = f + (x(i-1)+x(i)+float(i))**2
       end do
       return


cF43                          NONDQUAR
*              Tridiagonal Double Borded Arrow-Down
*
43    continue

      f = (x(1)-x(2))**2 + (x(n-1)+x(n))**2

      do i=1,n-2
        f = f + (x(i)+x(i+1)+x(n))**4
      end do
      return


cF44                             TRIDIA  (CUTE)
*
*                     Initial point x0=[1,1,...,1].
*
44    continue
*
      alpha=5.d0
      beta =1.d0
      gamma=1.d0
      delta=1.d0

      f=gamma*(delta*x(1)-1.d0)**2

      do i=2,n
        f = f + float(i)*(alpha*x(i)-beta*x(i-1))**2
      end do
      return


cF45                         ARWHEAD  (CUTE)
*
*                     Initial point x0=[1,1,...,1].
*
45    continue

      f=0.d0
      do i=1,n-1
        f = f + (-4.d0*x(i)+3.d0) + (x(i)**2+x(n)**2)**2
      end do
      return


cF46
*                       NONDIA (Shanno-78)  (CUTE)
*
*                     Initial point x0=[-1,-1,...,-1].
*
*
46    continue
```

```
      c=100.d0

      f=(x(1)-1.d0)**2 + c*(x(1)-x(1)**2)**2

      do i=2,n
        f = f + c*(x(1)-x(i)**2)**2
      end do
      return


cF47                            BDQRTIC (CUTE)
*
*                     Initial point x0=[1.,1.,...,1.].
*
*
47    continue
*
      n4=n-4
      f=0.d0

      do i=1,n4
        temp(i) = x(i)**2 + 2.d0*x(i+1)**2 + 3.d0*x(i+2)**2
     *                    + 4.d0*x(i+3)**2 + 5.d0*x(n)**2
      end do

      do i=1,n4
        f = f + (-4.d0*x(i)+3.d0)**2 + temp(i)**2
      end do
      return


cF48                            DQDRTIC  (CUTE)
*
*                     Initial point x0=[3,3,3...,3].
*
*
48    continue

      c=1000.d0
      d=1000.d0

      f=0.d0
      do i=1,n-2
        f = f + (x(i)**2 + c*x(i+1)**2 + d*x(i+2)**2)
      end do
      return


cF49                            EG2 (CUTE)
*
*                     Initial point x0=[1,1,1...,1].
*
*
49    continue

      f=0.5d0*dsin(x(n)*x(n))
      do i=1,n-1
        f = f + dsin(x(1)+x(i)*x(i)-1.d0)
      end do
      return
```

```
cF50                          EG3
*
*                  Initial point x0=[1,1,1...,1].
*
*
50    continue

      f=0.5d0*dcos(x(n)*x(n))

      do i=1,n-1
        f = f + dcos(x(1)+x(i)*x(i)-1.d0)
      end do
      return


cF51                       EDENSCH Function  (CUTE)
*
*                      Initial Point: [0., 0., ..., 0.].
51    continue

      f = 16.d0

      do i=1,n-1
        f = f + (x(i)-2.d0)**4 +
     *          (x(i)*x(i+1)-2.d0*x(i+1))**2 +
     *          (x(i+1)+1.d0)**2
      end do
      return


cF52                       FLETCHCR (CUTE)
*
*                  Initial Point: [0.5,0.5,...0.5]
52    continue

      f=0.d0
      do i=1,n-1
        f = f + 100.d0*(x(i+1)-x(i)+1.d0-x(i)*x(i))**2
      end do
      return


cF53                         ENGVAL1 (CUTE)
*
*                    Initial point x0=[2.,2.,2...,2.].
*
53    continue

      do i=1,n-1
        t(i) = x(i)*x(i) + x(i+1)*x(i+1)
      end do

      f = 0.d0

      do i=1,n-1
        f = f + t(i)*t(i) + (-4.d0*x(i) + 3.d0)
      end do
      return


cF54                         DENSCHNA (CUTE)
*
*                       Initial point: [8, 8,...,8]
```

```
*
54    continue

      f=0.d0
      do i=1,n/2
        f = f + x(2*i-1)**4 +
     *          (x(2*i-1)+x(2*i))**2 +
     *          (-1.d0+dexp(x(2*i)))**2
      end do
      return


cF55                    DENSCHNB  (CUTE)
*
*               Initial point: [0.1, 0.1,...,0.1]

55    continue

      f=0.d0
      do i=1,n/2
        f = f + (x(2*i-1)-2.d0)**2 +
     *          ((x(2*i-1)-2.d0)**2)*(x(2*i)**2) +
     *          (x(2*i)+1.d0)**2
      end do
      return


cF56                    DENSCHNC  (CUTE)
*
*               Initial point: [8, 8,...,8]

56    continue

      f=0.d0
      do i=1,n/2
        f = f + (-2.d0+x(2*i-1)**2+x(2*i)**2)**2 +
     *          (-2.d0+dexp(x(2*i-1)-1.d0)+x(2*i)**3)**2
      end do
      return



cF57                   DENSCHNF  (CUTE)
*
*               Initial point: [2,0,2,0,...,2,0]

57    continue

      f=0.d0
      do  i=1,n/2
        f=f+(2.d0*(x(2*i-1)+x(2*i))**2+(x(2*i-1)-x(2*i))**2-8.d0)**2+
     *      (5.d0*x(2*i-1)**2+(x(2*i)-3.d0)**2-9.d0)**2
      end do
      return


cF58                      SINQUAD (CUTE)
*
*             Initial Point: [0.1, 0.1, ..., 0.1]

58    continue

      f=(x(1)-1.d0)**4 + (x(n)**2-x(1)**2)**2
```

```
      do i=1,n-2
        t(i) = dsin(x(i+1)-x(n)) - x(1)**2 + x(i+1)**2
        f = f + t(i)*t(i)
      end do
      return


cF59                        DIXON3DQ  (CUTE)
*
*                  Initial Point x0=[-1, -1,..., -1]
*
59    continue

      f=(x(1)-2.d0)**2

      do i=1,n-1
        f = f + (x(i)-x(i+1))**2
      end do

      f = f + (x(n)-1.d0)**2
      return


cF60                      BIGGSB1 (CUTE)
*
*               Initial Point: [0., 0., ....,0.]

60    continue

      f=(x(1)-1.d0)**2 + (1.d0-x(n))**2
      do i=2,n
        f = f + (x(i)-x(i-1))**2
      end do
      return
cF61                           PRODsin (m=n-1)
*
*                Initial point x0=[5. 5, 5, 5,...,5].
*
61    continue

       m = n-1
       t1=0.d0
       t2=0.d0

       do i=1,m
         t1 = t1 + x(i)*x(i)
       end do

       do i=1,n
         t2 = t2 + dsin(x(i))
       end do

       f = t1*t2
       return


cF62                           PROD1 (m=n)
*
*              Initial point x0=[1. 1, 1, 1,...,1].
*
62    continue
```

```
      m = n
      t1=0.d0
      t2=0.d0

      do i=1,m
        t1 = t1 + x(i)
      end do

      do i=1,n
        t2 = t2 + x(i)
      end do

      f = t1*t2
      return


cF63                          PRODcos (m=n-1)
*
*                Initial point x0=[5. 5, 5, 5,...,5].
*
63    continue

c     m = n-1
      m=n/2
      t1=0.d0
      t2=0.d0

      do i=1,m
        t1 = t1 + x(i)*x(i)
      end do

      do i=1,n
        t2 = t2 + dcos(x(i))
      end do

      f = t1*t2
      return


cF64                          PROD2 (m=1)
*
*              Initial point x0=[15. 15, 15, 15,...,15].
*
64    continue

      m = 1
      t1=0.d0
      t2=0.d0

      do i=1,m
        t1 = t1 + x(i)**4
      end do

      do i=1,n
        t2 = t2 + float(i)*x(i)
      end do

      f = t1*t2
      return


cF65                        DIXMAANA (CUTE)
*
```

```
*                    Initial point x0=[2.,2.,2...,2.].
*                           Modified m=n/4
65    continue
*
        alpha = 1.d0
        beta  = 0.d0
        gamma = 0.125d0
        delta = 0.125d0

        k1 = 0
        k2 = 0
        k3 = 0
        k4 = 0

        m = n/4

       f = 1.d0

       do i=1,n
         f = f + alpha * x(i)*x(i)*((float(i)/float(n))**k1)
       end do

      do i=1,n-1
        f = f + beta * x(i)*x(i)*((x(i+1)+x(i+1)*x(i+1))**2) *
     *          ((float(i)/float(n))**k2)
      end do

      do i=1,2*m
        f = f + gamma * x(i)*x(i) * (x(i+m)**4) *
     *          ((float(i)/float(n))**k3)
      end do

      do i=1,m
        f = f + delta * x(i) * x(i+2*m) *
     *          ((float(i)/float(n))**k4)
      end do
      return


cF66                        DIXMAANB (CUTE)
*
*                    Initial point x0=[2.,2.,2...,2.].
*                           Modified m=n/4
66    continue
*
        alpha = 1.d0
        beta  = 0.0625d0
        gamma = 0.0625d0
        delta = 0.0625d0

        k1 = 0
        k2 = 0
        k3 = 0
        k4 = 1

        m = n/4
       f = 1.d0

       do i=1,n
         f = f + alpha * x(i)*x(i)*((float(i)/float(n))**k1)
       end do

      do i=1,n-1
```

```
        f = f + beta * x(i)*x(i)*((x(i+1)+x(i+1)*x(i+1))**2) *
     *          ((float(i)/float(n))**k2)
      end do

      do i=1,2*m
        f = f + gamma * x(i)*x(i) * (x(i+m)**4) *
     *          ((float(i)/float(n))**k3)
      end do

      do i=1,m
        f = f + delta * x(i) * x(i+2*m) *
     *          ((float(i)/float(n))**k4)
      end do
      return

cF67                        DIXMAANC (CUTE)
*
*                    Initial point x0=[2.,2.,2...,2.].
*                           Modified m=n/4
67    continue
*
        alpha = 1.d0
        beta  = 0.125d0
        gamma = 0.125d0
        delta = 0.125d0

        k1 = 0
        k2 = 0
        k3 = 0
        k4 = 0

        m = n/4
        f = 1.d0

       do i=1,n
         f = f + alpha * x(i)*x(i)*((float(i)/float(n))**k1)
       end do

      do i=1,n-1
        f = f + beta * x(i)*x(i)*((x(i+1)+x(i+1)*x(i+1))**2) *
     *          ((float(i)/float(n))**k2)
      end do

      do i=1,2*m
        f = f + gamma * x(i)*x(i) * (x(i+m)**4) *
     *          ((float(i)/float(n))**k3)
      end do

      do i=1,m
        f = f + delta * x(i) * x(i+2*m) *
     *          ((float(i)/float(n))**k4)
      end do
      return


cF68                        DIXMAAND (CUTE)
*
*                    Initial point x0=[2.,2.,2...,2.].
*                           Modified m=n/4
68    continue
*
        alpha = 1.d0
        beta  = 0.26d0
```

```
        gamma = 0.26d0
        delta = 0.26d0

        k1 = 0
        k2 = 0
        k3 = 0
        k4 = 0

        m = n/4
        f = 1.d0

       do i=1,n
         f = f + alpha * x(i)*x(i)*((float(i)/float(n))**k1)
       end do

      do i=1,n-1
        f = f + beta * x(i)*x(i)*((x(i+1)+x(i+1)*x(i+1))**2) *
     *          ((float(i)/float(n))**k2)
      end do

      do i=1,2*m
        f = f + gamma * x(i)*x(i) * (x(i+m)**4) *
     *          ((float(i)/float(n))**k3)
      end do

      do i=1,m
        f = f + delta * x(i) * x(i+2*m) *
     *          ((float(i)/float(n))**k4)
      end do
      return


cF69                        DIXMAANL (CUTE)
*
*                    Initial point x0=[2.,2.,2...,2.].
*                           Modified m=n/4
69    continue
*
        alpha = 1.d0
        beta  = 0.26d0
        gamma = 0.26d0
        delta = 0.26d0

        k1 = 2
        k2 = 0
        k3 = 0
        k4 = 2

        m = n/4
       f = 1.d0

       do i=1,n
         f = f + alpha * x(i)*x(i)*((float(i)/float(n))**k1)
       end do

      do i=1,n-1
        f = f + beta * x(i)*x(i)*((x(i+1)+x(i+1)*x(i+1))**2) *
     *          ((float(i)/float(n))**k2)
      end do

      do i=1,2*m
        f = f + gamma * x(i)*x(i) * (x(i+m)**4) *
     *          ((float(i)/float(n))**k3)
```

```
      end do

      do i=1,m
        f = f + delta * x(i) * x(i+2*m) *
     *          ((float(i)/float(n))**k4)
      end do
      return


cF70                          ARGLINB (m=5)
*
*         Initial point x0=[0.01   0.001, ....  ,0.01   0.001].
*
*
70    continue

      m=5
      f=0.d0

      do i=1,m
        u(i)=0.d0
        do j=1,n
          u(i) = u(i) + float(i)*float(j)*x(j)
        end do

        f = f + (u(i)-1.d0)**2
      end do
      return



cF71                        VARDIM (CUTE)
*
*                    Initial point x0=[1-1/n, 1-2/n,...,1-n/n.].
*                           Modified m=n/4
71    continue

       s = float(n)*float(n+1)/2.d0

       t1=0.d0
       do i=1,n
         t1 = t1 + float(i)*x(i)
       end do
       t1 = t1-s

       f = 0.d0
       do i=1,n
         f = f + (x(i)-1.d0)**2
       end do

       f = f + t1**2 + t1**4
       return



cF72                             DIAG-AUP1
*
*                    Initial point x0=[4., 4., ....4.].
*

72    continue
```

```
      f=0.d0
      do i=1,n
        f = f + 4.d0*(x(i)*x(i) - x(1))**2 + (x(i)**2-1.d0)**2
      end do
      return


cF73                              ENGVAL8
*
*                    Initial point x0=[2., 2., ....2.].
*
73     continue

       f = 0.d0
       do i=1,n-1
         f = f + (x(i)**2+x(i+1)**2)**2 - (7.d0-8.d0*x(i))
       end do
       return


cF74                              QUARTIC   (CUTE)
*
*                    Initial point x0=[2., 2., ....2.].
*
74     continue

       f = 0.d0
       do i=1,n
         f = f + (x(i)-1.d0)**4
       end do
       return


cF75                               LIARWHD   (CUTE)
*
*                    Initial point x0=[4., 4., ....4.].
*
75     continue

       f = 0.d0
       do i=1,n
         f = f + 4.d0*(x(i)**2-x(1))**2 + (x(i)-1.d0)**2
      end do

       return


cF76                              NONSCOMP (CUTE)
*
*                      Initial point x0=[3., 3., ..., 3.].
*
76    continue

       f = (x(1)-1.d0)**2
       do i=2,n
         f = f + 4.d0*(x(i)-x(i-1)**2)**2
       end do
       return

cF77                              Linear Perturbed
*
```

```
*                    Initial point x0=[2., ...,2.]
*
77     continue

       f = 0.d0
       do i=1,n
         f = f + float(i)*x(i)**2 + x(i)/100.d0
       end do
       return


cF78                              CUBE
*
*                     Initial point x0=[-1.2, 1, -1.2, 1.,....,-1.2, 1]
*
78     continue

       f = (x(1)-1.d0)**2
       do i=2,n
         f = f + 100.d0*(x(i)-x(i-1)**3)**2
       end do
       return


cF79                              HARKERP
*
*                      Initial point x0=[1,2,...,n]
*
79     continue

       s = 0.d0
       do i=1,n
         s = s + x(i)
       end do

       f = s*s
       do i=1,n
         f = f - (x(i) + 0.5d0*x(i)*x(i))
       end do
       return


cF80                              QUARTICM
*
*                      Initial point x0=[2,2,...,2]
*
80     continue
       f = 0.d0
       do i=1,n
         f = f + (x(i)-float(i))**4
       end do
       return

       end
* ----------------------------------------------------------Last line CG_VALUE
```

```
************************************************************
*                            Date created:  November 26, 2018
*
*                     GRADIENT OF FUNCTIONS
*                    ========================
*
*                                          Dr. Neculai Andrei
************************************************************
*
      subroutine cg_grad(g, x, n, nexp)
      real*8 x(n), g(n)

      real*8 t1,t2,t3,t4, c, d
      real*8 s, temp(1000000), temp1, tsum, sum
      real*8 u(1000000), v(1000000), t(1000000)
      real*8 u1, v1, c1, c2
      real*8 alpha, beta, gamma, delta

      integer k1, k2, k3, k4

*
      go to ( 1, 2, 3, 4, 5, 6, 7, 8, 9,10,
     *       11,12,13,14,15,16,17,18,19,20,
     *       21,22,23,24,25,26,27,28,29,30,
     *       31,32,33,34,35,36,37,38,39,40,
     *       41,42,43,44,45,46,47,48,49,50,
     *       51,52,53,54,55,56,57,58,59,60,
     *       61,62,63,64,65,66,67,68,69,70,
     *       71,72,73,74,75,76,77,78,79,80) nexp


cF1                                 FREUROTH (CUTE)
*                            Extended Freudenstein & Roth
*
*                       Initial Point: [0.5, -2, ...,0.5, -2].
*

1     continue
      j=1

      do i=1,n/2
        t1=-13.d0+x(2*i-1)+5.d0*x(2*i)*x(2*i)-x(2*i)**3-2.d0*x(2*i)
        t2=-29.d0+x(2*i-1)+x(2*i)**3+x(2*i)**2-14.d0*x(2*i)

        g(j)  =2.d0*(t1+t2)
        g(j+1)=2.d0*t1*(10.d0*x(2*i)-3.d0*x(2*i)*x(2*i)-2.d0) +
     *          2.d0*t2*(3.d0*x(2*i)*x(2*i)+2.d0*x(2*i)-14.d0)
        j=j+2
      end do
      return


cF2                       Extended White & Holst function
*
*           Initial point: [-1.2, 1, -1.2, 1, ........., -1.2, 1]

2     continue
      c=1.d0

      j=1
      do i=1,n/2
        g(j)   = -6.d0*c*x(2*i-1)*x(2*i-1)*(x(2*i)-x(2*i-1)**3) -
     *             2.d0*(1.d0-x(2*i-1))
```

```
        g(j+1) =  2.d0*c*(x(2*i)-x(2*i-1)**3)
        j = j + 2
      end do
      return


cF3                          TR-WHITEHOLST
*                     Tridiagonal. White-Holst (c=4)
*             Initial point x =[-1.2,  1, ..., -1.2, 1]
*
3     continue
      c = 4.d0

      g(1) = -6.d0*c*(x(2)-x(1)**3)*x(1)*x(1) - 2.d0*(1.d0-x(1))

      do i=2,n-1
        g(i) = 2.d0*c*(x(i)-x(i-1)**3) -
     *         6.d0*c*(x(i+1)-x(i)**3)*x(i)*x(i) -
     *         2.d0*(1.d0-x(i))
      end do

      g(n) = 2.d0*c*(x(n)-x(n-1)**3)
      return


cF4                   Extended Beale Function  BEALE (CUTE)
*
*                    Initial Point: [1, 0.8, ...., 1, 0.8]
*
4     continue
*
      j=1
      do i=1,n/2
        t1=1.5d0  -x(2*i-1)+x(2*i-1)*x(2*i)
        t2=2.25d0 -x(2*i-1)+x(2*i-1)*x(2*i)*x(2*i)
        t3=2.625d0-x(2*i-1)+x(2*i-1)*x(2*i)*x(2*i)*x(2*i)

        g(j)  =2.d0*t1*(-1.d0+x(2*i)) +
     *         2.d0*t2*(-1.d0+x(2*i)**2) +
     *         2.d0*t3*(-1.d0+x(2*i)**3)
        g(j+1)=2.d0*t1*x(2*i-1) +
     *         2.d0*t2*2.d0*x(2*i-1)*x(2*i) +
     *         2.d0*t3*3.d0*x(2*i-1)*x(2*i)*x(2*i)
        j=j+2
      end do
      return


cF5                               Extended Powell
*
*                       Initial Point: [3, -1, 0, 1, ......].
*
5     continue
      j=1
      do i=1,n/4
        t1= x(4*i-3) + 10.d0*x(4*i-2)
        t2= x(4*i-1) - x(4*i)
        t3= x(4*i-2) - 2.d0*x(4*i-1)
        t4= x(4*i-3) - x(4*i)

        g(j)  =   2.d0*t1 + 40.d0*t4**3
        g(j+1)=  20.d0*t1 +  4.d0*t3**3
        g(j+2)=  10.d0*t2 -  8.d0*t3**3
```

```
        g(j+3)= -10.d0*t2 - 40.d0*t4**3

        j=j+4
      end do
      return


cF6                           Extended Maratos Function
*
*                      Initial Point: [1.1, 0.1, ...,1.1, 0.1].
*
6     continue
      c = 1.d0

      j=1
      do i=1,n/2
        t1 = x(2*i-1)**2 + x(2*i)**2 - 1.d0

        g(j)   = 1.d0 + 4.d0 * c * t1 * x(2*i-1)
        g(j+1) =        4.d0 * c * t1 * x(2*i)
        j=j+2
      end do
      return
cF7                             Extended CLIFF (CUTE)
*
*                      Initial Point:  [0, -0.1, ......, 0, -0.1].
*
7     continue

      j=1
      do i=1,n/2
       temp1 = (x(2*i-1)-3.d0)/100.d0

       g(j)   = temp1/50.d0 - 1.d0 + 2.d0*dexp(2.d0*(x(2*i-1)-x(2*i)))
       g(j+1) = 1.d0 - 2.d0*dexp(2.d0*(x(2*i-1)-x(2*i)))
       j=j+2
      end do
      return


cF8                           Extended Wood Function
*                                   WOODS (CUTE)
*
*                        Initial Point: [-3,-1,-3,-1,......]
8     continue

      j=1
      do i=1,n/4
        g(j)   = 400.d0*(x(4*i-3)**2-x(4*i-2))*x(4*i-3)
     *             + 2.d0*(x(4*i-3)-1.d0)
        g(j+1) =-200.d0*(x(4*i-3)**2-x(4*i-2))
     *             + 20.2d0*(x(4*i-2)-1.d0)
     *             + 19.8d0*(x(4*i)-1.d0)
        g(j+2) = 360.d0*(x(4*i-1)**2-x(4*i))*x(4*i-1)
     *             - 2.d0*(1.d0-x(4*i-1))
        g(j+3) =-180.d0*(x(4*i-1)**2-x(4*i))
     *             + 20.2d0*(x(4*i)  -1.d0)
     *             + 19.8d0*(x(4*i-2)-1.d0)
        j=j+4
      end do
      return
```

```
cF9                           Extended Hiebert Function
*
*                              Initial Point: [0,0,...0].
9     continue

      c1 = 10.d0
      c2 = 500.d0

      j=1
      do i=1,n/2
        g(j)   =   2.d0*(x(2*i-1)-c1)
     *           + 2.d0*(x(2*i-1)*x(2*i)-c2)*x(2*i)
        g(j+1) =   2.d0*(x(2*i-1)*x(2*i)-c2)*x(2*i-1)
        j=j+2
      end do
      return


cF10                           SROSENBR (CUTE)
*                      Extended Rosenbrock function
*
* Initial point: [-1.2, 1, -1.2, 1, ........., -1.2, 1]

10    continue
      c=1000.d0

      j=1
      do i=1,n/2
        g(j)   = -4.d0*c*x(2*i-1)*(x(2*i)-x(2*i-1)**2) -
     *            2.d0*(1.d0-x(2*i-1))
        g(j+1) =  2.d0*c*(x(2*i)-x(2*i-1)**2)
        j = j + 2
      end do
      return


cF11                           GENROSNB (CUTE)
*                           Generalized Rosenbrock
*                     Initial Point: [-1.2, 1, ...  -1.2, 1]

11    continue

      g(1) = 2.d0*(x(1)-1.d0)-400.d0*x(1)*(x(2)-x(1)**2)
      do i=2,n-1
        g(i) = 200.d0*(x(i)-x(i-1)**2)-400.d0*x(i)*(x(i+1)-x(i)**2)
      end do

      g(n) = 200.d0*(x(n)-x(n-1)**2)
      return


cF12                               HIMMELBC (CUTE)
*                          Extended Himmelblau Function
*
*                            Initial Point: [1, 1, ....., 1]

12    continue

      j=1
      do i=1,n/2
        u1 = x(2*i-1)**2 + x(2*i)    - 11.d0
        v1 = x(2*i-1)    + x(2*i)**2 - 7.d0
```

```
        g(j)   = 4.d0*u1*x(2*i-1) + 2.d0*v1
        g(j+1) = 2.d0*u1          + 4.d0*v1*x(2*i)
        j=j+2
      end do
      return



cF13                          HIMMELBG (CUTE)
*
*                     Initial Point: [1.5,1.5,...,1.5]

13    continue

      j=1
      do i=1,n/2
        t1 = 2.d0*x(2*i-1)**2+3.d0*x(2*i)**2
        t2 = dexp(-x(2*i-1)-x(2*i))

        g(j)   =  4.d0*x(2*i-1)*t2 - t1*t2

        g(j+1) =  6.d0*x(2*i)*t2   - t1*t2

        j=j+2
      end do
      return


cF14                          HIMMELBH (CUTE)
*
*                     Initial Point: [1.5,1.5,...,1.5]

14    continue

      j=1
      do i=1,n/2

        g(j)   =  -3.d0 + 3.d0*x(2*i-1)**2

        g(j+1) =  -2.d0 + 2.d0*x(2*i)

        j=j+2
      end do
      return


cF15                            Extended Trigonometric ET1
*
*                          Initial Point: [0.2, 0.2, ....,0.2].

15    continue
      s= float(n)
      do i=1,n
        s = s - dcos(x(i))
      end do

      do i=1,n
        temp(i) = s + float(i)*(1.d0-dcos(x(i))) - dsin(x(i))
      end do

      s=0.d0
      do i=1,n
```

```
        s = s + temp(i)
      end do

      do i=1,n
        g(i) = 2.d0*s*dsin(x(i)) +
     +            2.d0*temp(i)*(float(i)*dsin(x(i))-dcos(x(i)))
      end do
      return


cF16                          Extended Trigonometric ET2
*
*                          Initial Point: [0.2, 0.2, ....,0.2].

16    continue
      s= float(n)
      do i=1,n
        s = s - dsin(x(i))
      end do

      do i=1,n
        temp(i) = s + float(i)*(1.d0-dsin(x(i))) - dsin(x(i))
      end do

      s=0.d0
      do i=1,n
        s = s + temp(i)
      end do

      do i=1,n
        g(i) = -2.d0*s*dcos(x(i)) +
     +          2.d0*temp(i)*(-float(i)*dcos(x(i))-dcos(x(i)))
      end do
      return


cF17                  Extended Block Diagonal BD1 Function
*
*                     Initial Point: [0.1, 0.1, ..., 0.1].
*
17    continue

      j=1
      do i=1,n/2
        t1 = x(2*i-1)**2 + x(2*i)**2 - 2.d0
        t2 = dexp(x(2*i-1)) - x(2*i)

        g(j)   = 4.d0*t1*x(2*i-1) + 2.d0*t2*dexp(x(2*i-1))
        g(j+1) = 4.d0*t1*x(2*i) - 2.d0*t2
        j=j+2
      end do
      return


cF18                   Extended Tridiagonal-1 Function
*
*                        Initial Point: [2,2,...,2]
18    continue

      do i=1,n/2
        u(i) = x(2*i-1) + x(2*i) - 3.d0
        v(i) = x(2*i-1) - x(2*i) + 1.d0
      end do
```

```
      j=1
      do i=1,n/2
        g(j)   = 2.d0*u(i) + 4.d0*v(i)**3
        g(j+1) = 2.d0*u(i) - 4.d0*v(i)**3
        j=j+2
      end do
      return


cF19                        Extended Three Exponential Terms
*
*                          Intial Point: [0.1,0.1,.....,0.1].
*
19    continue

      j=1
      do i=1,n/2
        t1= x(2*i-1) + 3.d0*x(2*i) - 0.1d0
        t2= x(2*i-1) - 3.d0*x(2*i) - 0.1d0
        t3=-x(2*i-1) - 0.1d0

        g(j)   = dexp(t1) + dexp(t2) - dexp(t3)
        g(j+1) = 3.d0*dexp(t1) - 3.d0*dexp(t2)
        j=j+2
      end do
      return


cF20                 Generalized Tridiagonal-1 Function
*
*                        Initial Point: [2,2,...,2]
20    continue

      do i=1,n-1
        u(i) = x(i) + x(i+1) - 3.d0
        v(i) = x(i) - x(i+1) + 1.d0
      end do

      g(1) = 2.d0*u(1) + 4.d0*v(1)**3

      do i=2,n-1
        g(i) = 2.d0*u(i-1) - 4.d0*v(i-1)**3 + 2.d0*u(i) + 4.d0*v(i)**3
      end do

      g(n) = 2.d0*u(n-1) - 4.d0*v(n-1)**3
      return



cF21                        Generalized Tridiagonal-2
*                                 Penta Diagonal
*                    Initial point: [-1, -1, ........., -1., -1]
21    continue
      u(1) = 5.d0*x(1)-3.d0*x(1)**2-x(1)**3-3.d0*x(2)+1.d0

      do i=2,n-1
        u(i)=5.d0*x(i)-3.d0*x(i)**2-x(i)**3-x(i-1)-3.d0*x(i+1)+1.d0
      end do

      u(n)=5.d0*x(n)-3.d0*x(n)**2-x(n)**3-x(n-1)+1.d0
*
```

```
      do i=1,n
        v(i) = 5.d0 -6.d0*x(i) -3.d0*x(i)**2
      end do
*
      g(1) = 2.d0*u(1)*v(1) - 2.d0*u(2)

      do i=2,n-1
        g(i) = -6.d0*u(i-1) + 2.d0*u(i)*v(i) - 2.d0*u(i+1)
      end do

      g(n) = -6.d0*u(n-1) + 2.d0*u(n)*v(n)
      return



cF22                        TR-DB1
*                  Tridiagonal Double Borded
*               Initial Point:  [-1, -1, ...., -1]
*
22    continue

       do i=1,n-1
         temp(i) = x(1) - 0.5d0*x(i)**2 - 0.5d0*x(i+1)**2
              end do
*--
       g(1) = 2.d0*(x(1)-1.d0) + 2.d0*temp(1)*(1.d0-x(1))

       do i=2,n-1
         g(1) = g(1) + 2.d0*temp(i)
       end do

       do i=2,n-1
         g(i) = -2.d0*x(i)*(temp(i-1) + temp(i))
       end do

       g(n) = -2.d0*x(n)*temp(n-1)
      return


cF23                       Broyden Pentadiagonal
*
*                  Initial point x0=[-1., -1., ..., -1.].
*
23    continue
*

      temp(1) = 3.d0*x(1) - 2.d0*x(1)*x(1)
      do i=2,n-1
        temp(i) = 3.d0*x(i)-2.d0*x(i)*x(i)-x(i-1)-2.d0*x(i+1)+1.d0
      end do

      temp(n) = 3.d0*x(n)-2.d0*x(n)*x(n)-x(n-1)+1.d0

      g(1) = 2.d0*temp(1)*(3.d0-4.d0*x(1)) - 2.d0*temp(2)

      g(2) = 2.d0*temp(2)*(3.d0-4.d0*x(2)) - 2.d0*temp(3)

      do i=3,n-1
        g(i) = -4.d0*temp(i-1)
     *         +2.d0*temp(i)*(3.d0-4.d0*x(i))
     *         -2.d0*temp(i+1)
      end do
```

```
      g(n) = -4.d0*temp(n-1) + 2.d0*temp(n)*(3.d0-4.d0*x(n))
      return


cF24                        Extended PSC1 Function
*
*                   Initial point: [3, 0.1, ..., 3, 0.1]

24    continue

      j=1
      do i=1,n/2
      g(j)   = 2.d0*(x(2*i-1)**2+x(2*i)**2+x(2*i-1)*x(2*i)) *
     +              (2.d0*x(2*i-1)+x(2*i)) +
     +              2.d0*(dsin(x(2*i-1)))*(dcos(x(2*i-1)))

      g(j+1) = 2.d0*(x(2*i-1)**2+x(2*i)**2+x(2*i-1)*x(2*i)) *
     +              (2.d0*x(2*i)+x(2*i-1)) -
     +              2.d0*(dcos(x(2*i)))*(dsin(x(2*i)))
      j=j+2
      end do
      return


cF25                         Perturbed Quadratic function PQ1
*
*                        Initial Point:  [1, 1, ......,1].
*
25    continue

      temp1 = 0.d0
      do i=1,n
        temp1 = temp1 + x(i)
      end do

      do i=1,n
        g(i) = float(i) * 2.d0 * x(i) + temp1/50.d0
      end do
      return


cF26                        Perturbed Quadratic function PQ2
*
*                     Initial Point:  [0.5, 0.5, ......, 0.5].
*
26    continue

      temp1 = 0.d0
      do i=1,n
        temp1 = temp1 + float(i)*x(i)
      end do

      do i=1,n
        g(i) = float(i)*2.d0*x(i) + 2.d0*temp1*float(i)
      end do
      return


cF27                    Almost Perturbed Quadratic
*
*                  Initial point x0=[0.5, 0.5, ...,0.5].
*
27    continue
```

```
*
      g(1) = 2.d0*x(1) + (x(1)+x(n))/50.d0

      do i=2,n-1
        g(i) = 2.d0*float(i)*x(i)
      end do

      g(n) = 2.d0*float(n)*x(n) + (x(1)+x(n))/50.d0
      return


cF28                    Almost Perturbed Quartic
*
*                  Initial point x0=[0.5, 0.5, ...,0.5].
*
28    continue
*
      g(1) = 4.d0*x(1)**3 + (x(1)+x(n))/50.d0

      do i=2,n-1
        g(i) = 4.d0*float(i)*x(i)**3
      end do

      g(n) = 4.d0*float(n)*x(n)**3 + (x(1)+x(n))/50.d0
      return


cF29                    Extended Penalty Function  U52 (MatrixRom)
*
*                           Intial Point: [1,2,3,.....,n].
*
29    continue
      temp1=0.d0
      do i=1,n
        temp1 = temp1 + x(i)**2
      end do

      do i=1,n-1
        g(i) = 2.d0*(x(i)-1.d0) + 4.d0*x(i)*(temp1-0.25d0)
      end do

      g(n) = 4.d0*x(n)*(temp1-0.25d0)
      return


cF30                              TR-Summ of quadratics Function
*
*                                Initial Point: [1, 1, ....., 1]
*
30    continue

        c = 100000.d0

      g(1) = 2.d0*x(1) + 4.d0*c*(x(2)+x(1)*x(1))*x(1)

      do i=2,n-1
        g(i) = 2.d0*c*(x(i)+x(i-1)**2) + 2.d0*x(i) +
     *         4.d0*c*(x(i+1)+x(i)*x(i))*x(i)
      end do

      g(n) = 2.d0*c*(x(n)+x(n-1)**2)
      return
```

```
cF31                    Quadratic Diagonal Perturbed Function
*
*                      Initial Point:  [0.5, 0.5, ......, 0.5].
*
31    continue

      temp1 = 0.d0
      do i=1,n
        temp1 = temp1 + x(i)
      end do

      do i=1,n
        g(i) = float(i) * x(i) / 50.d0 + 2.d0*temp1
      end do
      return


cF32                          FH1 (m=50)
*        Full Hessian FH1 (Summ of Quadratics, Quadratic inside)
*
32    continue
      m=50

      do i=1,m
        u(i)=0.d0
        do j=1,n
          u(i) = u(i) + float(i)*float(j)*x(j)*x(j)
        end do
      end do
*--
      do j=1,n
        g(j) = 0.d0
        do i=1,m
          g(j) = g(j) + 4.d0*(u(i)-1.d0)*float(i)*float(j)*x(j)
        end do
      end do
      return


cF33                         FH2
*      Full Hessian FH2 (Quadratic, perturbed with sin/cos)
*           Initial Point:  [1, 1, ......, 1].
*
33    continue
       s = 0.d0
       do i=1,n
         s = s + x(i)
      end do

       do i=1,n
         g(i) = 2.d0*s + float(i)*(dcos(x(i)) - dsin(x(i)))/1000.d0
       end do
       return


cF34                         FH3
*      Full Hessian FH3 (Quartic, perturbed with sin/cos)
*           Initial Point:  [1, 1, ......, 1].
*
34    continue
       s = 0.d0
       do i=1,n
```

```
         s = s + x(i)**2
      end do

       do i=1,n
         g(i) = 4.d0*s*x(i) + float(i)*(dcos(x(i)) - dsin(x(i)))/1000.d0
       end do
       return


cF35                        Diagonal Full Borded
*
*                Initial point: [0.1, 0.1, ........., 0.1]

35    continue

      do i=1,n-2
        temp(i) = sin(x(i+1)-x(n)) - x(1)**2 - x(i+1)**2
      end do
*--
      g(1) = 4.d0*(x(1)-1.d0)**3 - 4.d0*x(1)*(x(n)**2-x(1)**2)
      do i=1,n-2
        g(1) = g(1) - 4.d0*temp(i)*x(1)
      end do

      do i=2,n-1
        g(i) = 2.d0*temp(i-1)*(cos(x(i)-x(n))-2.d0*x(i))
      end do

      g(n) = 4.d0*x(n)*(x(n)**2-x(1)**2)
      do i=1,n-2
        g(n) = g(n) - 2.d0*temp(i)*cos(x(i+1)-x(n))
      end do
      return


cF36                          D-DBAUP3
*                 Diagonal Double Borded Arrow Up
*                Initial point: x0=[4, 0, ....,4,0]
*
36    continue

      g(1) = 2.d0*(x(1)-1.d0) + 8.d0*(x(1)*x(1)-x(1))*(2.d0*x(1)-1.d0)
      do i=2,n
        g(1) = g(1) - 8.d0*(x(i)*x(i)-x(1))
      end do

      do i=2,n
        g(i) = 16.d0*x(i)*(x(i)*x(i)-x(1)) + 2.d0*(x(i)-1.d0)
      end do
      return


cF37                    QP1 Extended Quadratic Penalty
*
*                      Initial Point:  [1, 1, ......,1].
*
37    continue

      t1=0.d0
      do i=1,n
        t1 = t1 + x(i)*x(i)
      end do
      t1 = t1 - 0.5d0
```

```
      do i=1,n-1
        g(i) = 4.d0*(x(i)*x(i)-2.d0)*x(i) + 4.d0*t1*x(i)
      end do

      g(n) = 4.d0*t1*x(n)
      return


cF38                 QP2 Extended Quadratic Penalty Function
*
*                     Initial Point:  [1, 1, ......,1].
*
38    continue

      t1=0.d0
      do i=1,n
        t1 = t1 + x(i)*x(i)
      end do
      t1 = t1 - 100.d0

      do i=1,n-1
        g(i) = 2.d0*(x(i)*x(i)-dsin(x(i)))*(2.d0*x(i)-dcos(x(i)))
     *       + 4.d0*t1*x(i)
      end do

      g(n) = 4.d0*t1*x(n)
      return


cF39                   QP3 Extended Quadratic Penalty
*
*                     Initial Point: [1., 1., ....,1.].

39    continue

      t1=0.d0
      do i=1,n
        t1 = t1 + x(i)*x(i)
      end do
      t1 = t1 - 0.25d0

      do i=1,n-1
        g(i) = -4.d0*(x(i)*x(i)-1.d0)*x(i) + 4.d0*t1*x(i)
      end do

      g(n) = 4.d0*t1*x(n)
      return


cF40                            STAIRCASE S1
*
*                      Initial point x0=[1,1,...,1].
*
40    continue

         g(1) = 2.d0*(x(1)+x(2)-1.d0)
      do i=2,n-1
        g(i) = 2.d0*(x(i-1)+x(i)-float(i-1)) +
     *         2.d0*(x(i)+x(i+1)-float(i))
      end do
      g(n) = 2.d0*(x(n-1)+x(n)-float(n-1))
      return
```

```
cF41                            STAIRCASE S2
*
*                       Initial point x0=[1,1,...,1].
*
41    continue

       g(1)=2.d0*(x(1)+x(2)-2.d0)
       do i=2,n-1
         g(i) = 2.d0*(x(i-1)+x(i)-float(i)) +
     *         2.d0*(x(i)+x(i+1)-float(i+1))
       end do
       g(n) = 2.d0*(x(n-1)+x(n)-float(n))
       return


cF42                            STAIRCASE S3
*
*                       Initial point x0=[2,2,...,2].
*
42    continue

       g(1) = 2.d0*(x(1)+x(2)+2.d0)

       do i=2,n-1
         g(i) = 2.d0*(x(i-1)+x(i)+float(i))+
     *         2.d0*(x(i)+x(i+1)+float(i+1))
       end do

       g(n) = 2.d0*(x(n-1)+x(n)+float(n))
       return


cF43                          NONDQUAR
*              Tridiagonal Double Borded Arrow-Down
*
43    continue

      g(1) = 2.d0*(x(1)-x(2))+4.d0*(x(1)+x(2)+x(n))**3
      g(2) =-2.d0*(x(1)-x(2))+4.d0*(x(1)+x(2)+x(n))**3 +
     *                         4.d0*(x(2)+x(3)+x(n))**3

      do i=3,n-2
        g(i) = 4.d0*(x(i-1)+x(i)+x(n))**3 +
     *         4.d0*(x(i)+x(i+1)+x(n))**3
      end do

      g(n-1) = 4.d0*(x(n-2)+x(n-1)+x(n))**3 +
     *         2.d0*(x(n-1)+x(n))

      g(n) = 2.d0*(x(n-1)+x(n))
      do i=1,n-2
        g(n) = g(n) + 4.d0*(x(i)+x(i+1)+x(n))**3
      end do
      return


cF44                            TRIDIA  (CUTE)
*
*                      Initial point x0=[1,1,...,1].
*
44    continue
```

```
*
      alpha=5.d0
      beta =1.d0
      gamma=1.d0
      delta=1.d0

      g(1) = 2.d0*gamma*(delta*x(1)-1.d0)*delta -
     *       4.d0*(alpha*x(2)-beta*x(1))*beta

      do i=2,n-1
        g(i) = 2.d0*float(i)*(alpha*x(i)-beta*x(i-1))*alpha -
     *         2.d0*float(i+1)*(alpha*x(i+1)-beta*x(i))*beta
      end do

      g(n) = 2.d0*float(n)*(alpha*x(n)-beta*x(n-1))*alpha
      return


cF45                        ARWHEAD  (CUTE)
*
*                     Initial point x0=[1,1,...,1].
*
45    continue

      do i=1,n-1
        g(i) = -4.d0 + 4.d0*x(i)*(x(i)**2+x(n)**2)
      end do

      g(n) = 0.d0
      do i=1,n-1
        g(n) = g(n) + 4.d0*x(n)*(x(i)**2+x(n)**2)
      end do
      return


cF46
*                       NONDIA (Shanno-78)  (CUTE)
*
*                     Initial point x0=[-1,-1,...,-1].
*
*
46    continue

      c=100.d0

      g(1)=2.d0*(x(1)-1.d0) + 2.d0*c*(x(1)-x(1)**2)*(1.d0-2.d0*x(1))
      do i=2,n
        g(1) = g(1) + 2.d0*c*(x(1)-x(i)**2)
      end do

      do i=2,n
        g(i) = -4.d0*c*x(i)*(x(1)-x(i)**2)
      end do
      return


cF47                            BDQRTIC (CUTE)
*
*                     Initial point x0=[1.,1.,...,1.].
*
*
47    continue
*
```

```
      n4=n-4

      do i=1,n4
        temp(i) = x(i)**2 + 2.d0*x(i+1)**2 + 3.d0*x(i+2)**2
     *                    + 4.d0*x(i+3)**2 + 5.d0*x(n)**2
      end do

      g(1) = -8.d0*(-4.d0*x(1)+3.d0) +
     *        (4.d0*temp(1))*x(1)
      g(2) = -8.d0*(-4.d0*x(2)+3.d0) +
     *        (8.d0*temp(1)+  4.d0*temp(2))*x(2)
      g(3) = -8.d0*(-4.d0*x(3)+3.d0) +
     *        (12.d0*temp(1)+ 8.d0*temp(2)+ 4.d0*temp(3))*x(3)
      g(4) = -8.d0*(-4.d0*x(4)+3.d0) +
     *        (16.d0*temp(1)+12.d0*temp(2) +8.d0*temp(3)+
     *          4.d0*temp(4))*x(4)

      do i=5,n4
        g(i) = -8.d0*(-4.d0*x(i)+3.d0) +
     *         (16.d0*temp(i-3)+12.d0*temp(i-2)+
     *           8.d0*temp(i-1)+4.d0*temp(i))*x(i)
      end do

      g(n4+1) =(16.d0*temp(n4-2)+12.d0*temp(n4-1)+8.d0*temp(n4))*x(n4+1)
      g(n4+2) =(16.d0*temp(n4-1)+12.d0*temp(n4))*x(n4+2)
      g(n4+3) =(16.d0*temp(n4))*x(n4+3)

      tsum=0.d0
      do i=1,n4
        tsum = tsum + temp(i)
      end do
      g(n) = 20.d0*tsum*x(n)
      return


cF48                            DQDRTIC  (CUTE)
*
*                     Initial point x0=[3,3,3...,3].
*
*
48    continue

      c=1000.d0
      d=1000.d0

      g(1) = 2.d0*x(1)
      g(2) = 2.d0*c*x(2) + 2.d0*x(2)

      do i=3,n-2
        g(i) = 2.d0*(1.d0+d+c)*x(i)
      end do

      g(n-1) = 2.d0*(c+d)*x(n-1)
      g(n) = 2.d0*d*x(n)
      return


cF49                            EG2 (CUTE)
*
*                     Initial point x0=[1,1,1...,1].
*
*
49    continue
```

```
      g(1)=(1.d0+2.d0*x(1))*dcos(x(1)+x(1)*x(1)-1.d0)
      do i=2,n-1
        g(1) = g(1) + dcos(x(1)+x(i)*x(i)-1.d0)
      end do

      do i=2,n-1
        g(i) = 2.d0*x(i)*dcos(x(1)+x(i)*x(i)-1.d0)
      end do

      g(n) = x(n)*dcos(x(n)*x(n))
      return


cF50                          EG3
*
*                  Initial point x0=[1,1,1...,1].
*
*
50    continue

      g(1)=-(1.d0+2.d0*x(1))*dsin(x(1)+x(1)*x(1)-1.d0)
      do i=2,n-1
        g(1) = g(1) - dsin(x(1)+x(i)*x(i)-1.d0)
      end do

      do i=2,n-1
        g(i) = -2.d0*x(i)*dsin(x(1)+x(i)*x(i)-1.d0)
      end do

      g(n) = -x(n)*dsin(x(n)*x(n))
      return


cF51                        EDENSCH Function  (CUTE)
*
*                      Initial Point: [0., 0., ..., 0.].
51    continue

      g(1) = 4.d0*(x(1)-2.d0)**3 + 2.d0*x(2)*(x(1)*x(2)-2.d0*x(2))
*
      do i=2,n-1
        g(i) = 2.d0*(x(i-1)*x(i)-2.d0*x(i))*(x(i-1)-2.d0) +
     *         2.d0*(x(i)+1.d0) +
     *         4.d0*(x(i)-2.d0)**3 +
     *         2.d0*x(i+1)*(x(i)*x(i+1)-2.d0*x(i+1))
      end do

      g(n) = 2.d0*(x(n-1)*x(n)-2.d0*x(n))*(x(n-1)-2.d0) +
     *       2.d0*(x(n)+1.d0)
      return


cF52                        FLETCHCR (CUTE)
*
*                  Initial Point: [0.5,0.5,...0.5]
52    continue

      g(1) = 200.d0*(x(2)-x(1)+1.d0-x(1)*x(1))*(-1.d0-2.d0*x(1))

      do i=2,n-1
        g(i) = 200.d0*(x(i)-x(i-1)+1.d0-x(i-1)*x(i-1))+
     *         200.d0*(x(i+1)-x(i)+1.d0-x(i)*x(i))*(-1.d0-2.d0*x(i))
```

```
      end do

      g(n) = 200.d0*(x(n)-x(n-1)+1.d0-x(n-1)**2)
      return


cF53                          ENGVAL1 (CUTE)
*
*                     Initial point x0=[2.,2.,2...,2.].
*
53    continue

      do i=1,n-1
        t(i) = x(i)*x(i) + x(i+1)*x(i+1)
      end do

      g(1) = 4.d0*x(1)*t(1) - 4.d0

      do i=2,n-1
        g(i) = 4.d0*x(i)*t(i-1) + 4.d0*x(i)*t(i) - 4.d0
      end do

      g(n) = 4.d0*x(n)*t(n-1)
      return


cF54                          DENSCHNA (CUTE)
*
*                       Initial point: [8, 8,...,8]
*
54    continue

      j=1
      do i=1,n/2
        g(j) = 4.d0*x(2*i-1)**3 + 2.d0*(x(2*i-1)+x(2*i))
        g(j+1) = 2.d0*(x(2*i-1)+x(2*i)) +
     *           2.d0*(dexp(x(2*i)))*(-1.d0+dexp(x(2*i)))
        j=j+2
      end do
      return


cF55                    DENSCHNB  (CUTE)
*
*                Initial point: [0.1, 0.1,...,0.1]

55    continue

      j=1
      do i=1,n/2
        g(j) = 2.d0*(x(2*i-1)-2.d0) + 2.d0*(x(2*i-1)-2.d0)*x(2*i)*x(2*i)
        g(j+1) = ((x(2*i-1)-2.d0)**2)*2.d0*x(2*i) + 2.d0*(x(2*i)+1.d0)
        j=j+2
      end do
      return


cF56                    DENSCHNC  (CUTE)
*
*                Initial point: [8, 8,...,8]

56    continue
```

```
      j=1
      do i=1,n/2
       g(j) = 4.d0*x(2*i-1)*(-2.d0+x(2*i-1)**2+x(2*i)**2)+
     *  2.d0*(dexp(x(2*i-1)-1.d0))*(-2.d0+dexp(x(2*i-1)-1.d0)+x(2*i)**3)
       g(j+1) = 4.d0*x(2*i)*(-2.d0+x(2*i-1)**2+x(2*i)**2)+
     *  6.d0*(x(2*i)**2)*(-2.d0+dexp(x(2*i-1)-1.d0)+x(2*i)**3)
        j=j+2
      end do
      return


cF57                  DENSCHNF  (CUTE)
*
*               Initial point: [2,0,2,0,...,2,0]

57    continue

      j=1
      do i=1,n/2
      g(j)=2.d0*(2.d0*(x(2*i-1)+x(2*i))**2+(x(2*i-1)-x(2*i))**2-8.d0)*
     *  (4.d0*(x(2*i-1)+x(2*i))+2.d0*(x(2*i-1)-x(2*i))) +
     *  2.d0*(5.d0*x(2*i-1)**2+(x(2*i)-3.d0)**2-9.d0)*10.d0*x(2*i-1)
      g(j+1)=2.d0*(2.d0*(x(2*i-1)+x(2*i))**2+(x(2*i-1)-x(2*i))**2-8.d0)*
     *  (4.d0*(x(2*i-1)+x(2*i))-2.d0*(x(2*i-1)-x(2*i))) +
     *  2.d0*(5.d0*x(2*i-1)**2+(x(2*i)-3.d0)**2-9.d0)*2.d0*(x(2*i)-3.d0)
        j=j+2
      end do
      return


cF58                     SINQUAD (CUTE)
*
*             Initial Point: [0.1, 0.1, ..., 0.1]

58    continue

      do i=1,n-2
        t(i) = dsin(x(i+1)-x(n)) - x(1)**2 + x(i+1)**2
      end do
c
      g(1) = 4.d0*(x(1)-1.d0)**3 - 4.d0*x(1)*(x(n)**2-x(1)**2)
      do i=1,n-2
        g(1) = g(1) - 4.d0*t(i)*x(1)
      end do

      do i=2,n-1
        g(i) = 2.d0*t(i-1)*(dcos(x(i)-x(n))+2.d0*x(i))
      end do

      g(n) = 4.d0*x(n)*(x(n)**2-x(1)**2)
      do i=1,n-2
        g(n) = g(n) - 2.d0*t(i)*dcos(x(i+1)-x(n))
      end do
      return


cF59                       DIXON3DQ  (CUTE)
*
*                  Initial Point x0=[-1, -1,..., -1]
*
59    continue

      g(1) = 2.d0*(x(1)-2.d0) + 2.d0*(x(1)-x(2))
```

```
      do i=2,n-1
        g(i) = -2.d0*(x(i-1)-x(i)) + 2.d0*(x(i)-x(i+1))
      end do

      g(n) = -2.d0*(x(n-1)-x(n)) + 2.d0*(x(n)-1.d0)
      return


cF60                      BIGGSB1 (CUTE)
*
*               Initial Point: [0., 0., ....,0.]

60    continue

      g(1) = 4.d0*x(1) - 2.d0*x(2) - 2.d0

      do i=2,n-1
        g(i) = 4.d0*x(i) - 2.d0*x(i-1) - 2.d0*x(i+1)
      end do

      g(n) = 4.d0*x(n) - 2.d0*x(n-1) - 2.d0
      return


cF61                          PRODsin (m=n-1)
*
*                Initial point x0=[5. 5, 5, 5,...,5].
*
61    continue

       m = n-1
       t1=0.d0
       t2=0.d0

       do i=1,m
         t1 = t1 + x(i)*x(i)
       end do

       do i=1,n
         t2 = t2 + dsin(x(i))
       end do

       do i=1,m
         g(i) = 2.d0*x(i)*t2 + t1*dcos(x(i))
       end do

       do i=m+1,n
         g(i) = t1*dcos(x(i))
       end do
       return


cF62                          PROD1 (m=n)
*
*              Initial point x0=[1. 1, 1, 1,...,1].
*
62    continue

       m = n
       t1=0.d0
       t2=0.d0
```

```
      do i=1,m
        t1 = t1 + x(i)
      end do

      do i=1,n
        t2 = t2 + x(i)
      end do

      do i=1,m
        g(i) = t1+t2
      end do

      do i=m+1,n
        g(i) = t1
      end do
      return


cF63                          PRODcos (m=n-1)
*
*                Initial point x0=[5. 5, 5, 5,...,5].
*
63    continue

c     m = n-1
      m=n/2
      t1=0.d0
      t2=0.d0

      do i=1,m
        t1 = t1 + x(i)*x(i)
      end do

      do i=1,n
        t2 = t2 + dcos(x(i))
      end do

      do i=1,m
        g(i) = 2.d0*x(i)*t2 - t1*dsin(x(i))
      end do

      do i=m+1,n
        g(i) = -t1*dsin(x(i))
      end do
      return


cF64                          PROD2 (m=1)
*
*               Initial point x0=[15. 15, 15, 15,...,15].
*
64    continue

      m = 1
      t1=0.d0
      t2=0.d0

      do i=1,m
        t1 = t1 + x(i)**4
      end do

      do i=1,n
        t2 = t2 + float(i)*x(i)
```

```
      end do

      do i=1,m
        g(i) = 4.d0*t2*x(i)**3 + float(i)*t1
      end do

      do i=m+1,n
        g(i) = float(i)*t1
      end do
      return


cF65                        DIXMAANA (CUTE)
*
*                    Initial point x0=[2.,2.,2...,2.].
*                           Modified m=n/4
65    continue
*
        alpha = 1.d0
        beta  = 0.d0
        gamma = 0.125d0
        delta = 0.125d0

        k1 = 0
        k2 = 0
        k3 = 0
        k4 = 0

        m = n/4

      do i=1,n
        g(i) =0.d0
      end do
c1

      do i=1,n
        g(i) = g(i) + 2.d0*alpha*x(i)*((float(i)/float(n))**k1)
      end do
c2

      g(1) = g(1) + 2.d0*beta*x(1)*((x(2)+x(2)*x(2))**2)*
     *               ((float(1)/float(n))**k2)
      do i=2,n-1
        g(i) = g(i) + 2.d0*beta*(x(i-1)**2)*(x(i)+x(i)**2)*
     *                     (1.d0+2.d0*x(i))*((float(i-1)/float(n))**k2)+
     *                2.d0*beta*x(i)*((x(i+1)+x(i+1)**2)**2)*
     *                     ((float(i)/float(n))**k2)
      end do
      g(n) = g(n) + 2.d0*beta*(x(n-1)**2)*(x(n)+x(n)**2)*
     *                     (1.d0+2.d0*x(n))
c3

      do i=1,2*m
        g(i) = g(i) + 2.d0*gamma*x(i)*(x(i+m)**4)*
     *                ((float(i)/float(n))**k3)
        g(i+m) = g(i+m) + gamma*(x(i)**2)*4.d0*(x(i+m)**3)*
     *                ((float(i)/float(n))**k3)
      end do
c4

      do i=1,m
        g(i) = g(i) + delta*x(i+2*m)*((float(i)/float(n))**k4)
        g(i+2*m) = g(i+2*m) + delta*x(i)*((float(i)/float(n))**k4)
```

```
      end do
      return


cF66                        DIXMAANB (CUTE)
*
*                    Initial point x0=[2.,2.,2...,2.].
*                           Modified m=n/4
66    continue
*
        alpha = 1.d0
        beta  = 0.0625d0
        gamma = 0.0625d0
        delta = 0.0625d0

        k1 = 0
        k2 = 0
        k3 = 0
        k4 = 1

        m = n/4

      do i=1,n
        g(i) =0.d0
      end do
c1

      do i=1,n
        g(i) = g(i) + 2.d0*alpha*x(i)*((float(i)/float(n))**k1)
      end do
c2

       g(1) = g(1) + 2.d0*beta*x(1)*((x(2)+x(2)*x(2))**2)*
     *               ((float(1)/float(n))**k2)
      do i=2,n-1
        g(i) = g(i) + 2.d0*beta*(x(i-1)**2)*(x(i)+x(i)**2)*
     *                      (1.d0+2.d0*x(i))*((float(i-1)/float(n))**k2)+
     *                 2.d0*beta*x(i)*((x(i+1)+x(i+1)**2)**2)*
     *                      ((float(i)/float(n))**k2)
      end do
       g(n) = g(n) + 2.d0*beta*(x(n-1)**2)*(x(n)+x(n)**2)*
     *                      (1.d0+2.d0*x(n))
c3

      do i=1,2*m
        g(i) = g(i) + 2.d0*gamma*x(i)*(x(i+m)**4)*
     *                 ((float(i)/float(n))**k3)
        g(i+m) = g(i+m) + gamma*(x(i)**2)*4.d0*(x(i+m)**3)*
     *                 ((float(i)/float(n))**k3)
      end do
c4

      do i=1,m
        g(i) = g(i) + delta*x(i+2*m)*((float(i)/float(n))**k4)
        g(i+2*m) = g(i+2*m) + delta*x(i)*((float(i)/float(n))**k4)
      end do
      return


cF67                        DIXMAANC (CUTE)
*
*                    Initial point x0=[2.,2.,2...,2.].
*                           Modified m=n/4
```

```
67    continue
*
        alpha = 1.d0
        beta  = 0.125d0
        gamma = 0.125d0
        delta = 0.125d0

        k1 = 0
        k2 = 0
        k3 = 0
        k4 = 0

        m = n/4

      do i=1,n
        g(i) =0.d0
      end do
c1

      do i=1,n
        g(i) = g(i) + 2.d0*alpha*x(i)*((float(i)/float(n))**k1)
      end do
c2

       g(1) = g(1) + 2.d0*beta*x(1)*((x(2)+x(2)*x(2))**2)*
     *               ((float(1)/float(n))**k2)
      do i=2,n-1
        g(i) = g(i) + 2.d0*beta*(x(i-1)**2)*(x(i)+x(i)**2)*
     *                      (1.d0+2.d0*x(i))*((float(i-1)/float(n))**k2)+
     *                2.d0*beta*x(i)*((x(i+1)+x(i+1)**2)**2)*
     *                      ((float(i)/float(n))**k2)
      end do
       g(n) = g(n) + 2.d0*beta*(x(n-1)**2)*(x(n)+x(n)**2)*
     *                      (1.d0+2.d0*x(n))
c3

      do i=1,2*m
        g(i) = g(i) + 2.d0*gamma*x(i)*(x(i+m)**4)*
     *                ((float(i)/float(n))**k3)
        g(i+m) = g(i+m) + gamma*(x(i)**2)*4.d0*(x(i+m)**3)*
     *                ((float(i)/float(n))**k3)
      end do
c4

      do i=1,m
        g(i) = g(i) + delta*x(i+2*m)*((float(i)/float(n))**k4)
        g(i+2*m) = g(i+2*m) + delta*x(i)*((float(i)/float(n))**k4)
      end do
      return


cF68                        DIXMAAND (CUTE)
*
*                    Initial point x0=[2.,2.,2...,2.].
*                           Modified m=n/4
68    continue
*
        alpha = 1.d0
        beta  = 0.26d0
        gamma = 0.26d0
        delta = 0.26d0

        k1 = 0
```

```
        k2 = 0
        k3 = 0
        k4 = 0

        m = n/4

      do i=1,n
        g(i) =0.d0
      end do
c1

      do i=1,n
        g(i) = g(i) + 2.d0*alpha*x(i)*((float(i)/float(n))**k1)
      end do
c2
        g(1) = g(1) + 2.d0*beta*x(1)*((x(2)+x(2)*x(2))**2)*
     *              ((float(1)/float(n))**k2)
      do i=2,n-1
        g(i) = g(i) + 2.d0*beta*(x(i-1)**2)*(x(i)+x(i)**2)*
     *                    (1.d0+2.d0*x(i))*((float(i-1)/float(n))**k2)+
     *                2.d0*beta*x(i)*((x(i+1)+x(i+1)**2)**2)*
     *                    ((float(i)/float(n))**k2)
      end do
       g(n) = g(n) + 2.d0*beta*(x(n-1)**2)*(x(n)+x(n)**2)*
     *                    (1.d0+2.d0*x(n))
c3

      do i=1,2*m
        g(i) = g(i) + 2.d0*gamma*x(i)*(x(i+m)**4)*
     *                ((float(i)/float(n))**k3)
        g(i+m) = g(i+m) + gamma*(x(i)**2)*4.d0*(x(i+m)**3)*
     *                ((float(i)/float(n))**k3)
      end do
c4

      do i=1,m
        g(i) = g(i) + delta*x(i+2*m)*((float(i)/float(n))**k4)
        g(i+2*m) = g(i+2*m) + delta*x(i)*((float(i)/float(n))**k4)
      end do
      return


cF69                        DIXMAANL (CUTE)
*
*                    Initial point x0=[2.,2.,2...,2.].
*                           Modified m=n/4
69    continue
*
        alpha = 1.d0
        beta  = 0.26d0
        gamma = 0.26d0
        delta = 0.26d0

        k1 = 2
        k2 = 0
        k3 = 0
        k4 = 2

        m = n/4

      do i=1,n
        g(i) =0.d0
      end do
```

```
c1

      do i=1,n
        g(i) = g(i) + 2.d0*alpha*x(i)*((float(i)/float(n))**k1)
      end do
c2
         g(1) = g(1) + 2.d0*beta*x(1)*((x(2)+x(2)*x(2))**2)*
     *               ((float(1)/float(n))**k2)
      do i=2,n-1
        g(i) = g(i) + 2.d0*beta*(x(i-1)**2)*(x(i)+x(i)**2)*
     *                     (1.d0+2.d0*x(i))*((float(i-1)/float(n))**k2)+
     *                2.d0*beta*x(i)*((x(i+1)+x(i+1)**2)**2)*
     *                     ((float(i)/float(n))**k2)
      end do
       g(n) = g(n) + 2.d0*beta*(x(n-1)**2)*(x(n)+x(n)**2)*
     *                     (1.d0+2.d0*x(n))
c3

      do i=1,2*m
        g(i) = g(i) + 2.d0*gamma*x(i)*(x(i+m)**4)*
     *                ((float(i)/float(n))**k3)
        g(i+m) = g(i+m) + gamma*(x(i)**2)*4.d0*(x(i+m)**3)*
     *                ((float(i)/float(n))**k3)
      end do
c4
      do i=1,m
        g(i) = g(i) + delta*x(i+2*m)*((float(i)/float(n))**k4)
        g(i+2*m) = g(i+2*m) + delta*x(i)*((float(i)/float(n))**k4)
      end do
      return


cF70                          ARGLINB (m=5)
*
*         Initial point x0=[0.01   0.001, ....  ,0.01   0.001].
*
*
70    continue

      m=5

      do i=1,m
        u(i)=0.d0
        do j=1,n
          u(i) = u(i) + float(i)*float(j)*x(j)
        end do
      end do
c
      do j=1,n
        g(j) = 0.d0
        do i=1,m
          g(j) = g(j) + 2.d0*(u(i)-1.d0)*float(i)*float(j)
        end do
      end do
      return


cF71                         VARDIM (CUTE)
*
*                    Initial point x0=[1-1/n, 1-2/n,...,1-n/n.].
*                           Modified m=n/4
71    continue
```

```
      s = float(n)*float(n+1)/2.d0

      t1=0.d0
      do i=1,n
        t1 = t1 + float(i)*x(i)
      end do
      t1 = t1-s

      do i=1,n
        g(i) = 2.d0*(x(i)-1.d0) + 2.d0*t1*float(i) + 4.d0*float(i)*t1**3
      end do
      return


cF72                              DIAG-AUP1
*
*                    Initial point x0=[4., 4., ....4.].
*

72    continue

      g(1) = 4.d0*(x(1)**2-1.d0)*x(1) +
     *       8.d0*(x(1)*x(1)-x(1))*(2.d0*x(1)-1.d0)
      do i=2,n
        g(1) = g(1) - 8.d0*(x(i)*x(i)-x(1))
      end do

      do i=2,n
        g(i) = 16.d0*x(i)*(x(i)*x(i)-x(1)) + 4.d0*(x(i)**2-1.d0)*x(i)
      end do
      return



cF73                              ENGVAL8
*
*                    Initial point x0=[2., 2., ....2.].
*
73    continue

      g(1) = 4.d0*(x(1)**2+x(2)**2)*x(1) + 8.d0

      do i=2,n-1
        g(i)=4.d0*(x(i-1)**2+x(i)**2)*x(i) +
     *       4.d0*(x(i)**2+x(i+1)**2)*x(i) + 8.d0
      end do

      g(n) = 4.d0*(x(n-1)**2+x(n)**2)*x(n)
      return


cF74                              QUARTIC   (CUTE)
*
*                    Initial point x0=[2., 2., ....2.].
*
74    continue

      do i=1,n
        g(i) = 4.d0*((x(i)-1.d0)**3)
      end do
      return
```

```
cF75                                LIARWHD   (CUTE)
*
*                     Initial point x0=[4., 4., ....4.].
*
75     continue

       g(1) = 8.d0*(x(1)**2-x(1))*(2.d0*x(1)-1.d0) + 2.d0*(x(1)-1.d0)
       do i=2,n
         g(1) = g(1) - 8.d0*(x(i)**2-x(1))
       end do

       do i=2,n
         g(i) = 8.d0*(x(i)**2-x(1))*(2.d0*x(i)) + 2.d0*(x(i)-1.d0)
      end do
       return


cF76                               NONSCOMP (CUTE)
*
*                       Initial point x0=[3., 3., ..., 3.].
*
76    continue

       g(1) = 2.d0*(x(1)-1.d0) - 8.d0*(x(2)-x(1)**2)*(2.d0*x(1))

       do i=2,n-1
         g(i) = 8.d0*(x(i)-x(i-1)**2) - 8.d0*(x(i+1)-x(i)**2)*(2.d0*x(i))
       end do

       g(n) = 8.d0*(x(n)-x(n-1)**2)
       return


cF77                               Linear Perturbed
*
*                     Initial point x0=[2., ...,2.]
*
77     continue

       do i=1,n
         g(i) = float(i)*2.d0*x(i) + 1.d0/100.d0
       end do
       return



cF78                               CUBE
*
*                      Initial point x0=[-1.2, 1, -1.2, 1.,....,-1.2, 1]
*
78     continue

       g(1) = 2.d0*(x(1)-1.d0) - 600.d0*(x(2)-x(1)**3)*(x(1)**2)
       do i=2,n-1
         g(i) = 200.d0*(x(i)-x(i-1)**3) - 600.0*(x(i+1)-x(i)**3)*(x(i)**2)
       end do
       g(n) = 200.d0*(x(n)-x(n-1)**3)
       return


cF79                               HARKERP
*
```

```
*                       Initial point x0=[1,2,...,n]
*
79     continue

       s = 0.d0
       do i=1,n
         s = s + x(i)
       end do

       do i=1,n
         g(i) = 2.d0*s - 1.d0 - x(i)
       end do
       return

cF80                              QUARTICM
*
*                       Initial point x0=[2,2,...,2]
*
80     continue

       do i=1,n
         g(i) = 4.d0*(x(i)-float(i))**3
       end do
       return
       end

c ================================================ Last line CG_GRAD
c
c
c
c
c

c
c ================ Last line CG3x8.FOR package ================
c
c
c                        Dr. Neculai Andrei
c           Center for Advanced Modeling and optimization,
c                  Academy of Romanian Scientists,
c                        Bucharest, Romania
c
c
c==============================================================
```

---ooooOoooo---