JOURNAL OF OPTIMIZATION THEORY AND APPLICATIONS: Vol. 69, No. 1, APRIL 1991

Efficient Generalized Conjugate Gradient Algorithms, Part 2: Implementation

Y. F. HU[1] AND C. STOREY[2]

Communicated by L. C. W. Dixon

Abstract. In Part 1 of this paper (Ref. 1), a new, generalized conjugate gradient algorithm was proposed and its convergence investigated. In this second part, the new algorithm is compared numerically with other modified conjugate gradient methods and with limited-memory quasi-Newton methods.

Key Words. Unconstrained optimization, hybrid and restart conjugate gradient methods, limited-memory methods, inexact line search.

1. Introduction

In Part 1 of this paper (Ref. 1), Liu and Storey proposed a new conjugate gradient algorithm for solving the unconstrained minimization problem

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

The algorithm was shown to be globally convergent under standard line search conditions when f is twice continuously differentiable and has a bounded level set. The purpose of this second part is to compare the new algorithm (LS algorithm) with Dixon's implementation (Ref. 2) of Nazareth's three-term recurrence method and with the recent limited-memory quasi-Newton methods of Liu and Nocedal (Ref. 3). An interesting review of modified conjugate gradient algorithms can be found in Nazareth (Ref. 4).

As with conventional conjugate gradient methods, the new method uses a combination of the previous search direction and the current gradient

[1] Graduate Student, Department of Mathematical Sciences, Loughborough University of Technology, Loughborough, Leicestershire, England.

[2] Professor, Department of Mathematical Sciences, Loughborough University of Technology, Loughborough, Leicestershire, England.

0022-3239/91/0400-0139$06.50/0

to form the new direction in a general iteration. An important difference, however, is that instead of the new direction being conjugate (for quadratic functions) to the previous direction, it is conjugate to a vector which is perpendicular to the current gradient, and so the new method becomes the conventional method when the line search is exact.

It is also possible to regard the LS method as a combination of a conjugate gradient method and Newton's method, since it is, in fact, a two-dimensional Newton method in the sense that it uses, as the new direction, the Newton direction of the restriction of f on span $\{g, s\}$, with g the current gradient and s the previous search direction. On the subspace span $\{g, s\}$, the Hessian at the current point is

$$\begin{bmatrix} g^T Hg & g^T Hs \\ s^T Hg & s^T Hs \end{bmatrix}, \tag{2}$$

where $H = \nabla^2 f$ and the gradient at the current point is $[g^T g, g^T s]^T$. Thus, the new direction is given by

$$s^+ = \alpha g + \beta s, \tag{3}$$

where

$$\begin{bmatrix} \alpha \\ \beta \end{bmatrix} = -\begin{bmatrix} g^T Hg & g^T Hs \\ s^T Hg & s^T Hs \end{bmatrix}^{-1} \begin{bmatrix} g^T g \\ g^T s \end{bmatrix}. \tag{4}$$

Clearly, this idea could be extended to form an m-dimensional Newton method, with $2 \leq m \leq n$, and the m-dimensional subspace could be selected in many different ways. Such extensions are not dealt with in this paper, but similar ideas can be found in Nazareth (Ref. 5).

Before going on to investigate the performance of several different implementations of the LS algorithm, it is convenient to restate the algorithm in full.

2. Algorithms A1 and A2 of Ref. 1

Algorithm A1.

Step 1. Set $k = 1$, $s_1 = -g_1$.

Step 2. Line Search. Compute $x_{k+1} = x_k + a_k s_k$; set $k = k + 1$.

Step 3. If $g_k^T g_k < \epsilon$, then stop; otherwise, go to Step 4.

Step 4. If $k > n > 2$, go to Step 8; otherwise, go to Step 5.

Step 5. Let $t_k = s_{k-1}^T H_k s_{k-1}$, $v_k = g_k^T H_k g_k$, and $u_k = g_k^T H_k s_{k-1}$.

Step 6. If

$$t_k > 0, \qquad v_k > 0, \qquad 1 - u_k^2/(t_k v_k) \geq 1/(4r), \tag{5}$$

and

$$(v_k/g_k^T g_k)/(t_k/s_{k-1}^T s_{k-1}) \leq r, \qquad r > 0, \tag{6}$$

then go to Step 7; otherwise, go to Step 8.

Step 7. Let

$$s_k = [(u_k g_k^T s_{k-1} - t_k g_k^T g_k) g_k + (u_k g_k^T g_k - v_k g_k^T s_{k-1}) s_{k-1}]/w_k,$$

where $w_k = t_k v_k - u_k^2$; go to Step 2.

Step 8. Set x_k to x_1; go to Step 1.

Algorithm A2. In practice, to avoid the computation and the storage of H_k, the product of H_k and a vector in Step 5 is computed using some form of finite-difference approximation. For example, Step 5 could be replaced by the following step.

Step 5. Let

$$t_k = s_{k-1}^T (f'(x_k + \delta s_{k-1}) - g_k)/\delta, \tag{7a}$$

$$u_k = g_k^T (f'(x_k + \delta s_{k-1}) - g_k)/\delta, \tag{7b}$$

$$v_k = g_k^T [f'(x_k + \gamma g_k) - g_k]/\gamma, \tag{7c}$$

where δ and γ are suitable, small, positive numbers.

3. Test Problems

We have used the following ten test problems: Penalty 1, Trigonometric, Extended Rosenbrock, Extended Powell, Tridiagonal, Matrix Square Root 1, Matrix Square Root 2, Extended Beale, Extended Wood, and Penalty 2. These problems are numbered from 1 to 10 in the above order. For each of Problems 1–5 and 8–10, the dimensions $n = 100, 200, \ldots, 1000$ have been used. For Problems 6–7, because of their special structure, we have used $n = m^2$ with $m = 2, 3, \ldots, 10$.

Most of these test problems are highly separable, and as will be seen later the numbers of iterations to reach specified convergence criteria are, in general, independent of the increase in dimension. Problems 5–7 are exceptions, and the number of iterations tends to increase as n increases.

Details of the test functions are given in the Appendix (Section 6).

4. Implementation of the Algorithms

4.1. Choice of Parameters. We have used the line search in NAG routine EO4DBF (a version of the Fletcher–Reeves conjugate gradient method) with a slight modification, namely that the initial step length is taken to be

$$\lambda_0 = \min\{2, -2[f(x_k) - \text{Est}]/g_k^T s_k\},$$

where Est is an estimate of (preferably lower than) the optimal function value. For all the test problems considered, we set Est = 0, since the optimal function values are all greater than or equal to zero. In all cases, unless specified otherwise, the stopping condition (Ref. 3)

$$\|g_k\| < 10^{-5} \max\{1, \|x_k\|\}$$

is used.

Liu and Storey (Ref. 1) suggested taking

$$r = 1/\sqrt{\eta}, \qquad \delta = \sqrt{\eta}/\sqrt{s_{k-1}^T s_{k-1}}, \qquad \gamma = \sqrt{\eta}/\sqrt{g_{k-1}^T g_{k-1}},$$

with η the computer accuracy; they also used backward differences in Step 5, but we have found forward differences work rather better. The computer used is the Multics Honeywell System with $\eta = 1.08420217248550444\text{D} - 19$. For a fixed value of r, changing $\sqrt{\eta}$ in δ and γ within a reasonable interval, say $[4\times 10^{-10}, 10^{-3}]$, seems to have very little effect on the results and so we recommend taking $\sqrt{\eta}$ in δ and γ as 4×10^{-10}.

The effect of fixing $\sqrt{\eta}$ in δ and γ at 4×10^{-10} and changing it in r was investigated on Problems 3, 4, 5, 8, 9, with $n = 100$ and stopping condition $\|g_k\|^2 < 10^{-4}$. The results did not vary very much for $\sqrt{\eta}$ less than about 10^{-3}; as $\sqrt{\eta}$ increased, the results were better for some problems (4 and 9) and worse for others to begin with; but eventually, too large a value for $\sqrt{\eta}$ (greater than about 10^{-1}) gave a universally poorer performance.

The reason for the variations in the results is that $\sqrt{\eta}$ is used in Step 6 of the algorithm to ensure global convergence. The larger the value of $\sqrt{\eta}$, the smaller the angle allowed between s_k and g_k (to bound it away from $\pi/2$), with a consequent increase in the number of restarts, which is good for some functions but not for others. When $\sqrt{\eta} > 4$, the algorithm becomes equivalent to the steepest-descent algorithm. These observations lead us to fix $\sqrt{\eta}$ at 4×10^{-10}, which satisfies the condition for convergence but also allows reasonable angle criteria for restarting.

4.2. Algorithms A2 and A3. Algorithm A2 was now used on all 10 test functions and some of the results are shown in Table 2. We have in fact, to save space, only given the results for $n = 100$ and 1000 for all

Table 1. Performance of Liu and Nocedal algorithm.

P	n	NI	NF	NG	CL	CPU	NR
1	100	7	18	18	1818	—	—
1	1000	12	32	32	32,032	—	—
2	100	44	89	89	8989	—	—
2	1000	46	94	94	9494	—	—
3	100	31	73	73	7373	—	—
3	1000	31	73	73	73,073	—	—
4	100	25	54	54	5454	—	—
4	1000	37	79	79	79,079	—	—
5	100	74	149	149	15,049	—	—
5	1000	281	563	563	563,563	—	—
6	49	—	—	—	—	—	—
6	100	234	471	471	47,571	—	—
7	49	—	—	—	—	—	—
7	100	224	449	449	45,349	—	—
8	100	—	—	—	—	—	—
8	1000	—	—	—	—	—	—
9	100	—	—	—	—	—	—
9	1000	—	—	—	—	—	—
10	100	—	—	—	—	—	—
10	1000	—	—	—	—	—	

A dash signifies that no result is available.

problems except Problem 6 and Problem 7, for which $n = 49$ and 100. The complete sets of results are available from the authors. For comparison, we show in Table 1 the results from Liu and Nocedal (Ref. 3, Table 4) with accurate line search and $m = 5$. In Tables 1–7, the results are in the form of numbers of iterations NI, number of function calls NF, number of gradient calls NG, computational labor CL, central processor unit time CPU, and number of restarts NR.

As can be seen from Tables 1 and 2, in terms of number of iterations, Algorithm A2 is very good, except on the Powell singular function (Problem 4) which needs a very large number of iterations. Recall that, in the discussion on the choice of parameters, we observed that frequent restart was preferable for this function. Therefore, we tried Powell's restart condition (Ref. 6) to see if this would improve the situation; this condition demands a restart with current gradient direction, provided

$$|g_k^T g_{k-1}| \geq 0.2\|g_k\|^2;$$

the resulting algorithm (Algorithm A3) gave the results in Table 3.

Clearly, Algorithm A3 performs well on the Powell function (4) and the Wood function (9), but badly on the Rosenbrock function (3). In general,

Table 2. Performance of Algorithm A2.

P	n	NI	NF	NG	CL	CPU	NR
1	100	12	39	61	6139	1.84	2
1	1000	13	42	66	66,042	19.51	3
2	100	51	103	203	20,403	25.24	1
2	1000	46	93	183	183,093	233.79	1
3	100	12	32	54	5432	1.81	1
3	1000	12	32	54	54,032	18.08	1
4	100	194	393	777	78,093	21.57	2
4	1000	1002	2007	4007	4009,007	1128.67	2
5	100	75	151	299	30,051	9.03	1
5	1000	282	565	1127	1127,565	347.44	1
6	49	103	207	407	20,150	40.90	3
6	100	212	425	843	84,725	237.58	3
7	49	167	335	661	32,724	66.30	4
7	100	238	477	947	95,177	265.43	3
8	100	7	18	30	3018	1.00	1
8	1000	7	18	30	30,018	10.02	1
9	100	44	91	177	17,791	4.55	2
9	1000	44	91	177	177,091	46.02	2
10	100	4	12	18	1812	0.55	1
10	1000	5	16	24	24,016	6.89	1

Algorithm A3 is slightly better than Algorithm A2 and certainly makes the results more even; but to offset this, the former requires the storage of one more vector than the latter.

4.3. Algorithms A4 and A5. From the results in Section 4.2, it can be seen that the number of function calls and the number of gradient calls are about 2 times and 4 times the number of iterations, respectively. This is because, to compute the new direction in Step 7, we need to estimate t_k, v_k, u_k by finite differences, which requires two gradient calls per iteration. Moreover, we use a quite exact line search, which forces at least one cubic interpolation at each iteration requiring a further two gradient and two function calls. Thus, despite the very competitive iteration number, the computational labor and CPU time are not so satisfactory. We therefore attempt to seek ways for reducing the function and gradient calls.

In the introduction, the LS method was described as a two-dimensional Newton method and, as is well known, the best steplength for Newton-like methods is 1. Actually, in Step 7 of Algorithm A1 or A2, by using second-derivative information in t_k, u_k, v_k, the direction s_k has already been scaled by a factor w_k, which would make $x_k + s_k$ the minimizer along s_k if f were

Table 3. Performance of Algorithm A3.

P	n	NI	NF	NG	CL	CPU	NR
1	100	16	49	65	6549	2.05	8
1	1000	15	44	60	60,044	18.78	7
2	100	48	97	183	18,397	23.80	6
2	1000	49	99	187	187,099	245.50	5
3	100	24	56	84	8456	3.06	11
3	1000	24	56	84	84,056	30.32	11
4	100	49	104	178	17,904	5.37	12
4	1000	49	104	178	178,104	53.07	12
5	100	75	151	299	30,051	9.71	1
5	1000	282	565	1127	1127,565	367.13	1
6	49	99	199	391	19,358	39.83	3
6	100	211	423	835	84,123	236.18	4
7	49	168	337	663	32,824	66.93	5
7	100	235	471	931	93,571	263.68	5
8	100	8	21	31	3121	1.09	3
8	1000	8	21	31	31,021	10.85	3
9	100	33	70	120	12,070	3.38	9
9	1000	33	70	120	120,070	33.38	9
10	100	4	12	12	1212	0.38	4
10	1000	6	18	20	20,018	6.24	5

a quadratic function. Thus, it is to be expected that steplength 1 would be satisfactory, especially as the optimum of f is approached.

Algorithm A4 therefore replaces this accurate line search with one that firstly checks to see if the step $\lambda = 1$ satisfies the conditions

$$f(x_k + \lambda s_k) \leq f(x_k) + \beta' \lambda g_k^T s_k, \tag{8}$$

$$g(x_k + \lambda s_k)^T s_k \geq \beta g_k^T s_k, \tag{9}$$

with $\beta' = 10^{-4}$, $\beta = 0.9$, and only carries out a normal line search if this is not the case.

Of course, whenever a restart is made, this procedure cannot be carried out, and so a normal line search with at least one cubic interpolation is required. Algorithm A5 is just Algorithm A4 with a Powell restart condition included. Tables 4 and 5 show a selection of results using the two algorithms.

A comparison of Tables 4 and 5 and Tables 2 and 3 shows that considerable improvement has been made in terms of gradient calls, computational labor, and CPU time. Now, the number of function calls is about equal to the number of iterations and the number of gradient calls is about three times the number of iterations, demonstrating that the unit steplength is usually accepted. A curious feature of these results is that Algorithm A4

Table 4. Performance of Algorithm A4.

P	n	NI	NF	NG	CL	CPU	NR
1	100	23	33	77	7733	2.42	1
1	1000	17	30	62	62,030	19.49	2
2	100	47	50	142	1425	17.46	2
2	1000	48	50	144	14,405	177.46	1
3	100	20	27	65	6537	2.30	2
3	1000	20	27	65	65,027	23.08	2
4	100	35	38	106	10,638	3.28	1
4	1000	43	46	130	130,046	40.17	1
5	100	76	78	228	22,878	7.66	1
5	1000	281	283	843	843,283	284.45	1
6	49	96	99	287	14,162	28.37	2
6	100	211	215	631	63,315	174.21	3
7	49	147	151	439	21,662	43.69	3
7	100	255	259	763	76,559	209.67	3
8	100	7	11	23	2311	0.83	1
8	1000	7	11	23	23,011	8.21	1
9	100	59	65	181	18,165	5.26	2
9	1000	59	65	181	181,065	52.37	2
10	100	4	9	15	1509	0.46	1
10	1000	9	15	31	3105	9.73	1

needs fewer iterations than Algorithm A5 on the Powell function. In general, the effect of restart is not so noticeable as it was in Tables 2 and 3, and Algorithms A4 and A5 are quite compatible but both substantially better than Algorithms A2 and A3.

4.4. Algorithms A6 and A7. Although Algorithm A4 and Algorithm A5 are both considerably better than Algorithms A2 and A3, the fact that the gradient call is about three times the number of iterations is still very undesirable.

By the mean-value theorem,

$$g_k - g_{k-1} \approx H(\xi)(x_k - x_{k-1}), \qquad \xi \in (x_k, x_{k-1}),$$

so setting

$$y_k = g_k - g_{k-1}, \qquad x_k - x_{k-1} = \lambda_{k-1} s_{k-1},$$

and approximating $H(\xi)$ by H_k, we have

$$H_k s_{k-1} \lambda_{k-1} \approx y_k. \tag{10}$$

So instead of calculating $H_k s_{k-1}$ by finite differences, we can use the relation

$$H_k s_{k-1} \approx y_k / \lambda_{k-1},$$

Table 5. Performance of Algorithm A5.

P	n	NI	NF	NG	CL	CPU	NR
1	100	22	51	75	7551	2.42	10
1	1000	19	45	65	60,045	20.78	9
2	100	45	52	132	13,252	16.45	5
2	1000	44	50	128	128,050	161.75	5
3	100	29	54	86	8654	3.22	13
3	1000	29	54	86	86,054	31.95	13
4	100	43	52	124	124,528	3.96	7
4	1000	45	54	130	130,054	41.70	7
5	100	76	78	228	22,878	7.97	1
5	1000	281	283	843	843,283	292.12	1
6	49	97	102	288	14,214	28.25	4
6	100	211	217	629	63,117	174.81	5
7	49	171	178	508	2507	50.47	6
7	100	242	248	722	72,448	199.51	5
8	100	11	19	31	3119	1.15	5
8	1000	11	19	31	31,019	11.52	5
9	100	34	48	100	10,048	2.98	9
9	1000	34	48	100	100,048	29.51	9
10	100	4	12	12	1212	0.37	4
10	1000	7	19	21	21,019	6.59	6

giving a reduction of one gradient call per iteration. Using this approximation results in Algorithm 6. Notice that, in implementing this idea, although the approximation (10) is used, the switching conditions (5) and (6) need not be changed, since they will still ensure global convergence. As in the two previous sections, Algorithms A6 was combined with Powell's restart to give Algorithm A7. The results obtained with these last two algorithms are shown in Tables 6 and 7. Clearly, the number of gradient evaluations is now roughly twice the number of iterations but the number of iterations has increased. Thus, Algorithm A6 is slightly better than Algorithm A4 in terms of number of gradient computations, but in terms of CPU time there is little to choose between them. Again, the effect of restart on Algorithm A6 is inconclusive.

Finally, the results of Algorithms A6 and A7 are compared in Table 8 (here only the computational labor is given and the stopping condition is $\|g\|^2 < 10^{-4}$) with the results from the three-term recurrence conjugate gradient methods reported in Dixon (Ref. 2, Tables 1 and 2). It is clear that, in terms of computational labor, the new method is not as good as the best of the three-term methods, but it does work on the Wood problem (9), which defeats the latter.

Table 6. Performance of Algorithm A6.

P	n	NI	NF	NG	CL	CPU	NR
1	100	25	36	60	6036	2.72	3
1	1000	21	37	57	57,037	21.16	4
2	100	47	52	98	9852	13.31	1
2	1000	44	47	90	90,047	126.19	1
3	100	30	48	77	7748	3.28	5
3	1000	30	48	77	77,048	32.54	5
4	100	106	112	216	21,712	8.55	2
4	1000	251	261	511	511,261	202.13	1
5	100	76	78	153	15,378	6.41	1
5	1000	281	283	563	563,283	239.43	1
6	49	102	107	206	10,201	22.04	3
6	100	208	213	418	42,013	122.47	3
7	49	148	155	299	14,806	32.11	4
7	100	242	247	486	48,847	142.96	3
8	100	13	20	32	3220	1.34	2
8	1000	13	20	32	32,020	13.24	2
9	100	80	93	172	172,293	6.34	6
9	1000	72	85	156	156,085	56.70	6
10	100	4	10	13	1310	0.44	2
10	1000	6	15	20	20,015	6.72	3

Table 7. Performance of Algorithm A7.

P	n	NI	NF	NG	CL	CPU	NR
1	100	25	56	70	7056	2.55	11
1	1000	25	54	68	68,054	24.41	11
2	100	47	54	96	19,456	13.73	6
2	1000	53	61	108	108,061	154.76	6
3	100	35	64	84	8464	3.59	15
3	1000	35	64	84	84,064	35.52	15
4	100	54	75	111	11,172	4.31	15
4	1000	55	73	113	113,073	43.06	15
5	100	76	78	153	15,378	6.72	1
5	1000	281	283	563	563,283	247.79	1
6	49	101	107	204	10,103	22.15	4
6	100	204	210	409	41,110	121.32	5
7	49	156	164	313	15,501	33.59	7
7	100	242	248	485	48,748	143.48	5
8	100	12	22	27	2722	1.12	7
8	1000	12	22	27	27,022	10.89	7
9	100	44	66	97	9766	3.45	15
9	1000	44	66	97	97,066	34.16	15
10	100	4	12	12	1212	0.38	4
10	1000	7	19	20	20,019	6.51	6

Table 8. Comparison of Algorithms A6 and A7 with three-term recurrence methods.

	Problem 3			P9	Problem 4			Problem 5		
n	2	10	20	4	4	60	80	10	20	30
OPCG1	386	974	411	—	127	2111	6646	119	439	831
OPCG2	170	393	801	F4	119	1688	2055	119	439	831
OPCG3	386	782	2326	—	127	2183	1566	119	439	831
OPCG4	170	454	888	F4	126	2122	2964	119	439	831
BLALN	—	671	1281	—	—	7991	5913	209	819	1891
VA14A	195	671	1281	F2	345	2623	3483	297	819	1643
VA08A	276	957	2100	640	365	9577	16,443	440	1029	2015
FRCG	495	1362	2511	1111	104	3906	7463	119	439	863
Algorithm A7	187	839	1661	408	308	3760	5000	201	760	1557
Algorithm A6	197	809	1547	590	260	2664	3544	201	760	1557

A dash signifies that no result is available.

5. Conclusions

We have investigated the behavior of six different implementations of a generalized conjugate gradient method. Table 9 shows the storage requirements of these six algorithms along with those for the three-term conjugate gradient method and the limited-memory BFGS method.

In terms of CPU time, Algorithms A4, A5, A6, A7 are compatible with each other and outperform Algorithms A2 and A3. In terms of number of gradient calls, Algorithms A6 and A7 are best. We find that the restart condition proposed by Powell works well on some functions, but is poor on others; clearly, a more robust restart criterion is needed.

Generally speaking, the LS conjugate gradient method appears to be quite efficient especially in terms of iteration numbers and has modest

Table 9. Storage requirements of algorithms used.

Algorithm	A2	A3	A4	A5	A6	A7	3TRM	LBFGS
Storage	$5n$	$6n$	$5n$	$6n$	$6n$	$6n$	$13n$	$\sim 13n$

3TRM = three-term recurrence method.

storage requirements. Further work is under way to improve and refine this algorithm and to test it against other methods on the same computer. We are also studying the use of the new method in dynamic optimization. We have proved that the method is n-step superlinearly convergent, but the proof is technical and closely follows that of Cohan (Ref. 7), and so is not given here.

6. Appendix: Test Problems

Problem 1. Penalty 1 Function:

$$F(x) = 10^{-5} \sum_{j=1}^{n} (x_j - 1)^2 + \left[\sum_{j=1}^{n} x_j^2 - 0.25 \right]^2, \qquad n = 1, 2, \ldots,$$

$$(x_0)_j = j, \qquad j = 1, 2, \ldots, n.$$

Problem 2. Trigonometric Function:

$$F(x) = \sum_{i=1}^{n} \left\{ n + i - \sum_{j=1}^{n} [a_{ij} \sin(x_j) + b_{ij} \cos(x_j)] \right\}^2, \qquad n = 1, 2, \ldots,$$

$$a_{ij} = \delta_{ij}, \qquad b_{ij} = i\delta_{ij} + 1, \qquad \delta_{ij} = \begin{cases} 0, & i \neq j, \\ 1, & i = j, \end{cases}$$

$$x_0 = (1/n, 1/n, \ldots, 1/n)^T.$$

Problem 3. Extended Rosenbrock Function:

$$F(x) = \sum_{j=1}^{n/2} [100(x_{2j} - x_{2j-1}^2)^2 + (1 - x_{2j-1})^2], \qquad n = 2, 4, \ldots,$$

$$x_0 = (-1.2, 1.0, -1.2, 1.0, \ldots, -1.2, 1.0)^T.$$

Problem 4. Extended Powell Function:

$$F(x) = \sum_{j=1}^{n/4} [(x_{4j-3} + 10x_{4j-2})^2 + 5(x_{4j-1} - x_{4j})^2 + (x_{4j-2} - 2x_{4j-1})^4 + 10(x_{4j-3} - x_{4j})^4], \qquad n = 4, 8, \ldots,$$

$$x_0 = (3, -1, 0, 3, 3, -1, 0, 3, \ldots).$$

Problem 5. Tridiagonal Function:

$$F(x) = \sum_{i=2}^{n} [i(2x_i - x_{i-1})^2],$$

$$x_0 = (1, 1, 1, \ldots, 1)^T.$$

Problem 6. Matrix Square Root 1. This is a matrix-square-root problem. Given $A \in R^{m \times m}$, we wish to find B such that $B^2 = A$. This can be solved by minimizing the function $\|B^2 - A\|_2$. Let

$$x(m(i-1)+j) = B(i, j), \qquad i, j = 1, \ldots, m,$$

$$a(m(i-1)+j) = A(i, j), \qquad i, j = 1, \ldots, m,$$

which map A and B into vectors x and a. Then, the test function is

$$F(x) = \sum_{i=1}^{n} \left[a(i) - \sum_{j=1}^{m} x(j+lm)x(k+(j-1)m) \right]^2,$$

$$l = \text{mod}((i-1), m), \qquad k = 1 + \text{int}((i-1)/m).$$

We specify the matrix A by giving the solution B. For Problem 6, let

$$x_*(i) = \sin(i^2), \qquad i = 1, \ldots, m^2.$$

The initial point is

$$x_0(i) = x_*(i) - 0.8 \sin(i^2), \qquad i = 1, \ldots, m^2.$$

Problem 7. Matrix Square Root 2. For this problem, we let

$$x_*(i) = \begin{cases} \sin(i^2), & i = 1, \ldots, 2m, 2m+2, \ldots, m^2, \\ 0, & i = 2m+1. \end{cases}$$

The initial point is

$$x_0(i) = x_*(i) - 0.8 \sin(i^2), \qquad i = 1, \ldots, m^2.$$

Problem 8. Extended Beale Function:

$$F(x) = \sum_{i=1}^{n/2} \{[1.5 - x_{2i-1}(1 - x_{2i})]^2 + [2.25 - x_{2i-1}(1 - x_{2i}^2)]^2 + [2.625 - x_{2i-1}(1 - x_{2i}^3)]^2\}, \qquad n = 2, 4, \ldots,$$

$$x_0 = (1, 1, \ldots)^T.$$

Problem 9. Extended Wood Function:

$$F(x) = \sum_{i=1}^{n/4} [100(x_{4i-2} - x_{4i-3}^2)^2 + (1 - x_{4i-3})^2 + 90(x_{4i} - x_{4i-1}^2)^2 + (1 - x_{4i-1})^2 + 10(x_{4i-2} + x_{4i} - 2)^2 + 0.1(x_{4i-2} - x_{4i})^2], \qquad n = 4, 8, \ldots,$$

$$x_0 = (-3, -1, \ldots, -3, -1)^T.$$

Problem 10. Penalty 2 Function:

$$F(x) = \sum_{j=1}^{n} (x_j - 1)^2 + 10^{-3}\left[\sum_{j=1}^{n} x_j^2 - 0.25\right]^2, \qquad n = 1, 2, \ldots,$$

$$(x_0)_j = j, \qquad j = 1, 2, \ldots, n.$$

References

1. LIU, Y., and STOREY, C., *Efficient Generalized Conjugate Gradient Algorithms, Part 1, Theory*, Journal of Optimization Theory and Applications, Vol. 69, No. 1, pp. 129-137, 1991.
2. DIXON, L. C. W., DUCKSBURY, P. G., and SINGH, P., *A New Three-Term Conjugate Gradient Method*, Technical Report No. 130, Numerical Optimization Centre, Hatfield Polytechnic, Hatfield, Hertfordshire, England, 1985.
3. LIU, D. C., and NOCEDAL, J., *On the Limited-Memory BFGS Method for Large-Scale Optimization*, Technical Report No. NAM-03, Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, Illinois, 1988.
4. NAZARETH, J. L., *Conjugate Gradient Methods Less Dependent on Conjugacy*, SIAM Review, Vol. 28, pp. 501-511, 1986.
5. NAZARETH, J. L., *The Method of Successive Affine Reduction for Nonlinear Minimization*, Mathematical Programming, Vol. 35, pp. 97-109, 1986.
6. POWELL, M. J. D., *Restart Procedures for the Conjugate Gradient Method*, Mathematical Programming, Vol. 12, pp. 241-254, 1977.
7. COHAN, A., *Rate of Convergence of Several Conjugate Gradient Algorithms*, SIAM Journal on Numerical Analysis, Vol. 9, pp. 248-259, 1972.