A Performance Measurement Workbench

by Michael Kupferschmid

21 Jun 26

Contents

1 Introduction

Chapter 26 of *Introduction to Mathematical Programming* explains how computational experiments can be used to study the performance of optimization algorithms. The techniques described there are suitable for many projects, and by using simple shell programs such as the one suggested in §26.4.1 they can be extended to comparisons involving multiple solvers and multiple test problems. In §26.4.2, I suggest writing your own utility programs to perform various other tasks that arise in computational testing and algorithm development.

In practice the conduct of publishable experiments, and the implementation of utility programs to facilitate them, both require the support of a more robust computing infrastructure or **workbench**. This document introduces the Workbench that I have developed and illustrates how to use it. It is a supplement to §26, and it assumes a familiarity with that Chapter, basic concepts of nonlinear optimization, the Classical FORTRAN described in [100], and in a few places Unix shell programming [96]. The Section numbers §□, page numbers p□, and bibliography citations [□] referred to here are all in *Introduction to Mathematical Programming* rather than in this document. Many students[1] contributed to this work, which is a part of my research collaboration with Joe Ecker. This software was constructed for the study of small nasty problems (see §0.2.4).

The Workbench improves upon the environment described in §26 by providing these features.

- **more problem types** The scheme described in §26.2 assumes that each test problem definition file will contain FORTRAN subprograms to compute the function values, gradients, and Hessians specific to its particular objective and constraints. Many interesting test problems have special structure, which allows a single set of formulas to be used for all problems of that kind. Then a single set of subprograms can be used, and each test problem is defined by a data file containing the particular coefficients to be substituted in the general formulas. It should be convenient freely to mix test problems of different kinds in a computational study comparing algorithms. Many type-2 problems (see §8.5) require preprocessing, and the software should accommodate that need.

- **more problem descriptors** The list of test problem attributes given in §26.2.2 omits some, such as the problem's kind, that are often useful to the software described here, and includes others, such as the problem's provenance and aliases, that instead belong in a published catalog (for an example see §7 of *Bilevel Nonlinear Programs* at this website). Additional **problem descriptors** should be included, and they should be communicated in a way that allows each program component to access those it needs.

[1] Steve Dziuban, David Covey†, and Eric Johnson are mentioned in §0.5; others include Darryl Ahner, Jim Glackin, Craig Johnson, Terry Kelly, Dean Mengel, Kelley Mohrmann. Phil Muir, Lee Quintas, Randy Rotte, and Tyge Rugenstein. If I have forgotten to acknowledge anyone I apologize for the inadvertent oversight.

- **incomplete problem descriptors** In preparing a test problem for use in a computational study it is always necessary to conduct informal experiments to ascertain its properties, fair bounds to use in establishing a starting point, the optimal point and objective value, and the optimal values of the dual variables. Often some or all of these quantities are unknown at the outset, so the software should permit the use of a problem definition that is initially incomplete and allow the user to input missing values from the keyboard and control the course of the calculations interactively. The same program can then be run repeatedly to experiment with different values of some problem descriptors, such as variable bounds, while allowing others to be preset.

- **batch processing** It must be possible to invoke the algorithms under test in such a way that no user interaction is required. Then controlled experiments can be conducted in batch mode [100, p227], so that the experimenter's attendance is not required and timing measurements are not contaminated by interference from concurrent processes and the imprecise exclusion of long intervals spent in convenience code.

- **better timing** In §26.3.6 I mentioned the `TIMER` routine described in [100, §15.1.4], which returns overhead-corrected processor times partitioned into timing bins. I wrote the cycle-counting version described in [100, §18.5.4] in February of 2001 for programs running on a single-core 32-bit processor. Modern computers have 64-bit processors with multiple cores, on which the `GETCYC` and `GETMHZ` routines called by that version of `TIMER` no longer work. The `rdtsc` instruction used in `GETCYC` now has a different calling sequence, and it now counts cycles at the processor's base rate. The file `/proc/cpuinfo` read by `GETMHZ` now lists varying speeds for multiple processor threads, from which the base rate cannot be deduced. `TIMER` remains an indispensable measurement tool, so it must run on current hardware and the testing infrastructure must facilitate its use.

- **post-processing** Enough information about each experiment must be captured in files so that software tools can be used afterward to analyze the observations and reveal properties of the algorithms under test.

2 User Interface

The terminal sessions shown below illustrate how the user interface presented by the Workbench achieves some of the goals listed in the previous Section. All of these examples use the solver `EA3`, which is a production implementation[2] of the ellipsoid algorithm in FORTRAN, and the discussion of its behavior assumes a knowledge of the concepts and terminology introduced in §24. Other solvers differ from `EA3` in many particulars, but most can be integrated into the Workbench environment in a similar way.

[2]It differs from the FORTRAN code discussed in §26.3.4, which I wrote to resemble the MATLAB implementation `eacpu.m` of §26.3.3.

In §26.4.1 a FORTRAN compiler named `ftn` is used to combine a **driver** program, an algorithm implementation, and a test problem definition into an executable; here this process is always performed by a shell program named `prepare`.

2.1 Batch Mode

In the example below `prepare` produces an executable `ea3test` to solve Dembo 1b, a geometric programming problem defined by the data file `dem1.gp`. To run the executable in batch mode I supplied the **script file** `EA3dem1.scr`.

```
unix[1] prepare dem1.gp ea3test > /dev/null
unix[2] ea3test EA3dem1.scr
```

When `ea3test` is invoked in command `[2]` it runs silently, because the problem descriptors that it needs are all specified in `dem1.gp` and the experiment descriptors that it needs are specified in `EA3dem1.scr`. Depending on the **experiment descriptors** that are specified in the script, `ea3test` can time the algorithm and write a file containing performance measurement records for later analysis. These two Unix commands could be included in a shell program like the one in §26.4.1 to prepare and run experiments involving multiple solvers and multiple test problems without user intervention.

2.2 Interactive Mode

In the terminal session on the next page, when `ea3test` is prepared in `[3]` the output from `prepare` is allowed to appear on the screen (in the previous example it was redirected to `/dev/null`). This consists of the name of the directory where the test problem definition for `dem1` is found, the name of the program directory where the object code for `EA3` and its driver are found, and the news that the executable `ea3test` was successfully constructed.

For this example, in `[4]` I ran `ea3test` without a script file. As before the needed problem descriptors are all specified in `dem1.gp`, but now some experiment descriptors must be entered at the keyboard so a dialog ensues.

I asked that the algorithm be timed and that performance measurements be written to a file. The prompt for message level offered a default value `0` (only essential communications with the user) to minimize the effects of user interactions on timing, but I entered `2` instead.

The program reports its progress in preparing to run the experiment by printing the name of the performance measurements file it has attached and the pathname to the problem definition file. Then it reports the nominal processor speed that will be used in converting processor cycle counts to wallclock time, and prints the long name of the test problem, `Dembo 1b`.

I set an initial iteration limit of 1000. When those iterations are complete the program offers to show how it spent its time, and I elected to see the report. It reveals that most of the processor time used by the algorithm was devoted to computing gradient vectors. Some of the nonalgorithm time that is reported was spent in writing performance measurement records to `dem1.EA3`, but most of it was consumed in user interactions because I took a

moment to respond to each prompt. Part of the initialization process for timing is the measurement of base processor speed, which involves an intentional pause of 10000 μs, and that accounts for most of the timing overhead.

After the timing summary the program reports the result obtained by the algorithm. In the 1000 iterations that it was permitted to perform it has discovered at least one feasible point; the lowest objective value at any of the feasible points it found is the **record** `value` `FR`. I elected to see the 12 components of the **record point** `XR` having that objective value, which is reported to be inside the bounds that were given as part of the problem definition.

```
unix[3] prepare dem1.gp ea3test
using problem directory /home/mike/Workbench/NGC
using program directory /home/mike/bin/obj
linking of ea3test succeeded
unix[4] ea3test
Time the algorithm? yes
Write performance measurement records? yes
Message level [0]: 2
GETPMF attached performance measurements file dem1.EA3
GETPDF attached problem definition file /home/mike/Workbench/NGC/dem1.gp
TIMER found processor base speed 3071.999 MHz.
Dembo 1b

Initial iteration limit [1]: 1000
Want to see timing bins? yes

 after 10672 timing measurements in 1000 iterations
 bin   quantity                msec    %
   1   miscellaneous          1.156   14.82
   2   function evals         0.999   12.81
   3   gradient evals         5.645   72.37
       total                  7.800  100.00

       nonalgorithm        5127.934
       timing overhead       10.395

      1832 function evaluations
      1000 gradient evaluations

after      1000 iterations, EA3 return code is 0
iteration limit met with XR the best feasible point so far
record value FR =  3.1682674058550209D+00
record point is inside current bounds
Want to see XR? yes
XR
  8.7873542462020182D-01  8.7958966572895658D-01  2.0460417187684516D+00
  1.0690356235363008D-01  2.1039634609736475D+00  2.7126288959094530D-01
  1.4573826206692100D+00  1.0122047205793090D+00  6.8411296897709040D-01
  7.1333644739569613D-01  1.9122357609700276D+00  1.8678766946860927D+00

------------------------------------------------
Want to do more iterations? no
unix[5]
```

2.3 Investigative Mode

If `ea3test` is used to investigate a problem whose attributes are yet to be determined, additional user interactions are required. Below and on the next page `ea3test` is used to solve a problem whose definition includes only an `FCN` routine to compute the function values and a `GRD` routine to compute their gradients. Now the program prompts for the number of inequality constraints `MI`, the number of equality constraints `ME`, the number of variables `N`, and upper and lower bound vectors `XH` and `XL` to use in finding a starting point. If the problem definition included some of these descriptors, the program would prompt only for the ones that were missing.

I answered the questions about timing the algorithm and writing performance measurement records by entering `no`, set the message level at `2` instead of accepting the default level of `1`, and entered an initial iteration limit at the keyboard. If a script file were provided it could answer some or all of these questions about the experiment and then the program would ask only the others.

The starting point for the solution process is the midpoint of the bounds, $\mathbf{x}^0 = [5, 5]^\top$, which happens to be infeasible. I requested `10` iterations, so `EA3` moved from that point to $\mathbf{x}^{10} \approx [11.89, 11.91]^\top$ which is also infeasible. The return code of `1` means that `K=KMAX` with no feasible point found yet.

I increased the iteration limit to `100`, and in the additional 90 iterations the algorithm found at least one feasible point. Because no catalog record value `FR` was supplied with the problem definition, the best of those feasible points became `XR` $\approx [12.34, 12.85]^\top$. From now on the program reports the current record point rather than the current iterate.

This `XR` is outside of the bounds we began with, so I **recentered** the original bounds on the record point and asked for `100` more iterations. Once again the algorithm improved its solution. The new record point `XR` $\approx [15.63, 15.97]^\top$ is within the displaced bounds so I raised the iteration limit to `1000`, but `EA3` returned with `RC=2` at iteration `413` because the ellipsoid matrix $\mathbf{Q}$ became computationally non-positive-definite.

Suspecting that it might be possible for `EA3` to find a still better point if given the opportunity, I recentered the bounds again at half of their original width to construct a new positive-definite ellipsoid, and restarted the algorithm. At iteration `620` the ellipsoid matrix again became non-positive-definite after finding another better point. Continuing the search might turn up a feasible point with an even lower objective value so I recentered again, shrinking the bounds to 25% of their original width, and continued. This time no better point was found, so I gave up.

```
unix[6] prepare ek1new.f ea3test
using problem directory /home/mike/Workbench/NGC
using program directory /home/mike/bin/obj
linking of ea3test succeeded
unix[7] ea3test
Time the algorithm? no
Write performance measurement records? no
Message level [1]: 2
MI= 3
ME= 0
N= 2
XH(1)...= 10 10
XL(1)...= 0 0
```

```
Initial iteration limit: [1]: 10

after        10 iterations, EA3 return code is 1
iteration limit met with no feasible point found so far
X
  1.1885007929611202D+01  1.1910384012262927D+01
-------------------------------------------------
Want to do more iterations? yes
New iteration limit (> 10): 100

after       100 iterations, EA3 return code is 0
iteration limit met with XR the best feasible point so far
*** NEW BEST POINT *** FR= 3.4519873500044291D+03
record point is outside current bounds
XR
  1.2335192430885021D+01  1.2845081561633926D+01
-------------------------------------------------
Start over with recentered bounds? yes
% of previous width: 100
New iteration limit (> 100): 200

after       200 iterations, EA3 return code is 0
iteration limit met with XR the best feasible point so far
*** NEW BEST POINT *** FR= 6.1421210853647574D+02
record point is inside current bounds
XR
  1.5629348556658526D+01  1.5973579260898342D+01
-------------------------------------------------
Want to do more iterations? yes
New iteration limit (> 200): 1000

after       413 iterations, EA3 return code is 2
ellipsoid matrix non-pd with XR the best feasible point
*** NEW BEST POINT *** FR= 6.1421209720340460D+02
record point is inside current bounds
XR
  1.5629490898180746D+01  1.5973768612363482D+01
-------------------------------------------------
Start over with recentered bounds? yes
% of previous width: 50
New iteration limit (> 413): 1000

after       620 iterations, EA3 return code is 2
ellipsoid matrix non-pd with XR the best feasible point
*** NEW BEST POINT *** FR= 6.1421209720340437D+02
record point is inside current bounds
XR
  1.5629490911966785D+01  1.5973768630704686D+01
-------------------------------------------------
Start over with recentered bounds? yes
% of previous width: 50
New iteration limit (> 620): 1000

after       816 iterations, EA3 return code is 2
ellipsoid matrix non-pd with XR the best feasible point
record value FR =  6.1421209720340437D+02
record point is inside current bounds
XR
  1.5629490911966785D+01  1.5973768630704686D+01
-------------------------------------------------
Start over with recentered bounds? no
unix[8]
```

Once the optimal point and optimal value of a new test problem have been found by exploratory calculations like these they can be coded into the problem definition. This point is slightly suboptimal (the true FR is 614.21209720340380) but if a subsequent solution of the problem by this or another algorithm discovers a *** NEW BEST POINT *** it will be reported so that the problem descriptors can be updated.

3 Application Programming Interface

An executable that can be run to conduct a computational experiment, such as ea3test in the examples above, links together a driver program, an optimization subprogram, and subprograms defining the test problem.

3.1 Instrumenting Code

To control an experiment and collect performance measurements, the source code of some or all of these components must be **instrumented** by embedding calls to Workbench subprograms and including data structures that they use. To illustrate that process we will examine the components that make up ea3test. All of this code is written in the Classical dialect of FORTRAN described in [100], but the functionality achieved here could be realized using Modern Fortran or some other compiled programming language. The line numbers appearing on the left in the listings are only for reference and are not part of the source code. Source code lines having a C in column 1 are comments ignored by the compiler.

The driver program is listed in pieces below. The **preamble** begins with a dictionary 6-46 describing the variables appearing in the code, 48-51 receives certain problem descriptors from COMMON storage, 53-59 receives certain experiment descriptors from COMMON storage, and 61-69 declares local variables that must be typed or dimensioned. The subprogram names FCNEA and GRDEA are declared EXTERNAL because they are passed as arguments to EA3. The comment line 71 drawn across the page separates the non-executable statements of the preamble from the executable code below it.

The executable statements of the driver program begin 73-76 with a call to the Workbench routine GETEXP, which arranges for the necessary problem and experiment descriptors to be given values, initializes timing if measurements are to be made, and returns a starting point in X. Then 77-88 comes some sanity checking: if the problem has equality constraints or lacks bounds on the variables the program stops with return code 1. The logical vector SW3 contains switches that tell what is known in the COMMON block /NGC3/, the 4th and 5th members of which are the upper and lower bounds on the variables. Then 89-92 unless the program is running silently or timing is not enabled, the user is asked whether the contents of **timing bins** should be written whenever 107 EA3 returns. The iteration count is 93 initialized to 0 and copied into KNOW to make it available in COMMON block /EXP4/ 57.

```
 1 C
 2 Code by Michael Kupferschmid
 3 C
 4 C     This program invokes EA3 to measure its performance.
 5 C
 6 C     variable  meaning
 7 C     --------  -------
 8 C     D         direction vector workspace for EA3
 9 C     EA3       routine under test
10 C     EXIT      system routine stops program with a return code
11 C     FCNEA     stub times and counts a function evaluation
12 C     FR        record objective function value
13 C     G         gradient vector workspace for EA3
14 C     GETEXP    routine gets information for the computational expt
15 C     GETQS     routine finds ellipsoid containing variable bounds
16 C     GRDEA     stub times and counts a gradient evaluation
17 C     INBDS     T => the record point is inside the bounds
18 C     K         iteration count in EA3
19 C     KMAX      iteration limit
20 C     KNOW      shared iteration count
21 C     LOGBIN    routine writes out timing bin contents
22 C     LOGPMR    routine writes a performance measurement record
23 C     LQS       length of QS
24 C     ME        number of equality constraints (must be 0)
25 C     MI        number of inequalities
26 C     MSGLVL    message level; -1 => batch mode
27 C     N         number of variables
28 C     NEXT      return code from WATNOW; 1=> more iters 2=> new bounds
29 C     NFE       number of function evaluations used so far; unused
30 C     NGE       number of gradient evaluations used so far; unused
31 C     NHE       number of Hessian evaluations used so far; unused
32 C     NONALG    bin for non-algorithm time
33 C     QS        inverse matrix of quadratic form for EA3
34 C     QUERY     routine asks a yes-or-no question
35 C     RC        return code from EA3: 0,1,2,3,4,5,8
36 C     REPORT    routine tells what happened
37 C     SHOW      T => show timing bin counts
38 C     SW3       switches tell what is known in /NGC3/
39 C     TIMER     routine measures CPU time
40 C     TIMING    T => time the algorithm
41 C     WATNOW    routine finds out what to do next
42 C     WPMR      T => performance measurements are being written
43 C     X         initial estimate of the solution/center of ellipsoid
44 C     XH        upper bounds on the variables
45 C     XL        lower bounds on the variables
46 C     XR        record point
```

The next code stanza [96-100] calls GETQS to compute an initial ellipsoid, QS in symmetric storage mode, from the bounds XH and XL. This call is sandwiched between TIMER calls to [97] direct processor time into bin 1 (miscellaneous algorithm time) while GETQS is working and [99] back into bin NONALG (for non-algorithm time) after it is finished. Because the starting point X is the midpoint of the variable bounds and the initial ellipsoid is constructed to enclose them, the starting point is [100] within them.

```
C
C     receive problem descriptors
      COMMON /NGC3/ SW3,MI,ME,N,XH,XL
      LOGICAL*4 SW3(5)
      REAL*8 XH(50),XL(50)
C
C     receive experiment descriptors
      COMMON /EXP2/ MSGLVL,KMAX
      COMMON /EXP3/ WPMR
        LOGICAL*4 WPMR
      COMMON /EXP4/ NFE,NGE,NHE,KNOW
      COMMON /EXPT/ TIMING,NONALG
      LOGICAL*4 TIMING
C
C     set up to call EA3
      EXTERNAL FCNEA,GRDEA
      PARAMETER(LQS=50*51/2)
      REAL*8 QS(LQS),G(50),D(50),X(50),XR(50),FR
      INTEGER*4 RC
C
C     other local variables
      LOGICAL*4 INBDS,QUERY,SHOW
      INTEGER*4 NEXT
C
C ---------------------------------------------------------------------
C
C     set and check test problem and experiment descriptors
C     determine the starting point
C     initialize timing and set the timing bin to NONALG
      CALL GETEXP(X)
      IF(ME.GT.0) THEN
         WRITE(0,991) ME
  991    FORMAT('EA3 cannot solve equality-constrained problems;'/
     ;          'stopping because ME=',I2)
         CALL EXIT(1)
      ENDIF
      IF(.NOT.SW3(4) .OR. .NOT.SW3(5)) THEN
         WRITE(0,992)
  992    FORMAT('EA3 uses bounds to construct starting ellipsoid;'/
     ;          'stopping because XH or XL is missing')
         CALL EXIT(1)
      ENDIF
      SHOW=.FALSE.
      IF(MSGLVL.GE.1 .AND. TIMING) THEN
         SHOW=QUERY('Want to see timing bins?',24)
      ENDIF
      K=0
      KNOW=K
```

The call to EA3 in the next stanza is similarly sandwiched between TIMER calls, so upon entry to that routine processor time is flowing into bin 1 and upon its return is redirected into bin NONALG. EA3 updates K, so the new value is 106 copied into KNOW and 107 if the user requested that timing bins be displayed LOGBIN is called to do so.

```
 95 C
 96 C     construct a starting ellipsoid enclosing the current bounds
 97     2 CALL TIMER(1,1)
 98       CALL GETQS(XH,XL,N, QS,LQS)
 99       CALL TIMER(NONALG,2)
100       INBDS=.TRUE.
101 C
102 C     perform iterations of the algorithm
103     1 CALL TIMER(1,3)
104       CALL EA3(FCNEA,GRDEA,N,MI,KMAX,QS,LQS,G,D,X, K, XR,FR,RC)
105       CALL TIMER(NONALG,4)
106       KNOW=K
107       IF(SHOW) CALL LOGBIN
108 C
109 C     is user interaction desired?
110       IF(MSGLVL.GE.1) THEN
111 C        yes; report what happened
112          CALL REPORT(K,RC,X,FR,XR,N,XH,XL, INBDS)
113 C
114 C        and find out what to do next
115          CALL WATNOW(RC,K,INBDS,TIMING,XR,N, XH,XL, KMAX,NEXT)
116          IF(NEXT.EQ.3 .OR. KMAX.EQ.0) GO TO 3
117          IF(NEXT.EQ.2) GO TO 2
118          IF(NEXT.EQ.1) GO TO 1
119       ENDIF
120 C
121 C     the experiment is over; write the final PMR
122     3 CALL LOGPMR(X,N)
123       CALL EXIT(0)
124       END
```

If user interaction is possible [111-118] the routines REPORT and WATNOW are used to tell what happened and decide what to do next. That can be [116] to stop the program, or [117] to restart with a new ellipsoid, or [118] to re-enter EA3 so that its iterations can continue. If the experiment is over [122] LOGPMR is called to write the final performance measurement record (it returns immediately if that is not being done) and the program stops.

FCNEA and GRDEA, the EXTERNAL routine names passed to EA3 in the driver program, are stubs (see §26.3.2). FCNEA invokes FCN and GRDEA invokes GRD; those routines are part of the problem definition and do nothing but compute a function value or gradient vector. FCNEA and GRDEA are interposed to switch timing bins and increment NFE and NGE. Both routines use and update experiment descriptors in COMMON.

FCNEA begins [32-33] by saving the number of the bin into which processor time is flowing upon entry to the routine and [34] switches timing to bin 2. Then [36-38] it computes the function value and counts the function evaluation. Finally [40-42] it restores the timing bin that was in use upon entry and returns.

```
 2       FUNCTION FCNEA(X,N,I)
 3 C     This routine returns the value of function I at X
 4 C     and increments NFE.
 5 C
 6 C     variable  quantity
 7 C     --------  --------
 8 C     FCN       routine returns value of function I at X
 9 C     I         index of the function to be computed
10 C     N         number of variables
11 C     NBIN      number of timing bin in use
12 C     NBINSV    number of timing bin on entry
13 C     NFE       number of function evaluations used so far
14 C     NONALG    unused; bin for non-algorithm time
15 C     TIMER     timer routine
16 C     TIMING    T => algorithm is being timed
17 C     X         current point
18 C
19 C     formal parameters
20       REAL*8 FCNEA,X(N)
21 C
22 C     receive experiment descriptors from common
23       COMMON /EXP4/ NFE
24       COMMON /EXPT/ TIMING,NONALG,NBIN
25       LOGICAL*4 TIMING
26 C
27 C     local variable
28       REAL*8 FCN
29 C
30 C-----------------------------------------------------------------------
31 C
32 C     save the timing bin number on entry and switch to bin 2
33       NBINSV=NBIN
34       IF(TIMING) CALL TIMER(2,5)
35 C
36 C     compute and count the function value
37       FCNEA=FCN(X,N,I)
38       NFE=NFE+1
39 C
40 C     switch the timing bin back to what it was
41       IF(TIMING) CALL TIMER(NBINSV,6)
42       RETURN
43       END
```

GRDEA, listed on the next page, begins by 34-35 calling LOGPMR. That routine returns immediately if performance measurement records are not being written; otherwise it switches timing to bin NONALG, writes a performance measurement record, and switches timing back to the bin that was in use on entry.

Then GRDEA 37-38 saves the number of the bin that was set on entry and 39 switches timing to bin 3. Next 41-43 it computes the gradient and counts the gradient evaluation. Next 45-46 it updates the iteration count in /EXP4/. EA3 calls GRDEA once in each iteration, so this keeps KNOW synchronized with EA3's internal iteration counter K. Finally 48-50 it restores the timing bin that was in use upon entry and returns.

In FCN, GRD, HSN, and algorithm implementations such as EA3, inequalities are indexed from I=1 to I=MI, equalities from I=MI+1 to I=MI+ME, and the objective at I=MI+ME+1.

```
 2       SUBROUTINE GRDEA(X,N,I,G)
 3 C     This routine returns the gradient of function I at X,
 4 C     increments NGE, and writes a performance measurement record.
 5 C
 6 C     variable  quantity
 7 C     --------  --------
 8 C     G         gradient vector returned
 9 C     GRD       routine returns gradient vector
10 C     I         index of the function whose gradient is wanted
11 C     KNOW      number of iterations used so far
12 C     LOGPMR    routine writes a performance measurement record
13 C     N         number of variables in the problem
14 C     NBIN      number of timing bin in use
15 C     NBINSV    number of timing bin in use on entry
16 C     NFE       unused; number of function evaluations used so far
17 C     NGE       number of gradient evaluations used so far
18 C     NHE       unused; number of Hessian evaluations used = 0
19 C     NONALG    unused; bin for non-algorithm time
20 C     TIMER     timer routine
21 C     TIMING    T => time algorithm
22 C     X         current point
23 C
24 C     formal parameters
25       REAL*8 X(N),G(N)
26 C
27 C     receive experiment descriptors from common
28       COMMON /EXP4/ NFE,NGE,NHE,KNOW
29       COMMON /EXPT/ TIMING,NONALG,NBIN
30       LOGICAL*4 TIMING
31 C
32 C-----------------------------------------------------------------------
33 C
34 C     write a performance measurement record
35       CALL LOGPMR(X,N)
36 C
37 C     save the timing bin number on entry and switch to bin 3
38       NBINSV=NBIN
39       IF(TIMING) CALL TIMER(3,7)
40 C
41 C     compute and count the gradient vector
42       CALL GRD(X,N,I,G)
43       NGE=NGE+1
44 C
45 C     update the shared iteration count
46       KNOW=KNOW+1
47 C
48 C     switch the timing bin back to what it was
49       IF(TIMING) CALL TIMER(NBINSV,8)
50       RETURN
51       END
```

In these routines the time it takes to increment `NFE` or `NGE` is added to timing bin `2` or `3`, but processing that is performed as part of effort measurement should instead be charged to timing overhead. Switching from one timing bin to another is fast, but it takes much longer than incrementing an integer and might introduce more uncertainty to the measurements than it corrects (the variable overhead of a `TIMER` call averages about 9 cycles) so I decided that misallocating the operation is the lesser evil.

3.2 Experiment Descriptors

To study the performance of an algorithm it is necessary to instrument the code that implements it, but for a comparison of algorithms to be unbiased each solver routine should be modified as little as possible[3] in the process. *To avoid disturbing the calling sequence and internals of a solver routine, we must not pass experiment and problem descriptors through it* to reach Workbench routines like `LOGPMR` and `TIMER` or stubs like `FCNEA` and `GRDEA`. The standard way in Classical FORTRAN[4] to pass variables between routines that do not invoke each other is by allowing the routines to share memory by means of `COMMON` storage [100, §8].

The software described here stores experiment descriptors in `COMMON` blocks named `/EXP1/`, `/EXP2/`, `/EXP3/`, and `/EXP4/`; the `TIMER` subroutine stores its measurements in a `COMMON` block named `/EXPT/`. The data in these blocks that are specific to each algorithm implementation are initialized by a `BLOCK DATA` subprogram, having the format shown on the left below, that is part of the solver's driver program. The other variables are set by the `GETEXP` subroutine, which is called at the beginning of the driver program; it reads a script file if one is provided, or prompts the user for any needed data that are missing.

```
BLOCK DATA EXP
COMMON /EXP1/ SHORT,LONG,STAMP
  CHARACTER*1 SHORT(4),LONG(24),STAMP(22)
COMMON /EXP2/ MSGLVL,KMAX,XZERO
  REAL*8 XZERO(50)
COMMON /EXP3/ WPMR,PMF
  LOGICAL*4 WPMR
  CHARACTER*1 PMF(48)
COMMON /EXP4/ NFE,NGE,NHE,KNOW,TNAMES
  CHARACTER*1 TNAMES(16,22)
COMMON /EXPT/ TIMING,NONALG,NBIN,TOPBIN,NTMEAS,TOH,BINCPU,SPEED
  LOGICAL*4 TIMING
  INTEGER*4 TOPBIN,TOH,BINCPU(2,22)
  REAL*8 SPEED
END
```

`SHORT`	short name for algorithm
`LONG`	long name for algorithm
`STAMP`	day-date-time stamp for this experiment
`MSGLVL`	message level; `-1` ⇒ batch mode
`KMAX`	iteration limit
`XZERO`	starting point
`WPMR`	`T` ⇒ write performance measurement records
`PMF`	full name of performance measurements file
`NFE`	number of function evaluations used so far
`NGE`	number of gradient evaluations used so far
`NHE`	number of Hessian evaluations used so far
`KNOW`	current iteration count
`TNAMES`	timing bin names

`TIMING`	`T` ⇒ timing is enabled
`NONALG`	number of bin for non-algorithm time
`NBIN`	number of bin just receiving time
`TOPBIN`	highest bin number used so far
`NTMEAS`	number of timing measurements made so far
`TOH`	timing overhead correction in cycles
`BINCPU`	bin CPU times in [sec,nsec] format
`SPEED`	processor speed in MHz

The tables on the right above describe the variables in these `COMMON` blocks. The vector `XZERO` is initialized by `GETEXP` to the starting point `(XH+XL)/2.`

[3] In some cases a solver must be rewritten as a subroutine or revised to exclude convenience code from algorithm time, or it can be modified to be serially reusable without affecting its performance; see §26.3.6.

[4] Modern Fortran provides the elegant but more complicated and paternalistic `MODULE` construct to achieve this result; C routines can share a named block of memory by defining it as a `struct` and declaring it `extern` wherever it is used.

3.2.1 Message Levels

The integer MSGLVL controls the verbosity of output from the test program according to the conventions outlined below.

MSGLVL	verbosity of user interactions
all	explain errors that stop the program
-1	expect needed experiment descriptors to be defined in EXP expect needed problem descriptors to be defined in NGC stop the program with return code 1 if any are not
0	avoid unnecessary communication with the user but prompt for needed descriptors that are missing
1	report progress of the algorithm and prompt for instructions when K=KMAX or the solver stops
2	and report new record points
3	and report every iterate generated
4	and report problem descriptors and calculation details

Batch mode (MSGLVL=-1) and quiet mode (MSGLVL=0) assume that the progress of the algorithm will be recorded only by writing performance measurement records, and that the experiment will consist only of running the algorithm for KMAX iterations or until it stops. The ea3test driver and stub routines described in the previous Section make no provision for reporting every iterate (which could be done in GRDEA) or the details of calculations performed internal to EA3 (which would require the insertion of code there, bracketed by appropriate calls to TIMER).

3.2.2 The Script File

BLOCK DATA EXP is part of a solver's driver program so it specifies algorithm-specific experiment descriptors that do not change from one experiment to another. The other experiment descriptors are left unset so that they will be prompted for and the user can vary them from one experiment to another. To avoid this user interaction a **script file** can be supplied to set the values of those experiment descriptors. The default name of this file is constructed to indicate both the algorithm under test and the test problem being used.

```
[short name for algorithm][file name abbreviation for problem].scr
```

A script file suitable for batch runs of EA3 on the problem dem1 is EA3dem1.scr, which is shown below.

```
MSGLVL=-1
KMAX=100
WPMR=T
PMF=/home/mike/Workbench/dem1.EA3
TIMING=F
```

Some or all of these experiment descriptors can be set, in any order, using the syntax shown; any that remain unset but are needed will be prompted for.

3.3 Problem Descriptors

The problem definition file for `ek1` that is discussed in §26.3.5 includes a `BLOCK DATA` subprogram initializing a single `COMMON` block with the catalog number; the number of variables, inequalities, and equalities; variable bounds; the optimal point; and the KKT multipliers. In practice it is convenient to be able to make more information about each test problem available in `COMMON`, and to permit the use of a test problem definition file in which some data are missing or should be temporarily ignored. The `BLOCK DATA NGC`[5] subprogram listed below allows for the specification of more problem attributes, and it divides them into several `COMMON` blocks so that when a routine accesses the attributes it needs as few as possible unneeded ones are also revealed.

Each `COMMON` block contains as its first element a vector of switches, which indicate whether the following data elements are set. For example, if `SW1(1)` has the value `.TRUE.` then a value has been provided for `NBR`, the catalog number of the problem. It is essential to the proper functioning of the Workbench software that these switches accurately reflect the status of the data in `BLOCK DATA NGC`.

The driver program for an algorithm implementation begins by calling `GETEXP`. As explained above, that routine gets any experiment descriptors that are needed but not already set in `BLOCK DATA EXP`. Using the values of certain experiment descriptors, `GETEXP` determines what problem descriptors are needed. Then it calls `GETPRB` to fill in any that are missing, and if user interaction is permitted they are prompted for and read from the keyboard.

```
BLOCK DATA NGC
COMMON /NGC1/ SW1,NBR,PROBID,FNABBR
  LOGICAL*4 SW1(3)
  CHARACTER*1 PROBID(48),FNABBR(8)
COMMON /NGC2/ SW2,PKC,FLAGS,DEFFIL,NLPDIR
  LOGICAL*4 SW2(4)
  INTEGER*4 PKC
  INTEGER*4 FLAGS(3)
  CHARACTER*1 DEFFIL(48)
  CHARACTER*48 NLPDIR
COMMON /NGC3/ SW3,MI,ME,N,XH,XL
  LOGICAL*4 SW3(5)
  REAL*8 XH(50),XL(50)
COMMON /NGC4/ SW4,FR,XR,NEW
  LOGICAL*4 SW4(2),NEW
  REAL*8 FR,XR(50)
COMMON /NGC5/ SW5,M,TEQ,MULTS
  LOGICAL*4 SW5(3)
  REAL*8 TEQ,MULTS(50)
END
```

`NBR`	catalog number
`PROBID`	familiar name
`FNABBR`	file name abbreviation
`PKC`	problem kind
`FLAGS`	problem attributes
	convex
	smooth
	constr qualification
`DEFFIL`	problem definition file
`NLPDIR`	problem definition directory
`MI`	number of inequality constraints
`ME`	number of equality constraints
`N`	number of variables
`XH`	upper bounds
`XL`	lower bounds
`FR`	optimal value
`XR`	optimal point
`NEW`	`T` ⇒ `FR` and `XR` were updated
`M`	total number of constraints
`TEQ`	equality constraint tolerance
`MULTS`	KKT multipliers

If `PKC` is needed but missing, `ASKPRB` prompts the user to choose from the `problem kind` menu shown on the left at the top of the next page. If `FLAGS` is needed but missing, the user

[5]The name `NGC` is a grandiose allusion to the *New General Catalogue of Nebulae and Clusters of Stars,* which was compiled by Johan Dreyer in 1888. The `Workbench/NGC` directory contains a collection of test problems, which we can observe through computational experiments rather than through a telescope.

is prompted to choose from the `convex`, `smooth`, and `constr qualification` menus.

```
problem kind:              convex:          smooth:                          constr qualification:
  1  NLP type 1              1  unknown       1  unknown                        1  unknown
  2  NLP type 2              2  convex        2  yes by formula                 2  yes
  3  geometric program       3  concave       3  yes by finite difference       3  no
  4  quadratic program       4  neither       4  no
  5  linear program
```

The scheme described in §26.2 assumes that each test problem definition file will contain routines `FCN`, `GRD`, and `HSN` that use formulas to compute function values, gradients, and Hessians specific to its particular objective and constraints. Problems of kind `1` fit that description, including those in which derivatives must be approximated by finite differencing.

A problem of kind `2` is also defined by `FCN`, `GRD`, and `HSN` routines specific to its objective and constraints, but it uses an algorithm rather than a formula to find at least one function value, gradient vector, or Hessian matrix. In a problem of kind `2` that algorithm might perform a simulation or integrate a differential equation numerically; in the case of a bilevel program it solves an inner optimization. A problem of kind `2` often includes as part of its definition a `SETUP` subroutine that does some sort of preprocessing (such as reading regression data from a file). If a routine of that name is present the `GETEXP` routine calls it before the solution process begins.

Geometric, quadratic, and linear programs are assumed to have the forms described below. GPs are assumed always to have posynomial ≤ 1 constraints, so $m = m_i$ and $m_e = 0$. LPs are assumed to have all inequality constraints or all equality constraints, so that either $m_e = 0$ or $m_i = 0$. Whether nonnegativity of the variables in an LP is assumed or must be explicitly enforced depends on the solver.

$$\begin{aligned}
\text{GP: } \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}}\ f_{m+1}(\mathbf{x}) &= \sum_{t=1}^{T_{m+1}} p_{t,m+1} \exp\left({a_{t,m+1}}^{\top}\mathbf{x}\right) \\
\text{subject to } f_i(\mathbf{x}) &= \sum_{t=1}^{T_i} p_{ti} \exp\left({a_{ti}}^{\top}\mathbf{x}\right) - 1 \leq 0, \quad i = 1\ldots m
\end{aligned}$$

$$\begin{aligned}
\text{QP: } \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}}\ f_{m+1}(\mathbf{x}) &= \mathbf{x}^{\top}\mathbf{A}_{m+1}\mathbf{x} + {\mathbf{b}_{m+1}}^{\top}\mathbf{x} + d_{m+1} \\
\text{subject to } f_i(\mathbf{x}) &= \mathbf{x}^{\top}\mathbf{A}_i\mathbf{x} + {\mathbf{b}_i}^{\top}\mathbf{x} + d_i \leq 0, \quad i = 1\ldots m_i \\
&= \mathbf{x}^{\top}\mathbf{A}_i\mathbf{x} + {\mathbf{b}_i}^{\top}\mathbf{x} + d_i = 0, \quad i = m_i + 1 \ldots m_i + m_e
\end{aligned}$$

$$\begin{aligned}
\text{LP: } \underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}}\ f_{m+1}(\mathbf{x}) &= \mathbf{c}^{\top}\mathbf{x} + d \\
\text{subject to } f_i(\mathbf{x}) &= \mathbf{a}_i\mathbf{x} - b_i \,\square\, 0, \quad i = 1\ldots m
\end{aligned}$$

where $\mathbf{a}_i$ is a row vector and $\square$ is $\leq$ if $m = m_i > 0$ or $=$ if $m = m_e > 0$

The functions in a GP, QP, or LP are the same from one problem to another except for the values of the coefficients appearing in these formulas. Each kind therefore has a single set of

FCN, GRD, and HSN routines, which receive the data they require via a COMMON block named /COEFSG/, /COEFSQ/, or /COEFSL/. The problem descriptors and formula coefficients for a GP, QP, or LP are read from its definition file and placed in COMMON by a routine named GETGP, GETQP, or GETLP, which is invoked from GETPRB. A problem descriptor can be omitted from the definition file (in which case its line is present but blank) or ignored (in which case its line begins with *), so CHKPRB is used, as with a kind 1 or kind 2 problem, to prompt the user for any that are needed but not specified in the file.

The problem definition file for a GP, QP, or LP always begins with 17 lines each containing the value of one of the descriptors listed on the previous page for BLOCK DATA NGC, in order from top to bottom, except for NEW (which is always set to .FALSE. by GETEXP). Line 18 is always blank. The subsequent lines contain the coefficients to be used in the equations above, in a format that depends on the problem kind.

3.3.1 Geometric Programs

The problem definition file for a GP always has a name that ends in .gp and contains data that are arranged as in the example dem1.gp on the next page. This is problem 1B in [Dembo, Ron S., "A Set of Geometric Programming Test Problems and Their Solutions," *Mathematical Programming* 10, 192-213, 1976]. The line numbers at the left are for reference only and are not actually in the file. Lines 11, 12, 14, and 47 are too long to fit on the page, so the picture shows only the first 145 columns of each. The data for each function consist of the number of terms in the sum, a blank line, the coefficients p_{ti}, a blank line, and the matrix of a_{ti}; a few entries are identified in the first constraint to show the pattern. The set of data for each function is separated from that for the one before it by a single blank line (lines 27, 44, and 63).

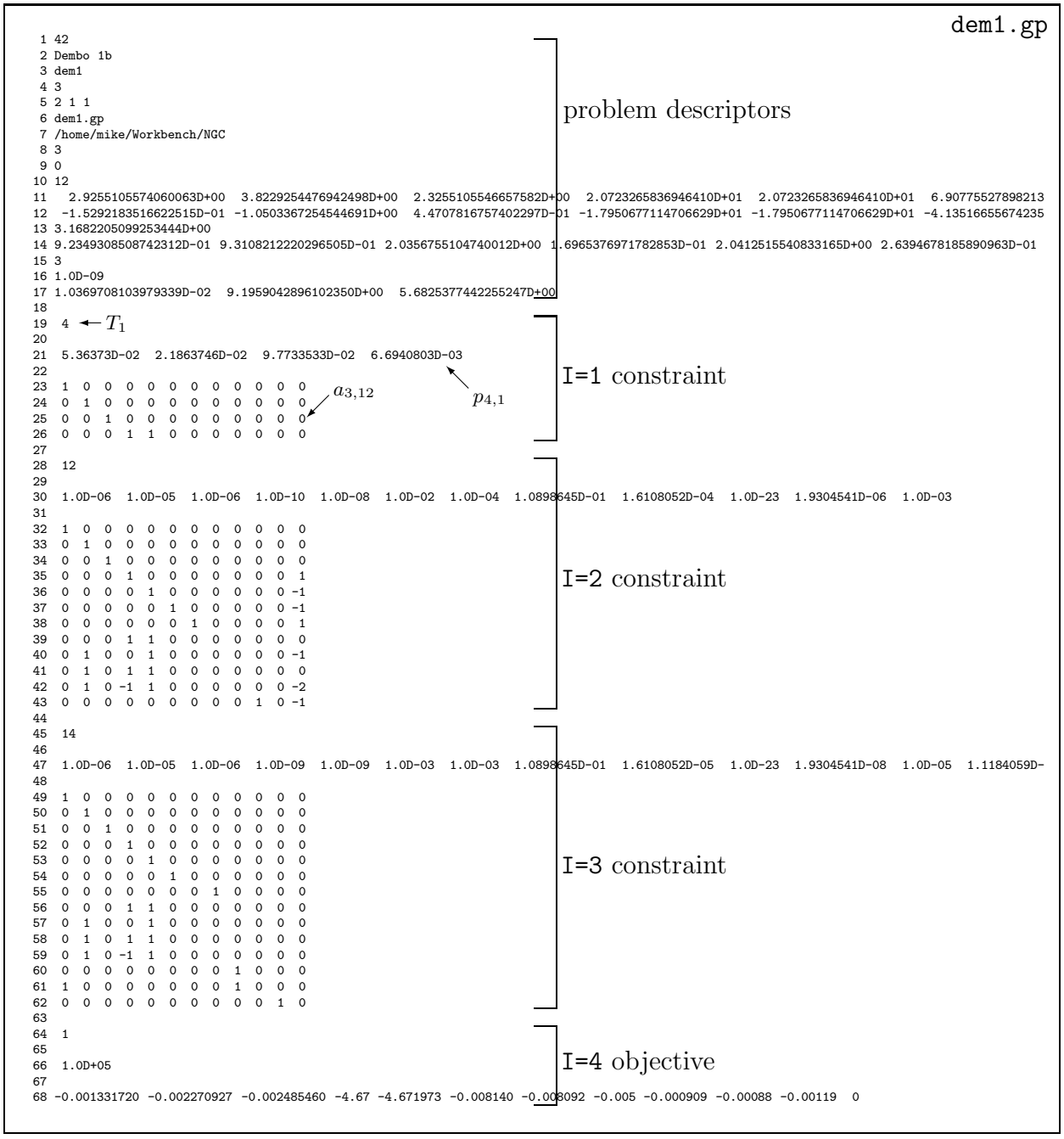
3.3.2 Quadratic Programs

The problem definition file for a QP always has a name that ends in .qp and contains data that are arranged as in the example him24.qp at the top of the page after next. This is problem 24 in [80, p428]. Himmelblau gives the source of the problem as [18, p19] but that problem is quite different from the one that he states; this sort of misattribution is one of the difficulties of computational testing described in §26.2.1. The **A** matrix, **b** vector, and scalar d of the first constraint are identified.

3.3.3 Linear Programs

The problem definition file for an LP always has a name that ends in .lp and contains data that are arranged as in the example at the bottom of the page after next. That problem, cyc.lp, is the degenerate problem discussed at length in §4.5. The format that is used in this file differs from both the standard-form simplex tableau defined in §2.2 and a tableau file that is read or written by the pivot program of §27 (cycle.tab is a tableau file example that you can find in impsrc.tar.gz). The constraints of this problem are equalities, because

`MI=0` (on line `8`) and `ME=3` (on line `9`). If we put the row vectors $\mathbf{c}$ and $\mathbf{a}_i$ into a tableau whose left-hand side consists of the column vector $\mathbf{b}$ and the constant d, we get the initial tableau shown on page 156. Thus the software described here can be used to study the performance of an LP solver, and this problem in particular might be of use for doing that because it causes the simplex method to cycle.




```
 1 42
 2 Dembo 1b
 3 dem1
 4 3
 5 2 1 1
 6 dem1.gp
 7 /home/mike/Workbench/NGC
 8 3
 9 0
10 12
11   2.9255105574060063D+00   3.8229254476942498D+00   2.3255105546657582D+00   2.0723265836946410D+01   2.0723265836946410D+01   6.90775527898213
12  -1.5292183516622515D-01  -1.0503367254544691D+00   4.4707816757402297D-01  -1.7950677114706629D+01  -1.7950677114706629D+01  -4.13516655674235
13 3.1682205099253444D+00
14 9.2349308508742312D-01 9.3108212220296505D-01 2.0356755104740012D+00 1.6965376971782853D-01 2.0412515540833165D+00 2.6394678185890963D-01
15 3
16 1.0D-09
17 1.0369708103979339D-02  9.1959042896102350D+00  5.6825377442255247D+00
18
19  4
20
21  5.36373D-02  2.1863746D-02  9.7733533D-02  6.6940803D-03
22
23  1  0  0  0  0  0  0  0  0  0  0  0
24  0  1  0  0  0  0  0  0  0  0  0  0
25  0  0  1  0  0  0  0  0  0  0  0  0
26  0  0  0  1  1  0  0  0  0  0  0  0
27
28  12
29
30  1.0D-06  1.0D-05  1.0D-06  1.0D-10  1.0D-08  1.0D-02  1.0D-04  1.0898645D-01  1.6108052D-04  1.0D-23  1.9304541D-06  1.0D-03
31
32  1  0  0  0  0  0  0  0  0  0  0  0
33  0  1  0  0  0  0  0  0  0  0  0  0
34  0  0  1  0  0  0  0  0  0  0  0  0
35  0  0  0  1  0  0  0  0  0  0  0  1
36  0  0  0  0  1  0  0  0  0  0  0 -1
37  0  0  0  0  0  1  0  0  0  0  0 -1
38  0  0  0  0  0  0  1  0  0  0  0  1
39  0  0  0  1  1  0  0  0  0  0  0  0
40  0  1  0  0  1  0  0  0  0  0  0 -1
41  0  1  0  1  1  0  0  0  0  0  0  0
42  0  1  0 -1  1  0  0  0  0  0  0 -2
43  0  0  0  0  0  0  0  0  0  1  0 -1
44
45  14
46
47  1.0D-06  1.0D-05  1.0D-06  1.0D-09  1.0D-09  1.0D-03  1.0D-03  1.0898645D-01  1.6108052D-05  1.0D-23  1.9304541D-08  1.0D-05  1.1184059D-
48
49  1  0  0  0  0  0  0  0  0  0  0  0
50  0  1  0  0  0  0  0  0  0  0  0  0
51  0  0  1  0  0  0  0  0  0  0  0  0
52  0  0  0  1  0  0  0  0  0  0  0  0
53  0  0  0  0  1  0  0  0  0  0  0  0
54  0  0  0  0  0  1  0  0  0  0  0  0
55  0  0  0  0  0  0  0  1  0  0  0  0
56  0  0  0  1  1  0  0  0  0  0  0  0
57  0  1  0  0  1  0  0  0  0  0  0  0
58  0  1  0  1  1  0  0  0  0  0  0  0
59  0  1  0 -1  1  0  0  0  0  0  0  0
60  0  0  0  0  0  0  0  0  1  0  0  0
61  1  0  0  0  0  0  0  0  1  0  0  0
62  0  0  0  0  0  0  0  0  0  0  1  0
63
64  1
65
66  1.0D+05
67
68 -0.001331720 -0.002270927 -0.002485460 -4.67 -4.671973 -0.008140 -0.008092 -0.005 -0.000909 -0.00088 -0.00119  0
```

him24.qp

```
 1 44
 2 Himmelblau 24
 3 him24
 4 4
 5 1 1 1
 6 him24.qp
 7 /home/mike/Workbench/NGC
 8 2
 9 0
10 2
11 4 4
12 0 0
13 1
14 1 1
15 2
16 0.D0
17 0.6666666666666666 .6666666666666666
18
19 1 0
20 0 0
21
22 0 -1
23
24 0
25
26 0 0
27 0 0
28
29 1 1
30
31 -2
32
33 1 0
34 0 1
35
36 -4 -2
37
38 5
```

problem descriptors

$\leftarrow \mathbf{A}_1$

$\leftarrow \mathbf{b}_1$

$\leftarrow d_1$

I=1 constraint

I=2 constraint

I=3 objective

cyc.lp

```
 1 14
 2 cycle
 3 cyc
 4 5
 5 1 1 1
 6 cyc.lp
 7 /home/mike/Workbench/NGC
 8 0
 9 3
10 7
11 1 1 1 1 1 1 1
12 0 0 0 0 0 0 0
13 -4.25
14 0.75 0 0 1 0 1 0
15 3
16 0.D0
17 0 1.5 1.25
18
19 1 0 0 0.25  -8 -1   9
20 0 1 0 0.50 -12 -0.5 3
21 0 0 1 0      0  1   0
22
23 0 0 1
24
25 0 0 0 -0.75 20 -0.5 6
26
27 -3
```

problem descriptors

$\leftarrow \mathbf{a}_1$

$\leftarrow \mathbf{a}_2$

$\leftarrow \mathbf{a}_3$

$\leftarrow \mathbf{b}$

$\leftarrow \mathbf{c}$

$\leftarrow d$

3.4 GETEXP

Instrumenting the driver program for EA3 involved the insertion of calls to the Workbench routines GETEXP, LOGPMR, and TIMER, and the tool programs discussed later call GETPRB to obtain problem descriptors. Each of these routines is therefore described in some detail in this Section and the three that follow. Each invokes other Workbench routines that are *not* described in detail, as well as numerous routines from the general-purpose subprogram library in lib.tar.gz. You can find out about all of them by reading their man pages in man.tar.gz and their commented source code.

Here is an outline of the Workbench routine invocations that begin with a call to GETEXP.

```
GETEXP get experiment and problem descriptors
   GETSCR reset some experiment descriptors for running in batch mode
   prompt for and read experiment descriptors needed but not set
   GETPMF attach a performance measurements file if one is needed
      FNANDU generate standard name for performance measurements file
   adjust NEED if writing performance measurements
   GETPRB gets problem descriptors missing but required in NEED
      GETGP read data defining a geometric program
         GETPDF attach problem definition file
            FNANDU generate standard problem definition file name
         GETNGC read problem descriptors
            RDNGC read a test problem descriptor line
            read coefficients
      GETQP read data defining a quadratic program
         GETPDF attach problem definition file
            FNANDU generate standard problem definition file name
         GETNGC read problem descriptors
            RDNGC read a test problem descriptor line
            read coefficients
      GETLP read data defining a linear program
         GETPDF attach problem definition file
            FNANDU generate standard problem definition file name
         GETNGC read problem descriptors
            RDNGC read a test problem descriptor line
            read coefficients
      SETUP execute user-supplied code if present
      CHKPRB check whether any needed problem descriptors are missing
      ASKPRB get any needed problem descriptors that are missing
   prepare to write performance measurement records if desired
   initialize timing if desired
   find the starting point as the midpoint of the bounds
```

GETSCR reads a script file if one is provided, so that the test program can run without user intervention.

GETPMF figures out if a performance measurements file is needed, and opens one if it is.

GETPRB does most of the work and is described in detail in the next Section.

```
 1 C
 2       SUBROUTINE GETEXP(X)
 3 C     This routine gets experiment and problem descriptors and sets
 4 C     up to run a computational experiment.
 5 C
 6 C     variable  meaning
 7 C     --------  -------
 8 C     DATIME    routine returns current "Www dd Mmm yy hh:mm:ss"
 9 C     EXIT      system routine stops program with a return code
10 C     GETI4S    routine prompts for and reads an INTEGER*4
11 C     GETPMF    routine opens a performance measurements file
12 C     GETPRB    routine gets problem identifiers
13 C     GETSCR    routine reads a script file if present
14 C     IDEF      default value for KMAX (=1)
15 C     J         index on variables
16 C     K         index on the characters of PROBID
17 C     KMAX      iteration limit
18 C     KNOW      current iteration count
19 C     LDEF      length of default value for KMAX
20 C     LENGTH    function gives index of last nonblank in a string
21 C     LID       length of PROBID
22 C     LOGPMR    routine writes a performance measurements record
23 C     LONG      long algorithm name; unused
24 C     ME        number of equality constraints; unused
25 C     MI        number of inequality constraints; unused
26 C     MSGLVL    message level; -1 => batch mode
27 C     N         number of variables
28 C     NBIN      number of bin just receiving time; unused
29 C     NBR       catalog number of test problem; unused
30 C     NEED      tells if problem descriptor (item,block) is needed
31 C     NFE       number of function evaluation used so far
32 C     NGE       number of gradient evaluation used so far
33 C     NHE       number of Hessian evaluation used so far
34 C     NONALG    number of bin for non-algorithm time
35 C     NTMEAS    number of timing measurements made so far
36 C     NUNIT     unit number of performance measurements file
37 C     PMF       full name of performance measurements file
38 C     PROBID    problem name
39 C     QUERY     routine asks a yes-or-no question
40 C     RC        return codes; 0 => all went well
41 C     SHIFTL    routine removes leading blanks from a string
42 C     SHORT     short algorithm name
43 C     STAMP     day-date-time stamp for this experiment
44 C     SWp       switches tell what is known in /NGCp/
45 C     TIMER     routine measures processor time
46 C     TIMING    T => timing is enabled
47 C     TOPBIN    highest bin number used so far; unused
48 C     WPMR      T => write performance measurements
49 C     X         initial point returned to driver program
50 C     XH        catalog upper bounds
51 C     XL        catalog lower bounds
52 C     XZERO     starting point shared in COMMON
53 C
54 C     formal parameter
55       REAL*8 X(50)
56 C
```

The preamble of GETEXP is listed on this page and the next. It begins 6-52 with a variable dictionary and 54-55 the declaration of its only formal parameter. Then 57-69 it includes the experiment descriptors that it will set and 71-82 the problem descriptors that it needs. Some of the variables in COMMON are present only for alignment (see [100, §8.3]).

```
C     receive and prepare to set experiment descriptors
      COMMON /EXP1/ SHORT,LONG,STAMP
        CHARACTER*4 SHORT
        CHARACTER*1 LONG(24),STAMP(22)
      COMMON /EXP2/ MSGLVL,KMAX,XZERO
        REAL*8 XZERO(50)
      COMMON /EXP3/ WPMR,PMF
        LOGICAL*4 WPMR
        CHARACTER*48 PMF
      COMMON /EXP4/ NFE,NGE,NHE,KNOW
      COMMON /EXPT/ TIMING,NONALG,NBIN,TOPBIN,NTMEAS
        LOGICAL*4 TIMING
        INTEGER*4 TOPBIN
C
C     request and receive problem descriptors
      INTEGER*4 NEED(5,5)/0,0,0,9,9,
     ;                    1,0,1,1,9,
     ;                    1,1,1,1,1,
     ;                    0,0,9,9,9,
     ;                    0,0,0,9,9/
      COMMON /NGC1/ SW1,NBR,PROBID
      LOGICAL*4 SW1(3)
      CHARACTER*1 PROBID(48)
      COMMON /NGC3/ SW3,MI,ME,N,XH,XL
      LOGICAL*4 SW3(5)
      REAL*8 XH(50),XL(50)
C
C     local variables
      LOGICAL*4 QUERY
      INTEGER*4 RC
C
C -------------------------------------------------------------------
C
```

The numbers appearing in row p of the initialization statement [72-76] for matrix NEED tell which problem descriptors are 1=needed, 0=not needed, or 9=not in /NGCp/. For example, the second row of values in the initialization of NEED, 1,0,1,1,9 says that elements 1, 3, and 4 of /NGC2/ are needed, because PKC, DEFFIL, and NLPDIR are all used somewhere in the code that will make up the test program in which GETEXP is used.

The executable code begins on the next page by [90-96] reading a script file if one is named in the first command parameter. If KMAX is zero the routine assumes that the other experiment descriptors TIMING, WPMR, and MSGLVL are also not set and obtains them from the user. Then [114-126] it makes sure that a performance measurements file is attached if one will be needed. If [128-135] performance measurements will be written then [130-131] PROBID, [132] FNABBR, and [133-134] FR are marked as needed. Next [137-138] GETPRB is called to make sure that the needed problem descriptors are present, including PROBID whose length [139] will be needed later.

If performance measurements are to be written [141-157] the performance measurements file, which is attached to unit 7, is labeled with the name of the problem, the short name of the algorithm, and [149-151] a day-date-time stamp telling when the experiment was run. The counts of function, gradient, and Hessian evaluations are [154-156] set to zero.

Next [159-161] an initialization call is made to TIMER. If TIMING=.FALSE. that routine returns immediately. Setting NTMEAS=-1 first [160] suppresses the processor speed report

```
 90 C     read a script file if one is specified
 91       CALL GETSCR(RC)
 92       IF(RC.EQ.2) THEN
 93          WRITE(0,991)
 94  991     FORMAT('a script file was provided but could not be read')
 95          CALL EXIT(1)
 96       ENDIF
 97 C
 98 C     is an initial interation limit preset in BLOCK DATA EXP?
 99       IF(KMAX.EQ.0) THEN
100 C        no; set up for interactive experimentation
101     1    IF(.NOT.TIMING) THEN
102              TIMING=QUERY('Time the algorithm?',19)
103          ENDIF
104          IF(.NOT.WPMR) THEN
105              WPMR=QUERY('Write performance measurement records?',39)
106          ENDIF
107          IF(TIMING) THEN
108             CALL GETI4S('Message level',13,0,1, MSGLVL,RC)
109          ELSE
110             CALL GETI4S('Message level',13,1,1, MSGLVL,RC)
111          ENDIF
112       ENDIF
113 C
114 C     if a performance measurements file is attached write to it
115 C     if performance records are to be written attach a file
116       CALL GETPMF(RC)
117       IF(RC.EQ.0) THEN
118          INQUIRE(UNIT=7,NAME=PMF)
119          IF(MSGLVL.GE.0) WRITE(0,993) PMF
120   993    FORMAT('GETPMF attached performance measurements file ',
121      ;          A48)
122       ENDIF
123       IF(RC.EQ.2) THEN
124          WRITE(0,992)
125   992    FORMAT('a PMF was requested but could not be opened')
126       ENDIF
127 C
128 C     request needed problem descriptors (item,block)
129       IF(WPMR) THEN
130 C        the name of the algorithm will be in the output file
131          NEED(2,1)=1
132          NEED(3,1)=1
133 C        catalog record value is needed to check for better points
134          NEED(1,4)=1
135       ENDIF
136 C
```

that TIMER normally makes upon initialization. Then 165-167 the starting point XZERO is found as the midpoint of the bounds and 171-173 XZERO is copied into X for return.

The iteration count KNOW is 174 set to zero and LOGPMR is called 175 to write a performance measurement record for the starting point. If WPMR=.FALSE. that routine returns immediately.

If it is desired and possible the name of the problem is 177-182 written to the screen. Finally 184-194 if an iteration limit has not yet been set (in BLOCK DATA EXP or in a script file) the user is prompted for an initial iteration limit.

```
C     get the needed problem descriptors
      CALL GETPRB(NEED)
      LID=LENGTH(PROBID,48)
C
C     set up to write performance measurement records
      IF(WPMR) THEN
C        write preamble information in performance measurements file
         NUNIT=7
         WRITE(NUNIT,901) (PROBID(K),K=1,LID)
  901    FORMAT(48A1)
         WRITE(NUNIT,902) SHORT
  902    FORMAT(A4)
         CALL DATIME(STAMP)
         WRITE(NUNIT,903) STAMP
  903    FORMAT(22A1/' ')
C
C        initialize function, gradient, Hessian evaluation counts
         NFE=0
         NGE=0
         NHE=0
      ENDIF
C
C     initialize timing
      IF(MSGLVL.LT.1) NTMEAS=-1
      CALL TIMER(-1,-1)
C
C     the starting point is the midpoint of the bounds
      CALL TIMER(1,-2)
      DO 2 J=1,N
           XZERO(J)=0.5D0*(XH(J)+XL(J))
    2 CONTINUE
      CALL TIMER(NONALG,-3)
C
C     write the starting performance measurement record
      DO 3 J=1,N
           X(J)=XZERO(J)
    3 CONTINUE
      KNOW=0
      CALL LOGPMR(X,N)
C
C     begin screen output with the problem name
      IF(MSGLVL.GE.1 .AND. SW1(2)) THEN
         WRITE(0,904) (PROBID(K),K=1,LID)
  904    FORMAT(48A1)
      ENDIF
      WRITE(0,904) ' '
C
C     set the starting iteration limit if not preset
      IF(KMAX.GT.0) THEN
         IF(MSGLVL.GE.0) WRITE(0,905) KMAX
  905    FORMAT('initial iteration limit set to',I5)
      ELSE
         IDEF=1
         LDEF=1
         CALL GETI4S('Initial iteration limit',23,
     ;               IDEF,LDEF, KMAX,RC)
         IF(RC.EQ.1 .OR. KMAX.EQ.0) CALL EXIT(0)
      ENDIF
      RETURN
      END
```

3.5 GETPRB

Once GETEXP knows the experiment descriptors to be used, it can figure out what problem descriptors are needed and call GETPRB to ensure that they are present.

```
 1 C
 2 Code by Michael Kupferschmid
 3 C
 4       SUBROUTINE GETPRB(NEED)
 5 C     This routine
 6 C       gets data for a GP, QP, or LP
 7 C       checks problem descriptors for all kinds of problems
 8 C       invokes SETUP, which might be a dummy
 9 C     PKC must be preset in the BLOCK DATA of problem.f
10 C
11 C     variable  meaning
12 C     --------  -------
13 C     ASKPRB    routine prompts for needed descriptors
14 C     CHKPRB    function checks for needed descriptors
15 C     EXIT      system routine stops program with a return code
16 C     GETGP     routine gets descriptors and data for a GP
17 C     GETLP     routine gets descriptors and data for an LP
18 C     GETQP     routine gets descriptors and data for a QP
19 C     MSGLVL    message level; -1 => batch mode
20 C     NEED      tells if problem descriptor (item,block) is needed
21 C     PKC       problem kind code
22 C     RC        return codes; 0 => all went well
23 C     SETUP     preprocessing routine for type-2 problems
24 C     SW2       switches tell what is known in /NGC2/
25 C
26 C     formal parameter passed on
27       INTEGER*4 NEED(5,5)
28 C
29 C     get the problem kind code
30       COMMON /NGC2/ SW2,PKC
31       LOGICAL*4 SW2(4)
32       INTEGER*4 PKC
33 C
34 C     get the message level
35       COMMON /EXP2/ MSGLVL
36 C
37 C     local variables
38       INTEGER*4 RC
39       LOGICAL*4 CHKPRB
40 C
41 C ---------------------------------------------------------------------
42 C
```

The preamble of this routine includes [11-24] a variable dictionary, the declaration [26-27] of its only formal parameter, and COMMON blocks [29-32] containing the problem kind code PKC and [34-35] the message level MSGLVL.

The executable code begins [43-48] by ensuring that the problem kind code PKC has been set in BLOCK DATA NGC. For problem kinds 1 and 2 that is normally done in the problem definition file; if the BLOCK DATA subprogram is missing or if the problem kind is 3, 4, or 5, it is added to the problem definition by the prepare script.

Next [50-55] the other problem descriptors in BLOCK DATA NGC are filled in if the problem is of type [52] 3, [53] 4, or [54] 5. The routine GETGP, GETQP, or GETLP reads the data file that defines the geometric, quadratic, or linear program.

```
43 C     is the problem kind specified?
44       IF(.NOT.SW2(1)) THEN
45          WRITE(0,991)
46   991    FORMAT('PKC must be preset in /NGC2/')
47          GO TO 1
48       ENDIF
49 C
50 C     yes; get remaining descriptors and data
51       RC=0
52       IF(PKC.EQ.3) CALL GETGP(RC)
53       IF(PKC.EQ.4) CALL GETQP(RC)
54       IF(PKC.EQ.5) CALL GETLP(RC)
55       IF(RC.NE.0) GO TO 1
56 C
57 C     is this problem of type 2?
58       IF(PKC.EQ.2) THEN
59          CALL SETUP(RC)
60          IF(RC.EQ.-1) THEN
61 C           this must be the dummy routine
62             IF(MSGLVL.GT.0) WRITE(0,901)
63   901       FORMAT('no user-supplied SETUP routine was found')
64          ELSE
65             IF(MSGLVL.GT.0) WRITE(0,902) RC
66   902       FORMAT('user-supplied SETUP routine returned RC=',I2)
67             IF(RC.NE.0) GO TO 1
68          ENDIF
69       ENDIF
70 C
71 C     are all needed problem descriptors present?
72       IF(CHKPRB(NEED)) RETURN
73       IF(MSGLVL.GE.0) THEN
74          CALL ASKPRB(NEED)
75          RETURN
76       ELSE
77          WRITE(0,992)
78   992    FORMAT('one or more needed problem descriptors is missing')
79          CALL EXIT(1)
80       ENDIF
81 C
82 C     if there was a problem report it
83     1 WRITE(0,993)
84   993 FORMAT('GETPRB failed to define the test problem')
85       CALL EXIT(1)
86       END
```

Then [57-69] if the problem is of kind 2, SETUP is called. This routine is either part of the problem definition or a dummy that returns immediately with RC=-1. If MSGLVL is high enough, messages are written to show which it is, and [67] if it is user-supplied and fails, that is [82-84] reported and [85] the program stops with return code 1.

Otherwise, [71-80] this routine ensures that all of the needed problem descriptors are present. If any NEED(r,p)=1 and the corresponding switch vector element SWp(r)=.FALSE., then CHKPRB returns .FALSE.; in that case ASKPRB is used to prompt the user for the missing data, unless [73] user interaction is not possible. In that case [77-78] an error message is written and [79] the program stops with return code 1.

If GETPRB fails for some other reason, that is reported [83-84] and [85] the program also stops with return code 1

3.6 LOGPMR

To write performance measurement records, an instrumented algorithm code calls LOGPMR.

```
 1 C
 2       SUBROUTINE LOGPMR(X,N)
 3 C     This routine writes one performance measurement record and
 4 C     checks the current point against the best one known so far.
 5 C
 6 C     variable  quantity
 7 C     --------  ---------------------------------------------------
 8 C     BINRPT    routine reports timing bin contents
 9 C     BINSUM    routine computes total algorithm time used so far
10 C     DABS      Fortran function gives |REAL*8|
11 C     EXIT      system routine stops program with a return code
12 C     F         objective function value at X
13 C     FCN       basic (not stub) routine computes function values
14 C     FR        best known record value
15 C     I         index on constraints
16 C     J         index on variables
17 C     KNOW      number of iteration just completed
18 C     KOLD      previous iteration number
19 C     M         total number of constraints; unused
20 C     ME        number of equality constraints
21 C     MI        number of inequality constraints
22 C     MSGLVL    message level
23 C     N         number of variables
24 C     NBIN      index of current timing bin
25 C     NBINSV    timing bin on entry, reset before exit
26 C     NEW       T => FR and XR were updated
27 C     NFE       number of function evaluations used so far
28 C     NGE       number of gradient evaluations used so far
29 C     NHE       number of Hessian evaluations used so far
30 C     NONALG    non-algorithm time bin
31 C     PHASE     1 => infeasible; 2 => feasible
32 C     QUERY     routine asks a yes or no question
33 C     SAME      this iterate is the same as the previous one
34 C     SWx       vector of switches tell what is known in /NGCx/
35 C     TALG      time used by algorithm steps
36 C     TEQ       equality constraint tolerance
37 C     TIMER     timing routine
38 C     TIMING    T => the algorithm is being timed
39 C     TNON      cumulative non-algorithm time; unused
40 C     TOLD      TALG on the previous call
41 C     TTIM      cumulative timing overhead; unused
42 C     WPMR      T => write PMFs; else return immediately
43 C     X         current solution vector
44 C     XOLD      previous point logged
45 C     XR        record point
46 C
47 C     formal parameters
48       REAL*8 X(N)
49 C
```

The preamble of this routine begins with [6-45] a variable dictionary and [47-48] the declaration of X, the current iterate passed in as a formal parameter.

Then it receives [50-56] the experiment descriptors and [58-66] problem descriptors that it needs, and [68-81] declares local variables. KOLD is [69] initialized to the largest negative value, and XOLD is [70] initialized to NaN. Notice that function values are calculated using [73] FCN, *not* FCNEA, so these evaluations are not counted or separately timed.

```
50 C     receive experiment descriptors from common
51       COMMON /EXP2/ MSGLVL
52       COMMON /EXP3/ WPMR
53         LOGICAL*4 WPMR
54       COMMON /EXP4/ NFE,NGE,NHE,KNOW
55       COMMON /EXPT/ TIMING,NONALG,NBIN
56         LOGICAL*4 TIMING
57 C
58 C     share problem in common
59       COMMON /NGC3/ SW3,MI,ME
60         LOGICAL*4 SW3(5)
61       COMMON /NGC4/ SW4,FR,XR,NEW
62         LOGICAL*4 SW4(2),NEW
63         REAL*8 FR,XR(50)
64       COMMON /NGC5/ SW5,M,TEQ
65         LOGICAL*4 SW5(3)
66         REAL*8 TEQ
67 C
68 C     prepare to check for a repeated iterate
69       INTEGER*4 KOLD/Z'80000000'/
70       REAL*8 XOLD(50)/50*Z'7FFFFFFFFFFFFFFF'/
71 C
72 C     prepare to compute the objective and check feasibility
73       REAL*8 F,FCN
74       INTEGER*4 PHASE
75 C
76 C     prepare to check for backwards in effort
77       REAL*8 TALG,TOLD/0.D0/
78 C
79 C     other local variables
80       REAL*8 TNON,TTIM
81       LOGICAL*4 SAME,QUERY
82 C
83 C -------------------------------------------------------------------
84 C
```

The executable code begins [85-86] by testing `WPMR`, and if performance measurements are not to be written the routine returns immediately.

If timing is enabled [88-93] the routine switches the timing bin from whatever it was on entry to bin `NONALG`, so that the effort expended in this routine is accounted to non-algorithm time.

Sometimes in instrumenting a code it is difficult to ensure that `LOGPMR` will always be called only once in each iteration of the solver. There is no point in writing the same record twice, so the routine checks [95-103] whether this iterate is the same as the previous one, and skips writing it if it is.

A performance measurement record includes the current objective value and a `PHASE` variable to indicate whether the point is feasible, so [105-106] the objective is computed and [108-121] the point is tested for feasibility. Each inequality must be ≤ 0 and each equality must be $= 0 \pm$ `TEQ` for `X` to be a `PHASE=2` point; otherwise it is a `PHASE=1` point.

If the algorithm is being timed, the time it has used so far is part of the performance measurement record so [125] it is computed. If the clock-cycle counter wraps it is possible for the measured `TALG` to decrease; in that case [126-136] the user should be made aware that the timing measurements are corrupted and [133] given the opportunity to stop the experiment.

```
 85 C     are performance measurement records to be written?
 86       IF(.NOT.WPMR) RETURN
 87 C
 88 C     is timing enabled?
 89       IF(TIMING) THEN
 90 C        yes; switch to the non-algorithm timing bin
 91          NBINSV=NBIN
 92          IF(NBINSV.NE.NONALG) CALL TIMER(NONALG,-4)
 93       ENDIF
 94 C
 95 C     is this iterate the same as the previous one?
 96       SAME=.TRUE.
 97       IF(KNOW.NE.KOLD) SAME=.FALSE.
 98       KOLD=KNOW
 99       DO 1 J=1,N
100            IF(X(J).NE.XOLD(J)) SAME=.FALSE.
101            XOLD(J)=X(J)
102     1 CONTINUE
103       IF(SAME) GO TO 2
104 C
105 C     compute the objective function value
106       F=FCN(X,N,MI+ME+1)
107 C
108 C     check feasibility of the point
109       PHASE=2
110       IF(MI.EQ.0) GO TO 3
111       DO 4 I=1,MI
112            IF(FCN(X,N,I).LE.0.D0) GO TO 4
113            PHASE=1
114            GO TO 5
115     4 CONTINUE
116     3 IF(ME.EQ.0) GO TO 5
117       DO 6 I=MI+1,MI+ME
118            IF(DABS(FCN(X,N,I)).LE.TEQ) GO TO 6
119            PHASE=1
120            GO TO 5
121     6 CONTINUE
122 C
123 C     compute the algorithm time used so far
124     5 IF(TIMING) THEN
125          CALL BINSUM(TALG,TNON,TTIM)
126 C        algorithm time must not decrease
127          IF(TALG.LT.TOLD) THEN
128             WRITE(0,901) KNOW,TALG,TOLD
129   901       FORMAT('at iteration',I6,
130      ;             ' TALG=',1PD13.6,' < ',1PD13.6,'=TOLD')
131             IF(MSGLVL.GT.0) THEN
132                CALL BINRPT
133                IF(.NOT.QUERY('Continue?',9)) CALL EXIT(1)
134             ENDIF
135          ENDIF
136          TOLD=TALG
137       ELSE
138          TALG=0.D0
139       ENDIF
140 C
```

Next 141-145 the routine writes a performance measurement record to the file that was already attached (by GETPMF) to unit 7.

```
141 C     write out the performance measurements record
142       WRITE(7,902) PHASE,KNOW,TALG,NFE,NGE,NHE,
143      ;             F,N,(X(J),J=1,N)
144   902 FORMAT(I1,1X,I10,1X,1PD23.16,1X,I10,1X,I10,1X,I10,1X,
145      ;             1PD23.16,1X,I2,50(1X,1PD23.16))
146 C
147 C     have we discovered a record point better than the best known?
148       IF(PHASE.EQ.1) GO TO 2
149       IF(.NOT.SW4(1)) GO TO 2
150       IF(F.GE.FR)  GO TO 2
151 C     yes; update FR and XR in /NGC4/
152       FR=F
153       SW4(1)=.TRUE.
154       DO 7 J=1,N
155             XR(J)=X(J)
156     7 CONTINUE
157       SW4(2)=.TRUE.
158       NEW=.TRUE.
159 C
160 C     restore the timing bin that was in use upon entry
161     2 IF(TIMING) THEN
162          IF(NBINSV.NE.NONALG) CALL TIMER(NBINSV,-5)
163       ENDIF
164       RETURN
165       END
```

The current iterate might be better than the best known point, so if it is 148 feasible and 149 a record value is known, the current objective is 150 compared to it. If it is better 151-153 the record value is replaced, 154-157 the record point is replaced, and 158 NEW is set to indicate to other users of those quantities that they have been updated.

Finally, 160-163 the timing bin is restored to what it was on entry and 164 the routine returns.

3.7 TIMER

This routine is invoked by the Workbench routines BINSUM, GETEXP, LOGBIN, and LOGPMR, and earlier we encountered it in the driver program for EA3 and in the stub routines FCNEA and GRDEA. It permits the overhead-corrected measurement of processor time and its partition into bins, as described in [100, §15.1.4], and uses the cycle-counting approach discussed in [100, §18.5.4]. In instrumenting a complicated optimization algorithm for performance measurements it might be necessary to invoke TIMER in many places to exclude convenience code and to find out where the code is spending its time.

The preamble of the routine explains its 5-6 purpose and 8-16 arguments. Then it provides 18-52 a variable dictionary and 54-55 declares the formal parameters.

Next 57-63 it includes the complete COMMON block /EXPT, which shares all of the timing measurement data between the routines that use it. When called to initialize timing this routine sets NONALG, TOH, and SPEED, and on each subsequent call it uses TIMING and NBIN and updates TOPBIN, NTMEAS, and perhaps BINCPU.

```
 1 C
 2 Code by Michael Kupferschmid
 3 C
 4       SUBROUTINE TIMER(TCC,STA)
 5 C     This routine measures the processor time used in executing
 6 C     different parts of a program.
 7 C
 8 C     TCC  action taken
 9 C     ---  ------------
10 C     -1   initialize; reset all bin counts and times to zero
11 C      0   update counts and times without changing bin number
12 C     >0   update counts and switch timing to bin number TCC
13 C
14 C     The parameter STA is used to identify the location of the
15 C     TIMER call in the event that TCC has an illegal value or
16 C     some other error occurs.
17 C
18 C     variable  meaning
19 C     --------  -------
20 C     ABS       Fortran intrinsic for absolute value of INTEGER*8
21 C     BINCPU    matrix of visible [sec,nsec] bin processor times
22 C     BINCYC    vector of processor cycle bin counts
23 C     CPS       number of processor cycles per second
24 C     DELTA     a cycle count increment
25 C     DFLOAT    Fortran function gives REAL*8 for INTEGER*4
26 C     EXIT      system routine stops program with a return code
27 C     GETCYC    routine gets Pentium processor cycle count
28 C     GETTSC    routine gets Pentium processor base speed
29 C     IFIX      Fortran function gives INTEGER*4 for REAL*4
30 C     INITMX    parameter sets MAXBIN
31 C     INITOH    initial value for TOH in cycles
32 C     L         index on the bins
33 C     MAXBIN    highest permissible bin number
34 C     NBIN      number of bin that has just been receiving time
35 C     NCYC      a number of cycles
36 C     NONALG    number of bin for non-algorithm time
37 C     NSEC      a number of nanoseconds
38 C     NSPCYC    nanoseconds per cycle
39 C     NTMEAS    number of timing measurements made so far
40 C     OLDTCC    previous value of TCC
41 C     SECCYC    the number of cycles in SECS seconds
42 C     SECS      a number of seconds
43 C     SNGL      Fortran function gives REAL*4 for REAL*8
44 C     SPEED     processor speed in MHz
45 C     STA       number designating CALL statement in source code
46 C     TCC       timer control code; see table above
47 C     TE        cycle count upon entry to this routine
48 C     TIMING    T => timing is enabled
49 C     TL        cycle count at previous entry to this routine
50 C     TOH       overhead correction in cycles
51 C     TOPBIN    highest algorithm bin number used so far
52 C     TX        cycle count upon exit from this routine
53 C
54 C     formal parameters
55       INTEGER*4 TCC,STA
56 C
57 C     communicate timing information in common
58       PARAMETER(INITMX=22)
59       COMMON /EXPT/ TIMING,NONALG,NBIN,TOPBIN,NTMEAS,TOH,BINCPU,
60      ;              SPEED
61       LOGICAL*4 TIMING
62       INTEGER*4 TOPBIN,TOH,BINCPU(2,INITMX)
63       REAL*8 SPEED
64 C
```

```
 65 C     cycle counts and increment
 66       INTEGER*8 TE/0/,TL,TX/0/,DELTA
 67       INTEGER*8 BINCYC(INITMX),NCYC,SECCYC,ABS
 68 C
 69 C     other local variables
 70       REAL*8 CPS,NSPCYC
 71       INTEGER*4 OLDTCC/-2/,INITOH/9/,SECS,NSEC
 72 C
 73 C ---------------------------------------------------------------------
 74 C
 75 C     do nothing unless timing is enabled
 76       IF(.NOT.TIMING) RETURN
 77 C
 78 C     remember the cycle count on the previous entry
 79       TL=TE
 80 C
 81 C     get the cycle count on this entry
 82       CALL GETCYC(TE)
 83       TE=ABS(TE)
 84 C
 85 C     TCC should be in the range -1,0,1...MAXBIN
 86       IF(TCC.LT.-1 .OR. TCC.GT.MAXBIN) GO TO 1
 87 C
 88 C     a nonzero TCC cannot repeat
 89       IF(TCC.NE.0) THEN
 90          IF(TCC.EQ.OLDTCC) GO TO 1
 91       ENDIF
 92 C
 93 C     an update cannot follow an initialization
 94       IF(OLDTCC.EQ.-1) THEN
 95          IF(TCC.EQ.0) GO TO 1
 96       ENDIF
 97 C
 98 C     the first call must be to initialize
 99       IF(OLDTCC.EQ.-2) THEN
100          IF(TCC.NE.-1) GO TO 1
101       ENDIF
102 C
103 C     a subsequent call might be to initialize
104       IF(TCC.EQ.-1) THEN
105 C        reset counts and times
106          MAXBIN=INITMX
107          NONALG=MAXBIN-1
108          NBIN=NONALG
109          TOPBIN=0
110          TOH=INITOH
111          CALL GETTSC(SPEED)
112          IF(NTMEAS.GE.0) WRITE(0,901) SPEED
113   901    FORMAT('TIMER found processor base speed ',F8.3,' MHz.')
114          CPS=SPEED*1.D+06
115          NSPCYC=1000.D0/SPEED
116          DO 2 L=1,MAXBIN
117               BINCYC(L)=0
118               BINCPU(1,L)=0
119               BINCPU(2,L)=0
120     2    CONTINUE
121          OLDTCC=-1
122          NTMEAS=0
123          GO TO 3
124       ENDIF
125 C
126 C     keep track of the highest algorithm bin index used so far
127       IF(NBIN.LT.NONALG) THEN
128          IF(NBIN.GT.TOPBIN) TOPBIN=NBIN
129       ENDIF
```

```
130 C
131 C     add processor cycles that have just gone into bin NBIN
132       DELTA=TE-TX-TOH
133       IF(DELTA.LT.0) DELTA=0
134       BINCYC(NBIN)=BINCYC(NBIN)+DELTA
135       NTMEAS=NTMEAS+1
136 C
137 C     add processor cycles used by TIMER during the previous call
138       DELTA=TX-TL+TOH
139       IF(DELTA.LT.0) DELTA=0
140       BINCYC(MAXBIN)=BINCYC(MAXBIN)+DELTA
141 C
142 C     is this an update call?
143       IF(TCC.EQ.0) THEN
144 C        yes; convert cycle counts into times
145          L=0
146     4    L=L+1
147          IF(L.EQ.TOPBIN+1) L=NONALG
148             NCYC=BINCYC(L)
149             SECS=IFIX(SNGL(DFLOAT(NCYC)/CPS))
150             BINCPU(1,L)=SECS
151 C
152 C            explicitly fixing this product can overflow a fullword
153 C            so convert to long integer implicitly by assignment
154             SECCYC=CPS*DFLOAT(SECS)
155 C
156             NCYC=NCYC-SECCYC
157             NSEC=IFIX(0.5+SNGL(NSPCYC*DFLOAT(NCYC)))
158             BINCPU(2,L)=NSEC
159          IF(L.LT.MAXBIN) GO TO 4
160       ELSE
161 C        no; update the bin number
162          NBIN=TCC
163          OLDTCC=TCC
164       ENDIF
165 C
166 C     get cycle count upon returning to the instrumented code
167     3 CALL GETCYC(TX)
168       TX=ABS(TX)
169       RETURN
170 C
171 C     report mistakes
172     1 IF(STA.EQ.0) WRITE(0,991)
173   991 FORMAT('A call to TIMER')
174       IF(STA.NE.0) WRITE(0,992) STA
175   992 FORMAT('A call to TIMER at station',I10)
176       WRITE(0,993) TCC,OLDTCC
177   993 FORMAT('gives the illegal control code',I10/
178      ;       'the previous control code was ',I10/)
179       CALL EXIT(1)
180       END
```

The local variables 65-71 include several that are declared `INTEGER*8` to match the 64-bit integer cycle count returned by `GETCYC`, *which assumes that the processor is an Intel Pentium.* If 75-76 `TIMING` is not `.TRUE.` on entry, this routine returns immediately.

The processor cycles used between calls to `TIMER` (which we are trying to measure) and the cycles used by `TIMER` itself are computed using the cycle count `TL` on entry to the routine at the previous call, the count `TE` on entry to the routine at this call, and the count `TX` on exit from the routine on the previous call. These quantities are obtained 78-79,81-83 166-168 from `GETCYC` calls that bracket the body of this routine. Each value returned by `GETCYC` is

an unsigned integer; if its sign bit is 1 that must be [83] [168] removed so that the differences between these values will come out right (except in the unlikely event that the counter wraps between calls to TIMER).

Once the entry cycle count has been found, several sanity checks [85-101] are performed on the parameter TCC to guard against typical coding errors in the use of the routine. In the event that one is detected [171-178] an error message is written and [179] the program stops with return code 1. If the station number STA of the TIMER call is nonzero, it is included in the error message as an aid to debugging. By convention negative station numbers are for the Workbench routines that call TIMER, single-digit positive stations are for calls in the algorithm driver program and stubs, and higher numbers identify calls embedded within the code of the algorithm.

station number	location of TIMER call
-1,-2,-3	getexp.f
-4,-5	logpmr.f
-6,-7,-8	binsum.f
-9,-10,-11	logbin.f
1,2,3...9	driver and stubs
10...	algorithm code

If the call is to initialize timing [103-124], MAXBIN is set to the highest possible bin number, NONALG is set to the bin number before that, NBIN is set to NONALG so that when TIMER returns time will begin accumulating in that bin, TOPBIN is set to indicate that no other bin has had time flowing into it yet, and GETTSC is called to find the processor speed. The gettsc.c routine measures the speed in MHz, and this quantity is used to compute the number of processor cycles per second CPS and the number of nanoseconds per processor cycle NSPCYC. Then [116-120] the cycle counts and processor time estimates are initialized, [121] the value of TCC at this call is remembered in OLDTCC, [122] the number of timing measurements NTMEAS is initialized, and [123] the routine exits.

If the call is not to initialize, [126-129] TOPBIN is updated to the highest bin number used so far.

The next two stanzas increment the cycle counts in [131-135] BINCYC(NBIN) and [137-140] in BINCYC(MAXBIN), which measures timing overhead. The time that was spent out in the instrumented code between the previous TIMER call and this one is TE-TX-TOH. The time that was spent in TIMER during its previous call is TX-TL+TOH. In the unlikely event that either of these quantities is negative because the cycle counter wrapped, it is [133] [139] set to zero. The updating of the cycle count for bin NBIN is a timing measurement, so [135] NTMEAS is incremented.

The timing overhead correction TOH is the experimentally determined cycle count that elapses from beginning the entry to TIMER until the cycle count is acquired in the first call [82] to GETCYC, plus the cycle count that elapses between the acquisition of the cycle count in the last call [167] to GETCYC and the completion of the return from TIMER.

Only if TCC=0, indicating an update call, are the cycle counts stored in BINCYC converted into processor times in BINCPU. (LOGBIN and BINSUM make an update call to TIMER for that

purpose.) The loop 144-159 steps through the bins, skipping those between TOPBIN and NONALG. For each bin L, the cycle count NCYC is 148 extracted from BINCYC(L) and 149-158 converted to a [sec,nsec] two-part value (see [100, §18.4]) in BINCPU. If TCC is not zero 161-163 NBIN and OLDTCC are updated to its value.

A program that executes in a Unix environment on a modern processor might be automatically switched between threads running on different cores at changing speeds, and then many of the simplifying assumptions on which TIMER is based will not be realized. To obtain *reproducible* results in an algorithm comparison that is based on processor time it might be necessary to ensure that as far as possible the machine is otherwise empty and idle, and to adopt other measures such as those suggested in [100, §18.5.4]. In other studies of algorithm behavior, results that are *realistic* might be more interesting (an example is shown below in the Section on perfplot).

4 Program Assembly

To build the executable for a test program in §26.3.4 and §26.4.1 we invoked a hypothetical FORTRAN compiler named ftn to compile the algorithm and its driver program along with the definition of a test problem and link the resulting object modules together, producing an executable program named a.out. The shell program listed at the end of §26.4.1 repeats the process for 3 driver programs and 20 test problems, to produce 20 error-versus-effort graphs each containing three curves, one for each solver. This requires that each algorithm code and its driver program be compiled 20 times.

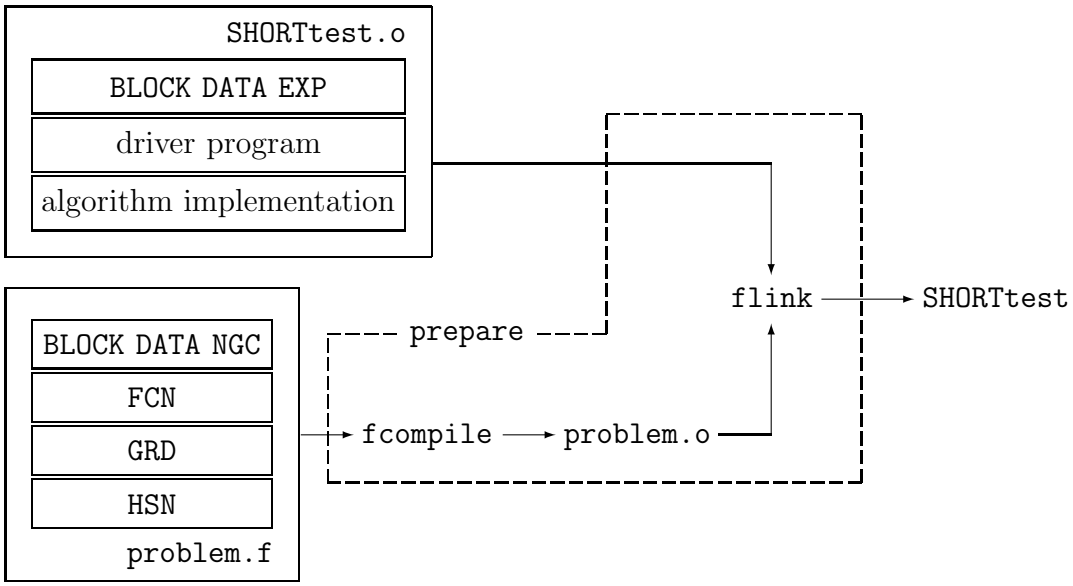
4.1 Compilation and Linking

In practice it is more convenient to compile each algorithm implementation with its driver program just once, and save the resulting partially-linked .o file [100, §1.2 §14.4-14.5]. I name this file SHORTtest.o, where SHORT is the short name of the algorithm. A problem definition can be compiled in a similar way to produce a .o file. Usually there are many more problems than algorithms so it is cumbersome to save all of their .o files, but most problem definitions are simple and compile very quickly so not much effort is wasted by reconstructing problem.o for each experiment. The shell program flink can be used to link each problem.o with SHORTtest.o, producing an executable named SHORTtest.

In several examples above we used the prepare shell program to construct an executable named ea3test. In general its invocation sequence is

```
unix[9] prepare FNABBR[.f|.gp|.qp|.lp] program[.o]
```

where FNABBR is the filename abbreviation for the test problem and program.o is the partially-linked executable containing the object code for the algorithm and its driver program. The compilation and linking process performed by prepare is pictured on the next page. The need to accommodate test problems that are not of kind 1 results in several complications to this simple picture, all of which are also addressed by the prepare shell program.



- If the problem kind is `1` or `2`, then `problem.f` is a copy of the source code in `FNABBR.f`, but if the problem kind is `3`, `4`, or `5` then `problem.f` must contain the source code for the appropriate set of `FCN`, `GRD`, and `HSN` routines for a GP, QP, or LP.
- The `GETPRB` routine described earlier, which is invoked when the driver program calls `GETEXP`, uses the problem kind code `PKC` from `/NGC1/` to decide whether to call `GETGP`, `GETQP`, or `GETLP`, but if the problem is of kind `3`, `4`, or `5` that descriptor is data in the problem definition file to be read by one of those routines. Before `problem.f` is compiled it must therefore be provided with a `BLOCK DATA NGC` routine setting the correct value of `PKC` and the name of the problem definition data file to be read.
- Because `GETPRB` references `SETUP`, a routine of that name must be present in order for `flink` to complete the link step. Only when a problem is of kind `2` might its definition file include a `SETUP` subroutine, so in other cases a dummy must be provided.

4.2 The `prepare` Shell Program

The first part of `prepare` is listed on the next page. As always the numbers appearing at the left of each line are for reference and are not part of the code. In the shell program `$1` is the filename abbreviation of the problem, possibly with an extension denoting its kind, and `$2` is the name of the partially-linked executable to be completed, possibly with a `.o` extension.

The first stanza `4-22` sets the shell variables `NLPDIR`, `PGMDIR`, and `SRCDIR` if they are not already set. These contain the pathnames of directories used later. `NLPDIR`, the directory containing test problem definitions, is `6-11` set to a directory under `$HOME` having the name `NGC` (for me this is `/home/mike/Workbench/NGC`), or to the current directory if no directory having that name is found.

```
 1 #! /bin/sh
 2 # Build code for a test problem into an algorithm or tool program.
 3
 4 if [ -z "$NLPDIR" ]
 5 then
 6 # set the directory containing test problems
 7   NLPDIR=‘/usr/bin/find $HOME -xdev -name NGC -type d 2> /dev/null‘
 8   if [ -z "NLPDIR" ]
 9   then
10     NLPDIR=‘pwd‘
11   fi
12 fi
13 if [ -z "$PGMDIR" ]
14 then
15 # set the directory containing partially-linked programs
16   PGMDIR=$HOME/bin/obj
17 fi
18 if [ -z "$SRCDIR" ]
19 then
20 # set the directory containing source components
21   SRCDIR=$HOME/src/lib
22 fi
23
24 # check that the requested program exists
25 rc=1
26 if [ $# -eq 2 ]
27 then
28   file $PGMDIR/$2.o 2> /dev/null | grep ELF.*relocatable > /dev/null
29   rc=$?
30 fi
31 if [ $rc -ne 0 ]
32 then
33   echo "usage: prepare problem[.f|.gp|.qp|.lp] program"
34   echo "       where program is one of the following:"
35   for pgm in $PGMDIR/*.o
36   do
37     basename $pgm | sed -e "s/.o$//" | sed -e"s/^/       /"
38   done
39   exit 1
40 fi
41
42 # make sure the executable doesn't wind up in the user's bin
43 forbid=$HOME/bin
44 if [ "‘pwd‘" = "$forbid" ]
45 then
46   echo "can't prepare in $forbid"
47   exit 3
48 fi
49
```

The second stanza 24-40 checks that the specified test program has a `.o` file in `$PGMDIR`; if none is specified or if the named file is not a partially-linked executable 33-38 usage instructions are printed and 39 `prepare` stops with return code `1`.

The third stanza 42-48 is present to guard against the common user mistake of preparing an executable in the `bin` directory. If that is the current directory, an error message is written and the program stops with return code `3`. The `prepare` program assumes that it will be used in some directory that is dedicated to the computational testing project, and that the user's `$PATH` to executables ends in `;.` to include the current directory [96, p248-249].

```
50 echo "using problem directory $NLPDIR" #-----------------------------------
51
52 # identify a unique filename defining the problem
53 echo $1 | egrep "\.f$|\.gp$|\.qp$|\.lp$" > /dev/null
54 if [ $? -eq 0 ]
55 then
56    if [ -s $NLPDIR/$1 ]
57    then
58       kind=‘echo $1 | fnext‘
59       prob=‘basename $1 $kind | sed -e"s/\.$//"‘
60    else
61       echo "no definition file found for problem $1 in $NLPDIR"
62       exit 1
63    fi
64 else
65    hits=‘/bin/ls $NLPDIR/$1.* 2> /dev/null‘
66    nhits=‘echo $hits | wc -w‘
67    if [ $nhits -eq 1 ]
68    then
69       kind=‘basename $hits | fnext‘
70       prob=$1
71    fi
72    if [ $nhits -eq 0 ]
73    then
74       echo "no definition file found for problem $1"
75       exit 1
76    fi
77    if [ $nhits -gt 1 ]
78    then
79       echo "Which one?"
80       echo $hits | tr ’ ’ ’\n’
81       exit 1
82    fi
83 fi
84 fyle="$NLPDIR/$prob.$kind"
85 if [ ! -s $fyle ]
86 then
87    echo "problem definition file $fyle is empty."
88    exit 1
89 fi
90
```

The next segment of the program 50-90 finds the full pathname of the problem definition file. If 53-54 the first parameter ends with one of the recognized extensions the program assumes that is the name of the problem definition file and 56-63 searches for it in `$NLPDIR`.

If 56 it finds a nonempty file having that name, it 58 uses `fnext` to extract its extension into `kind` and 59 constructs the problem name `prob`. For example, if `$1=dem1.gp` then `kind=gp` and `prob=dem1`. If 60 the precise filename in `$1` is not found in `$NLPDIR` then 61 an error message is written and 62 the program stops with return code `1`.

If 64 a recognized extension is not found on `$1`, 65 `$NLPDIR` is searched for filenames beginning with it and those files are 66 counted. If exactly one file was found then 67-71 its kind and the problem name are determined. If no file was found then 73-76 an error message is written and the program stops with return code `1`. If multiple files have names beginning `$1`, the program 79-81 asks `Which one?` and reports the possibilities before stopping with return code `1`.

```
 91 # copy the code for FCN, GRD, and HSN into the local problem definition
 92 rm -f problem.f
 93 if [ "$kind" = "f" ]
 94 then
 95    cp $fyle problem.f
 96    cat problem.f | grep -v ^C | grep "SUBROUTINE SETUP" > /dev/null
 97    if [ $? -eq 0 ]
 98    then
 99       echo "found a SETUP routine"
100       pkc=2
101    else
102       pkc=1
103    fi
104 fi
105 if [ "$kind" = "gp" ]
106 then
107    pkc=3
108    cat $SRCDIR/fcngp.f > problem.f
109    cat $SRCDIR/grdgp.f >> problem.f
110    cat $SRCDIR/hsngp.f >> problem.f
111 fi
112 if [ "$kind" = "qp" ]
113 then
114    pkc=4
115    cat $SRCDIR/fcnqp.f > problem.f
116    cat $SRCDIR/grdqp.f >> problem.f
117    cat $SRCDIR/hsnqp.f >> problem.f
118 fi
119 if [ "$kind" = "lp" ]
120 then
121    pkc=5
122    cat $SRCDIR/fcnlp.f > problem.f
123    cat $SRCDIR/grdlp.f >> problem.f
124 fi
125 chmod +w problem.f
126
127 # provide a dummy SETUP routine if none is already present
128 if [ $pkc -ne 2 ]
129 then
130    echo "      SUBROUTINE SETUP(RC)" >> problem.f
131    echo "      INTEGER*4 RC" >> problem.f
132    echo "      RC=-1" >> problem.f
133    echo "      RETURN" >> problem.f
134    echo "      END" >> problem.f
135 fi
136
```

When the base filename `prob` and its extension `kind` are known [84] the full pathname `fyle` is constructed. If [85-89] the file is empty an error message is written and the program stops with return code 1.

Next [91-135] the test problem definition file `problem.f` is constructed. If `fyle` ends in `.f` it is copied into `problem.f` and searched for a `SETUP` subroutine. If one is found [99-100] a message is written to that effect and the problem kind code `pkc` is set to 2; otherwise the problem is assumed [102] to be of kind 1. If the problem kind determined earlier is 3 or 4 or 5 [105-124] routines appropriate to its kind are copied into `problem.f` instead. In case `fyle` is read-only, `problem.f` is [125] permitted to write.

```
137 # provide a skeleton BLOCK DATA routine if none is already present
138 cat problem.f | grep -v ^C | grep "BLOCK DATA NGC" > /dev/null
139 if [ $? -ne 0 ]
140 then
141    echo "      BLOCK DATA NGC" >> problem.f
142    echo "      COMMON /NGC1/ SW1,NBR,PROBID,FNABBR" >> problem.f
143    echo "        LOGICAL*4 SW1(3)/.FALSE.,.FALSE.,.TRUE./" >> problem.f
144    echo "        CHARACTER*8 PROBID(6),FNABBR/'$prob'/" >> problem.f
145    echo "      COMMON /NGC2/ SW2,PKC,FLAGS,DEFFIL,NLPDIR" >> problem.f
146    echo "        LOGICAL*4 SW2(4)/.TRUE.,.FALSE.,.TRUE.,.TRUE./" >> problem.f
147    echo "        INTEGER*4 PKC/$pkc/" >> problem.f
148    echo "        INTEGER*4 FLAGS(3)" >> problem.f
149    echo "        CHARACTER*48 DEFFIL/'$prob.$kind'/" >> problem.f
150    echo "        CHARACTER*48 NLPDIR/'$NLPDIR'/" >> problem.f
151    echo "      COMMON /NGC4/ SW4,FRBN,XRBN" >> problem.f
152    echo "        LOGICAL*4 SW4(2)/.FALSE.,.FALSE./" >> problem.f
153    echo "        REAL*8 FRBN/Z'7FF0000000000000'/" >> problem.f
154    echo "        REAL*8 XRBN(50)/50*Z'7FFFFFFFFFFFFFFF'/" >> problem.f
155    echo "      END" >> problem.f
156 fi
157
158 echo "using program directory $PGMDIR" #-----------------------------------
159
160 # compile the problem definition
161 rm -f /tmp/warnings
162 fcompile problem.f > /tmp/warnings 2>&1
163 if [ $? -ne 0 ]
164 then
165   echo "problem definition compilation failed"
166   more /tmp/warnings
167   exit 3
168 fi
169
170 # remove any previous executable in case linking the new one fails
171 /bin/rm -rf $2
172
173 # link the problem definition to the program
174 rm -f /tmp/warnings
175 flink $PGMDIR/$2.o problem.o > /tmp/warnings 2>&1
176 if [ $? -eq 0 ]
177 then
178    mv a.out $2
179    echo "linking of $2 succeeded"
180 else
181    echo "program linking failed"
182    more /tmp/warnings
183    exit 2
184 fi
185 exit 0
```

If the problem is not of kind 2 it lacks a SETUP subroutine so 129-135 a dummy routine is written to the end of problem.f. The only formal parameter of SETUP is RC, so a user-supplied routine must do whatever it does only by returning a REAL*8 scalar in its name and manipulating data in COMMON storage.

If 138-139 problem.f lacks a BLOCK DATA NGC, either because the problem is type 3, 4, or 5 or because fyle does not contain one, a mostly-empty routine is 141-155 appended to problem.f. In the skeleton BLOCK DATA NGC several variables are initialized to values that

we have determined and the corresponding switch elements are set to .TRUE.; the other variables are left uninitialized or given sensible default values. Now problem.f is complete and it can be compiled [160-168]. The shell program fcompile invokes gfortran with options appropriate to generating a .o file. If the compilation fails [163] then [165-166] the error output from fcompile is copied to the screen and [167] the program stops with return code 3.

Next the target executable named by $2 is removed if one is present, so that if the link fails the user will not be mislead by its presence. Finally [173-184] flink is used to link together program.o and problem.o, producing the executable a.out. The shell program flink invokes gfortran with options appropriate to combining .o files. If the link succeeds [177-179] a.out is given the name in $2 and the program announces that it was successful. If the link failed [181-183] a message is written to that effect, the errors reported by flink are copied to the screen, and the program stops with return code 2. If the exit 2 was not taken [185] the program stops with return code 0.

5 Tools

This Section describes some of the software tools that I have developed to facilitate the definition of test problems and to analyze the results of computational experiments. To link the executable for a tool program to a test problem, I use the prepare script. This works just like linking the executable of an algorithm driver to a test problem.

An algorithm driver calls GETEXP, which enables the user to set MSGLVL, interactively or by means of a script file, for use by GETPRB. The tool programs call GETPRB directly, so each sets MSGLVL=2 to permit user interactions. If perfplot is run with standard-in not attached to the keyboard it assumes that it is being used in batch mode and sets MSGLVL=-1 instead.

5.1 Selecting Fair Bounds: fair

The software described here rigidly enforces the convention that the starting point for solving a test problem is the midpoint of its catalog bounds, so every variable must have declared upper and lower bounds and the only way to change the starting point is to change them. Variable bounds that are to be enforced by a solver must be coded as explicit constraints, and might be different from the catalog bounds. In addition to determining the starting point for an algorithm to use in solving a problem, its catalog bounds are often useful for the other purposes described in §9.5.

If the variable bounds for a problem are left unset (SW3(4) or SW3(5) is .FALSE. in /NGC3/ or a bounds line in the .gp, .qp, or .lp file is blank or has a * in column 1) then the user can try different bounds by specifying them when an experiment is run interactively. To avoid bias in a comparison of algorithms, however, it is essential that once they are fixed the bounds for each problem are cataloged as part of its definition and used for every performance measurement run. That is why the script file described earlier, which permits a test program to be run in batch mode, does *not* provide for setting the bounds or starting point. Versions of a problem that specify different variable bounds are different problems.

The test problem identified above as `ek1` comes from §24.2 and these variable bounds are given for it in §24.3.1.

$$\mathbf{x}^{\mathrm{H}} = \left[18 + 9\sqrt{2},\ 21 + 13\sqrt{2}\right]^{\top} \text{ and } \mathbf{x}^{\mathrm{L}} = \left[18 - 9\sqrt{2},\ 21 - 13\sqrt{2}\right]^{\top}$$

The primitive `BLOCK DATA` subprogram shown in §26.3.5 and the standard `BLOCK DATA NGC` described above and used in `Workbench/NGC/ek1.f` both set `XH` and `XL` to these values.

`XH(1)`	=	$18 + 9/\sqrt{2}$	=	`24.3639610306789 3D0`
`XH(2)`	=	$21 + 13/\sqrt{2}$	=	`30.1923881554251 2D0`
`XL(1)`	=	$18 - 9/\sqrt{2}$	=	`11.6360389693210 7D0`
`XL(2)`	=	$21 - 13/\sqrt{2}$	=	`11.8076118445748 8D0`

Unfortunately, not all problem statements explicitly prescribe bounds on the variables. Test problems come from various sources which differ in the information they provide about variable limits and starting points, and many problems include inequality constraints from which some variable limits can be deduced. An algorithm is described in §26.2.2 for using this haphazard information to select **fair bounds**, which have the starting point at their center and several other desirable properties. This Section describes a program named `fair` which implements that algorithm.[6]

A practical tool for this calculation must address several issues that are not evident from the discussion in §26.2.2.

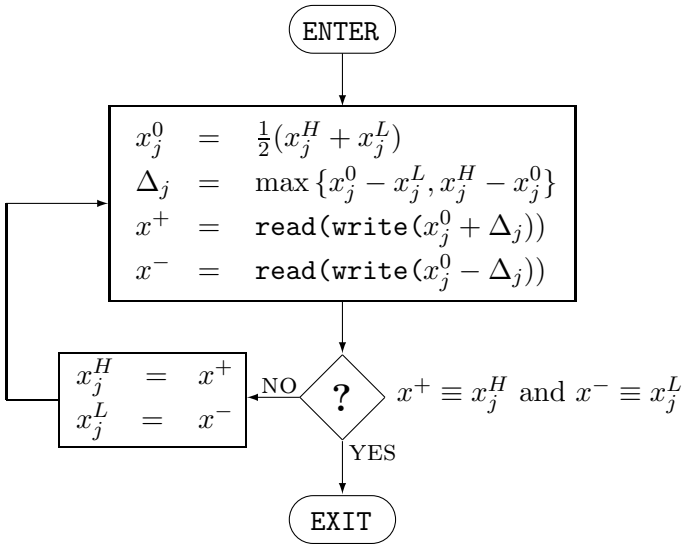
- The bounds that are provided in the problem definition file for a geometric program must be in terms of the $\mathbf{x}$ variables, and the routines `fcngp.f`, `grdgp.f`, and `hsngp.f` use the form of the problem that was introduced in the Section on Problem Descriptors above. However, geometric programs are often stated in terms of so-called natural variables $t_j = \exp(x_j)$ rather than the x_j, so the program permits derived or declared variable limits, the declared starting point, and the optimal point to be input as either t_j or x_j values.

- The table of formulas in §26.2.2 assumes that $\mathbf{x}^\star$ is known, but in the early stages of studying a problem it might not be. Because it is necessary to set variable bounds for experimenting with a problem, the `fair` program recommends provisional bounds to try even if `XR` is not yet known.

- It is convenient in specifying the bounds in a `BLOCK DATA NGC` subprogram or a data file to use decimal numbers, and for the decimal starting point we report to have its internal representation be exactly the midpoint of the internal representations of those bounds. Because of the way in which floating-point numbers are stored that seldom happens by accident, so the `fair` program insists that it happen by using the iterative process pictured at the top of the next page to make minute adjustments to the bounds (and hence to the starting point) in each coordinate direction $j = 1 \ldots n$. In the flowchart

[6] It solves Exercise 26.6.21b.

`write(•)` denotes the result (a decimal number) of writing its argument as characters and `read(•)` denotes the result (a floating-point number) of reading the characters.[7]



In the unlikely event that these adjustments leave an $x_j^\star$ on the wrong side of a bound, that bound is displaced by Δ_j to include it, and the adjustment process is repeated.

- Sometimes derived variable limits are numbers with many decimal points, which makes it tedious to enter them in response to prompts. The `fair` program therefore permits those responses to be read from a file instead of from the keyboard.

The terminal session on the following pages illustrates the use of `fair` to find catalog bounds for the `dem1` test problem. Now there are two files in `Workbench/NGC` whose names begin `dem1`, so the prepare command `[11]` must specify which one to use. In `dem1.gp` I made the `XH` and `XL` lines blank, so the program finds no bounds already present.

The program begins by printing the problem name and prompting for whether coordinates will be provided in terms of **x** or **t**, and my answer `no` says that I will enter t_j values. If we had been computing bounds for a problem of some other kind this prompt would not appear and the program would assume that all coordinates are in **x**.

Then it prompts for a starting point. The paper where this problem is stated (see the Section on Geometric Programs above) specifies a starting point, so I answer `yes` and enter the point at the prompt for `t0`. The problem statement also includes an approximation of its optimal point, so I elect to use an optimal point in finding the bounds. The problem definition file `dem1.gp` already includes an approximation to the optimal point (in **x** variables) that is slightly better than the one given in the paper, so I accept that point as $\mathbf{x}^\star$.

Next the program prompts for any variable limits that might have been deduced[8] from inequality constraints. Several of these have many digits so I used an editor to type them, in the order that they are prompted for, into the file `dem1.bds`.

[7] These operations are often unnecessary; see the `man` page for `TONBAK`.

[8] Eric Johnson analyzed the constraints of Dembo 1b to find the t_j limits they imply.

```
unix[10] prepare dem1 fair
using problem directory /home/mike/Workbench/NGC
Which one?
/home/mike/Workbench/NGC/dem1.bds
/home/mike/Workbench/NGC/dem1.gp
unix[11] prepare dem1.gp fair
using problem directory /home/mike/Workbench/NGC
using program directory /home/mike/bin/obj
linking of fair succeeded
unix[12] fair
GETPDF attached problem definition file /home/mike/Workbench/NGC/dem1.gp
problem name: Dembo 1b
this is a GP so coordinates can be in
  log variables ------> x(j)   or
  natural variables --> t(j) = exp(x(j))
Will you supply x coordinates? no
------------------------------
Starting point
Is a starting point given? yes
t0(1)...= 4 4 4 4 4 4 4 4 4 4 4 4
------------------------------
Optimal point
x*=
  9.2349308508742312D-01  9.3108212220296505D-01  2.0356755104740012D+00
  1.6965376971782853D-01  2.0412515540833165D+00  2.6394678185890963D-01
  1.4544070300971716D+00  1.0252100808836437D+00  5.5265690715826377D-01
  6.8004439678685880D-01  1.8923106529166507D+00  1.8734658601195413D+00

t*=
  2.5180708813511754D+00  2.5372533107484889D+00  7.6574230482487140D+00
  1.1848945339234866D+00  7.7002404379761682D+00  1.3020589013027175D+00
  4.2819436489496852D+00  2.7876810375501173D+00  1.7378642332906757D+00
  1.9739653680047697D+00  6.6346814290021374D+00  6.5108229413933074D+00

Is this the optimal point? yes
------------------------------
Bounds deduced from the constraints or given in the problem statement
read variable limits from a file? yes
Name of variable limits file: /home/mike/Workbench/NGC/dem1.bds
tH( 1)=1.8643742321108631D+01
          XH( 1)=log(tH( 1))= 2.9255105574060063D+00
tL( 1)=0
tL( 1) = 0 => no lower bound on X( 1)
tH( 2)=4.5737816383340714D+01
          XH( 2)=log(tH( 2))= 3.8229254476942498D+00
tL( 2)=0
tL( 2) = 0 => no lower bound on X( 2)
tH( 3)=1.0231902698125115D+01
          XH( 3)=log(tH( 3))= 2.3255105546657582D+00
tL( 3)=0
tL( 3) = 0 => no lower bound on X( 3)
tH( 4)=1.0D+09
          XH( 4)=log(tH( 4))= 2.0723265836946410D+01
tL( 4)=0
tL( 4) = 0 => no lower bound on X( 4)
tH( 5)=1.0D+09
          XH( 5)=log(tH( 5))= 2.0723265836946410D+01
tL( 5)=0
tL( 5) = 0 => no lower bound on X( 5)
tH( 6)=1.0D+03
          XH( 6)=log(tH( 6))= 6.9077552789821368D+00
tL( 6)=0
tL( 6) = 0 => no lower bound on X( 6)
```

```
tH( 7)=
tL( 7)=0
tL( 7) = 0 => no lower bound on X( 7)
tH( 8)=1.0D+03
          XH( 8)=log(tH( 8))= 6.907755278982136 8D+00
tL( 8)=0
tL( 8) = 0 => no lower bound on X( 8)
tH( 9)=1.0D+05
          XH( 9)=log(tH( 9))= 1.151292546497022 9D+01
tL( 9)=0
tL( 9) = 0 => no lower bound on X( 9)
tH(10)=
tL(10)=0
tL(10) = 0 => no lower bound on X(10)
tH(11)=1.0D+04
          XH(11)=log(tH(11))= 9.210340371976183 6D+00
tL(11)=0
tL(11) = 0 => no lower bound on X(11)
tH(12)=
tL(12)=0
tL(12) = 0 => no lower bound on X(12)
------------------------------
Computing new bounds using the algorithm...
XH=
  2.9255105574060063D+00  3.8229254476942498D+00  2.3255105546657582D+00
  2.0723265836946410D+01  2.0723265836946410D+01  6.9077552789821368D+00
  2.0674210508927011D+00  6.9077552789821368D+00  1.1512925464970229D+01
  8.4487940044502086D+00  9.2103403719761836D+00  6.2580093511163977D+00

XL=
 -1.5292183516622515D-01 -1.0503367254544689D+00  4.4707816757402297D-01
 -1.7950677114706629D+01 -1.7950677114706629D+01 -4.1351665567423552D+00
  7.0516767134707981D-01 -4.1351665567423552D+00 -8.7403367427304470D+00
 -5.6762052822104270D+00 -6.4377516497364020D+00 -3.4854206288766161D+00

Beginning adjustments...
 XL( 2) changes from -1.0503367254544689D+00 to -1.0503367254544691D+00
 XL( 7) changes from  7.0516767134707981D-01 to  7.0516767134707958D-01
... 1 adjustments were performed.
------------------------------
New bounds:
XH=
  2.9255105574060063D+00  3.8229254476942498D+00  2.3255105546657582D+00
  2.0723265836946410D+01  2.0723265836946410D+01  6.9077552789821368D+00
  2.0674210508927011D+00  6.9077552789821368D+00  1.1512925464970229D+01
  8.4487940044502086D+00  9.2103403719761836D+00  6.2580093511163977D+00

XL=
 -1.5292183516622515D-01 -1.0503367254544691D+00  4.4707816757402297D-01
 -1.7950677114706629D+01 -1.7950677114706629D+01 -4.1351665567423552D+00
  7.0516767134707958D-01 -4.1351665567423552D+00 -8.7403367427304470D+00
 -5.6762052822104270D+00 -6.4377516497364020D+00 -3.4854206288766161D+00

Want bounds in a file? yes
Name of fair bounds file [dem1.fair]:
unix[13]
```

If a variable limit could not be deduced from the inequality constraints, I left that line blank in the file (an interactive user would have pressed return in response to a prompt). The program echos the values as it reads them from the file and shows the corresponding x_j limits.

Once all of the data have been input, the program computes fair bounds using the algorithm of §26.2.2 and adjusts two of them using the algorithm described above. Here only one iteration of the adjustment process is needed, but sometimes it takes several. The resulting numbers have many digits, so I ask that the `x` values be written to the file `dem1.fair`.

Later I copied the new bounds from `dem1.fair` into `NGC/dem1.gp` as the catalog bounds for this problem. The midpoint of these bounds in each coordinate direction is the starting point that I entered (modulo the tiny adjustments described above). For example,

$$\begin{aligned} \texttt{XO(1)} &= (\texttt{XH(1)} + \texttt{XL(1)})/2 \\ &= (2.9255105574060063 - 0.15292183516622515)/2 \\ &= 1.3862943611198906 \\ t_1^0 = \exp(\texttt{XO(1)}) &= 4 \end{aligned}$$

It can also be verified that the final bounds include $\mathbf{x}^\star$, and that they include (or equal) the variable limits deduced from the constraints.

5.2 Validating Derivative Calculations

An important step in the definition of a type `1` or type `2` test problem is confirming that the routines it uses to compute gradient vectors and Hessian matrices are coded correctly and that the formulas they evaluate are themselves correct. In §25.6.6, I advocate comparing the outputs of `GRD` and `HSN` to approximations calculated by finite-differencing function values. This Section describes the central-differencing subroutine `GRDCD` and the program `grdtest`, which compares `GRDCD` approximations to the analytic gradients computed by `GRD`.

To be a practical tool, `grdtest` and `GRDCD` must address several issues that are not evident from the discussion in §25.6.6.

- The exposition there blithely assumes that all of the gradient and Hessian elements we find by finite-differencing will not be too different in magnitude from 1, and the MATLAB routines `gradcd`, `hesscd`, and `gradtest` listed there work for toy problems like `egg` in which that is the case. In many real optimizations, including some that are interesting as test problems, the components of `X` and the values of the constraint functions and objective can be much bigger or much smaller than 1. Instead of $\delta = u^{1/3}$, a practical routine for central-differencing $f(x)$ ideally uses the step size [132, §5.7]

  $$\delta = \left(\frac{f(x)}{f'''(x)} \right)^{1/3} u^{1/3} = s u^{1/3}$$

  in which the factor s is called the **curvature scale.** Information about the third derivative of the function is seldom conveniently available, so usually the best we can do is to scale the step in simple proportion to the size of x. That approach fails when $x \approx 0$, so `GRDCD` uses $\delta = \max(u, |x|) \times u^{1/3}$.

- The `gradcd` code in §25.6.5 looks like this

  ```
  yp=fcn(x+delta)
  ym=fcn(x-delta)
  g(j)=(yp-ym)/(2*delta)
  ```

  but in machine arithmetic the quantity `(x+delta)-(x-delta)` is hardly ever precisely equal to `2*delta`; instead `GRDCD` uses code that would look like this in MATLAB

  ```
  xp=x+delta
  yp=fcn(xp)
  xm=x-delta
  ym=fcn(xm)
  g(j)=(yp-ym)/(xp-xm)
  ```

  to remove that source of imprecision in the denominator.

- The `gradtest` routine of §25.6.6 computes the relative disagreement between the analytic and approximate gradient as the maximum absolute difference between their components divided by the norm of the approximate gradient, but only if that norm is not too small. It is more revealing to compute the discrepancy in each gradient element j separately by using the **compromise difference**

  $$\varepsilon = \frac{|\mathtt{g}_j - \mathtt{gcd}_j|}{1 + |\mathtt{gcd}_j|}$$

  discussed in [100, Exercise 4.10.47] (it is also mentioned in §25.5). This measure behaves like relative difference when $|\mathtt{gcd}_j|$ is large and like absolute difference when $|\mathtt{gcd}_j|$ is small.

- Despite the precautions described above, comparing analytic gradients to approximate gradients is a process rife with the exigencies that afflict all numerical calculations. The heuristic for setting δ sometimes yields an inappropriate value, and the central-difference derivative approximation is sometimes not sufficiently precise. Roundoff errors [100, §4.3] are always present and can sometimes have a dramatic effect on the comparison.

  If after testing enough points `grdtest` reports that the gradients are close enough, it is almost certain that both `FCN` and `GRD` are correct; if `grdtest` reports that the gradients are *not* close enough, it is probably because there is a mistake in one or the other. In that case a gradient element from `GRD` is usually so different from the corresponding element from `GRDCD` that it is obvious something is wrong.

  Sometimes, however, a comparison suggests that there might be an error when nothing is wrong (a false alarm is much more likely than a false confirmation) or the results are equivocal. Then it is important for the program to provide enough information to support a sound subjective judgement about whether its finding is spurious or justifies a minute and possibly arduous snipe hunt through both `FCN` and `GRD` in search of a mistake that might not be there.

5.2.1 GRDCD

The first two considerations discussed above are addressed in the source code of GRDCD, which is listed below and on the next page.

Here U is the unit roundoff, which is half of machine epsilon [125, p23] (see p573). Its value in REAL*8 numbers is 2^{-53} or 1.11022302462515654D-16, and for $x = 1$ the optimal value of DELTA is[9] its cube root, 4.80621738393735534D-06. Both of these quantities are specified [32,34] using hexadecimal numbers (see [100, §4]) to avoid any ambiguity about their internal representation.

To calculate function values at points displaced from X we must change X(J), so each iteration of the loop over J begins [44-45] by saving it to restore [62-63] later. To compute the scale factor XABS to use for this element of X we [46] find its absolute value and [47] if that is too small use U instead. Then [49-50] the step size for this coordinate can be found.

```
 1 C
 2 Code by Michael Kupferschmid
 3 C
 4       SUBROUTINE GRDCD(FCN,X,N,I, G)
 5 C     This routine approximates G by central-differencing function
 6 C     values.
 7 C
 8 C     variable  quantity
 9 C     --------  --------
10 C     DABS      Fortran function returns |REAL*8|
11 C     DELTA     step length in each direction for unit scaling
12 C     FCN       routine returns function value
13 C     FM        function value at X-SDELTA*e_j
14 C     FP        function value at X+SDELTA*e_j
15 C     G         approximate gradient vector returned
16 C     I         index of the function whose gradient is wanted
17 C     J         index on the variables
18 C     N         number of variables
19 C     SDELTA    scaled step size
20 C     U         machine epsilon for REAL*8
21 C     X         point at which gradient is to be approximated
22 C     XABS      |XSAVE|
23 C     XM        coordinate one step back
24 C     XP        coordinate one step forward
25 C     XSAVE     original value of X(J)
26 C
27 C     formal parameters
28       EXTERNAL FCN
29       REAL*8 FCN,X(N),G(N)
30 C
31 C     scale a step size of u^(1/3) to approximate first derivatives
32       REAL*8 U/Z'3CA0000000000000'/
33 C            U = 1.11022302462515654D-16
34       REAL*8 DELTA/Z'3ED428A2F98D728F'/
35 C            DELTA = 4.80621738393735534D-06
36 C
37 C     other local variables
38       REAL*8 XSAVE,XABS,SDELTA,XP,FP,XM,FM
```

[9] The values given for delta in both gradcd and hesscd in §25.6.5 were computed using a slightly different value for U, which I found experimentally (in antiquity) by running the program described in [100, Ex 4.10.24] on a processor that did not, alas, conform precisely to the IEEE floating point standard. This anachronism will be corrected in the next edition of *Introduction to Mathematical Programming.*

```
39 C
40 C ---------------------------------------------------------------
41 C
42 C     compute each component separately
43       DO 1 J=1,N
44 C          save the original component value
45            XSAVE=X(J)
46            XABS=DABS(XSAVE)
47            IF(XABS .LT. U) XABS=U
48 C
49 C          scale the step size
50            SDELTA=DELTA*XABS
51 C
52 C          find the function value +SDELTA away in direction J
53            XP=XSAVE+SDELTA
54            X(J)=XP
55            FP=FCN(X,N,I)
56 C
57 C          find the function value -SDELTA away in direction J
58            XM=XSAVE-SDELTA
59            X(J)=XM
60            FM=FCN(X,N,I)
61 C
62 C          reset X(J) to its original value
63            X(J)=XSAVE
64 C
65 C          approximate the G component by central difference
66            G(J)=(FP-FM)/(XP-XM)
67     1 CONTINUE
68       RETURN
69       END
```

The J'th coordinate of X is [53-54] displaced in the positive direction, [55] the function value is found at that point, [58-59] X(J) is displaced in the negative direction, and [60] the function value is found at that point. Finally [65-66] the central difference approximation of G(J) is calculated.

5.2.2 grdtest

The FCN and GRD routines listed on the next page are from ek1.f, but one sign error has been made in coding f_1 and another in coding ∇f_2.

The terminal session shows that these mistakes result in significant discrepancies between G(1) and its central-difference approximation GCD(1) for function 1 and function 2. In response to the prompt Source of X [1]: I pressed return to accept the default choice of 1, so the comparison is made at the optimal point. The discrepancy found for function 3 is zero and that for function 4 is on the order of 10^{-10}.

When a significant discrepancy is found the program prints the index I of the function and tabulates all of the elements J for which G(J) and GCD(J) are significantly different. The fourth field in each line of the table consists of three logical values, TTF in this example, indicating whether the values agree in sign, agree in being zero or nonzero, and agree in value. Here the discrepant components are of the same sign and both are nonzero, but the compromise differences between G(1) and GCD(1) are about 0.88 and 6.46.

```
      FUNCTION FCN(X,N,II)
      REAL*8 FCN,X(N)
      IF(II.EQ.1) FCN=8.D0*DEXP((X(1)+12.D0)/9.D0)-X(2)+4.D0
      IF(II.EQ.2) FCN=6.D0*(X(1)-12.D0)**2+25.D0*X(2)-600.D0
      IF(II.EQ.3) FCN=-X(1)+12.D0
      IF(II.EQ.4) FCN=(X(1)-20.D0)**4+(X(2)-12.D0)**4
      RETURN
      END
C                                                    sign should be -
      SUBROUTINE GRD(X,N,II, G)
      REAL*8 X(N),G(N)
      IF(II.EQ.1) THEN
         G(1)=8.D0*DEXP((X(1)-12.D0)/9.D0)*(1.D0/9.D0)
         G(2)=-1.D0
      ELSEIF(II.EQ.2) THEN
         G(1)=6.D0*2.D0*(X(1)+12.D0)
         G(2)=25.D0
      ELSEIF(II.EQ.3) THEN
         G(1)=-1.D0
         G(2)= 0.D0
      ELSEIF(II.EQ.4) THEN
         G(1)=4.D0*(X(1)-20.D0)**3
         G(2)=4.D0*(X(2)-12.D0)**3
      ENDIF
      RETURN
      END
```

```
unix[14] prepare ek1wrong.f grdtest
using problem directory /home/mike/Workbench/NGC
using program directory /home/mike/bin/obj
linking of grdtest succeeded
unix[15] grdtest
Ecker & Kupferschmid 1 wrong
N= 2  MI= 3  ME= 0
Functions to check: 1-4
How many random points? 0
Source of X:
  1  optimal point
  2  starting point, midpoint of bounds
  3  the keyboard; entered by the user
  4  a file containing performance measurement records
Source of X [1]:
for I= 1
  J  G(J)                    GCD(J)                  +0= error
  1  1.3304187353169163D+00  1.9147274810047701D+01 TTF 8.843308D-01
 error  8.843308D-01 for I=  1
function value =  1.603517D+02
Want to see full gradients? yes
G=
  1.3304187353169163D+00 -1.0000000000000000D+00
GCD=
  1.9147274810047701D+01 -9.9999999997686229D-01
for I= 2
  J  G(J)                    GCD(J)                  +0= error
  1  3.3155389082767607D+02  4.3553890827913406D+01 TTF 6.464082D+00
 error  6.464082D+00 for I=  2
function value = -1.216166D+02
Want to see full gradients? yes
G=
  3.3155389082767607D+02  2.5000000000000000D+01
GCD=
  4.3553890827913406D+01  2.4999999999976861D+01
```

```
 error  0.000000D+00 for I=  3
Want to see full gradients? no
 error  3.742265D-10 for I=  4
Want to see full gradients? no
mismatch found
Source of X:
  1  optimal point
  2  starting point, midpoint of bounds
  3  the keyboard; entered by the user
  4  a file containing performance measurement records
Source of X [1]: EOF
Functions to check: EOF
unix[16]
```

After each table of discrepant gradient elements the program prints another `error` value, which is the maximum of the errors in the components, and the value of the function at the test point. Here that is about `160` for the first function and about `-121` for the second, both of modest magnitude suggesting that the difference between the gradient values is probably not due to some numerical difficulty (e.g., underflow, overflow, loss of significance due to cancellation) in the calculation of either `G(J)` or `GCD(J)`.

Now that this point has been tested the program prompts for another, to which I responded by sending CTRL-D. The program then prompts again for `functions to check`, so if I wanted to further investigate one of the disagreements I could try that function at other points, but here I entered CTRL-D again to stop the program.

Testing one point might not reveal a mistake in `GRD`, so the program provides for testing multiple points generated at random within the catalog bounds for the problem. In the terminal session below it is used to validate the correctly-coded `ek1.f`, and reports that `all gradients match`.

```
unix[17] prepare ek1.f grdtest
using problem directory /home/mike/Workbench/NGC
using program directory /home/mike/bin/obj
linking of grdtest succeeded
unix[18] grdtest
Ecker & Kupferschmid 1
N= 2  MI= 3  ME= 0
Functions to check: 1-4
How many random points? 100
 error  2.682382D-11 for I=  1
 error  2.284025D-10 for I=  2
 error  0.000000D+00 for I=  3
 error  1.327998D-08 for I=  4
all gradients match
Functions to check: EOF
unix[19]
```

When the problem is a GP, QP, or LP its functions and gradients are computed by standard routines that have been carefully checked and tested, so there is no question that they are correctly coded. In the terminal session on the next page, `grdtest` is used to compare the gradients for `dem1.gp` with those returned by `GRDGP`.

```
unix[20] prepare dem1.gp grdtest
using problem directory /home/mike/Workbench/NGC
using program directory /home/mike/bin/obj
linking of grdtest succeeded
unix[21] grdtest
Dembo 1b
N=12  MI= 3  ME= 0
Functions to check: 1-4
How many random points? 10
 error  4.881446D-09 for I=  1
for I= 2
  J  G(J)                    GCD(J)                 +0= error
  2  7.2327997061554906D+03  7.2177364773891532D+03 TTF 2.086685D-03
  7  1.0727624209358387D-01  0.0000000000000000D+00 TFF
 12 -7.2327224811694914D+03 -7.2727036686235124D+03 TTF 5.496675D-03
function value =  5.540470D+13
Want to see X? no
Continue? yes
for I= 2
  J  G(J)                    GCD(J)                 +0= error
  2  6.0982481012192558D-02  0.0000000000000000D+00 TFF
  6  3.2925882778144441D-02  6.2956003859875556D-02 TTF 2.825152D-02
  7  8.5447803245334214D-02  0.0000000000000000D+00 TFF
 10  4.9840407540355507D-04  0.0000000000000000D+00 TFF
 12  1.5964439496130139D-01  1.5526409322004345D-01 TTF 3.791602D-03
function value =  2.432028D+10
Want to see X? no
Continue? yes
 error  2.825152D-02 for I=  2
for I= 3
  J  G(J)                    GCD(J)                 +0= error
  6  4.8106633944396983D-04  0.0000000000000000D+00 TFF
  8  2.6374344735476635D-02  2.6528301539472859D-02 TTF 1.499781D-04
function value =  2.729862D+08
Want to see X? no
Continue? no
Functions to check: EOF
unix[22]
```

Some randomly-generated points happen to be far from optimal, and there the comparisons produce equivocal results. In some cases `GRDCD` returns zero while `GRD` does not, and in others the compromise difference is big enough to arouse our suspicion that something is wrong. But the trouble is in *different* components at different trial points, and it is unlikely that they were all coded wrong. It would be surprising if the simple step-size scaling we used in `GRDCD` were a good approximation to the curvature scale s when $f(x)$ has the enormous values reported. These considerations should lead us to doubt the test rather than the code.

Although `grdtest` is usually a reliable way to check for agreement of the `FCN` and `GRD` routines of a test problem, it must be used with some discretion.

5.3 Testing Feasibility: `feastest`

Often in the course of conducting computational experiments it is necessary to compute constraint or objective function values at a trial point. The terminal session on the next page illustrates how the `feastest` program can be used to do that.

The `prepare` shell program is used as usual to link an executable, and at `[24]` the program is run. It writes out the long name of the test problem, `Dembo 1b`, and gives its dimensions. Then it prompts for the source of a trial point to test for feasibility and offers several alternatives. At the prompt I press return to accept the default choice of `1`, and decline to see `X`.

Then the values of the three inequality constraint functions are displayed, in increasing order (i.e., the most nearly infeasible one is on the bottom above the line). The function values are all negative so this point is feasible, and the objective value that the program calculates agrees with the record value that is given in the problem definition file. If a function value were exactly zero that constraint would also be considered satisfied, even though the display says `< 0`. If the problem had equality constraints, those that were satisfied would be marked `= 0`.

Next the program prompts for another point to try and I select option `4`, to read a point from a performance measurements file. The program prompts for the name of the file and the line of interest. The line I choose is `5`, which is the first performance measurement record and therefore contains the starting point. If I had instead chosen option `2` the program would have produced the same point by calculating the midpoint of the bounds. This point is feasible for constraint `1`, which is listed above the line, but it violates constraints `2` and `3`. They are now listed below the line, with the most infeasible one on the bottom, and they are marked `> 0`. If the problem had equality constraints, those that were violated (the function's absolute value is greater than `TEQ`) would be marked `# 0`.

In response to the third prompt for a trial point I chose option `3` and entered the point at the keyboard. This one violates all of the constraints, so none of them are listed above the line. In response to the final prompt I entered CTRL-D to stop the program.

```
unix[23] prepare dem1.gp feastest
using problem directory /home/mike/Workbench/NGC
using program directory /home/mike/bin/obj
linking of feastest succeeded
unix[24] feastest
GETPDF attached problem definition file /home/mike/Workbench/NGC/dem1.gp
Dembo 1b
N=12  MI= 3  ME= 0

Source of X:
  1  optimal point
  2  starting point, midpoint of bounds
  3  the keyboard; entered by the user
  4  a file containing performance measurement records
Source of X [1]:
Want to see X? no

  f01=-1.4965806371947110D-13 < 0
  f02=-3.3306690738754696D-16 < 0
  f03=-1.1102230246251565D-16 < 0
  -------------------------------

  f04= 3.1682205099253444D+00 objective

Source of X:
  1  optimal point
  2  starting point, midpoint of bounds
  3  the keyboard; entered by the user
  4  a file containing performance measurement records
Source of X [1]: 4
Name of performance measurements file: dem1.EA3
Line number: 5
Want to see X? no

  f01=-1.9995639919999997D-01 < 0
  -------------------------------
  f03= 7.5431846349016474D-01 > 0
  f02= 7.5707601629352550D-01 > 0

  f04= 2.2768264948101177D-01 objective

Source of X:
  1  optimal point
  2  starting point, midpoint of bounds
  3  the keyboard; entered by the user
  4  a file containing performance measurement records
Source of X [1]: 3
Enter 12 values
X(1)...= 2 2 2 2 2 2 2 2 2 2 2 2
  -------------------------------
  f01= 6.4552442305894875D-01 > 0
  f02= 4.9681975420733666D+00 > 0
  f03= 4.9731240422946046D+00 > 0

  f04= 7.2334899444213840D-04 objective

Source of X:
  1  optimal point
  2  starting point, midpoint of bounds
  3  the keyboard; entered by the user
  4  a file containing performance measurement records
Source of X [1]: EOF
unix[25]
```

5.4 Testing Optimality: `kktcheck`

How can we tell whether the answer $\bar{\mathbf{x}}$ returned by an optimization algorithm really is the optimal point $\mathbf{x}^\star$? If the objective and constraints of the problem are differentiable functions (`FLAGS(2)=2`) and if a constraint qualification holds (`FLAGS(3)=2`) then, according to the KKT necessary conditions of §16.4, $\bar{\mathbf{x}}$ is a local minimum only if it is feasible and we can find multipliers $\bar{\boldsymbol{\lambda}}$ that satisfy the stationarity, orthogonality, and nonnegativity conditions given in §16.3. If in addition the objective and constraints are convex functions (`FLAGS(1)=2`) then, according to the KKT sufficient conditions of §16.4, $\bar{\mathbf{x}}$ is a global minimizing point.

Many of the problems we use in computational testing are *not* convex, and then it is possible that a point $(\bar{\mathbf{x}}, \bar{\boldsymbol{\lambda}})$ satisfying the KKT conditions is *not* a local minimum, but if we claim to know $\mathbf{x}^\star$ for a problem, whether it is convex or not we should at least be able to find a $\boldsymbol{\lambda}^\star$ such that $(\mathbf{x}^\star, \boldsymbol{\lambda}^\star)$ does satisfy the conditions. As discussed in §16.10 that can be done by identifying the index set $\mathbb{I}$ of the tight inequalities and then finding a $\boldsymbol{\lambda}^\star \geq \mathbf{0}$ that comes as close as possible to satisfying the stationarity conditions including only those constraints. That we can do by solving a linear program to minimize the sum of the absolute values of the row deviations, and if that sum turns out to be near zero then $\bar{\mathbf{x}}$ is stationary. The simplex algorithm ensures that the λ_i we find in this way will be nonnegative, and setting the other λ_i to zero ensures that the orthogonality condition is satisfied.

Problems that also (or only) have equality constraints can be analyzed in the same way if we represent each equality by opposing inequalities, as described in §16.6. The KKT multiplier associated with an equality constraint can have either sign, but it can be written as $\lambda = \lambda^+ - \lambda^-$ where $\lambda^+ \geq 0$ and $\lambda^- \geq 0$, and then the stationarity conditions require that

$$\frac{\partial f_{m+1}}{\partial x_j} + \sum_{i \in \mathbb{I}} \lambda_i \frac{\partial f_i}{\partial x_j} + \sum_{i \in \mathbb{E}} (\lambda_i^+ - \lambda_i^-) \frac{\partial f_i}{\partial x_j} = 0, \quad j = 1 \ldots n$$

where $\mathbb{I}$ contains the indices in `1...MI` of those inequality constraints that are active at $\bar{\mathbf{x}}$ and $\mathbb{E}$ contains the indices of the equality constraints `MI+1...MI+ME`, all of which must be active because $\bar{\mathbf{x}}$ is feasible. The deviation for row j in this set of equations is the quantity on the left-hand side. If we represent it as $d_j^+ - d_j^-$, then according to §1.5.2 the *absolute* deviation for row j is $d_j^+ + d_j^-$ and we can minimize the sum of the absolute row deviations by solving this linear program.

$$\begin{aligned}
\underset{\mathbf{d}^+, \mathbf{d}^-, \boldsymbol{\lambda}}{\text{minimize}} \quad & z = \sum_{j=1}^{n} (d_j^+ + d_j^-) \\
\text{subject to} \quad & (d_j^+ - d_j^-) - \sum_{i \in \mathbb{I}} \lambda_i \frac{\partial f_i}{\partial x_j} - \sum_{i \in \mathbb{E}} (\lambda_i^+ - \lambda_i^-) \frac{\partial f_i}{\partial x_j} = \frac{\partial f_{m+1}}{\partial x_j}, \quad j = 1 \ldots n \\
& \mathbf{d}^+, \mathbf{d}^-, \boldsymbol{\lambda} \geq \mathbf{0}
\end{aligned}$$

The simplex tableau for this linear program looks like the second one in §16.10 except that it has extra columns for the λ_i^+ and λ_i^- of the equality constraints.

For a given $\bar{\mathbf{x}}$ that is alleged to be a KKT point, the `kktcheck` program[10] finds the active set, constructs the linear program described above, and uses the `SIMPLX` library subroutine to solve it for the corresponding $\bar{\boldsymbol{\lambda}}$. If $\bar{\mathbf{x}} = \mathbf{x}^\star$ (that is, if we use for `X` the catalog record point `XR` from `/NGC4/` and it really is optimal) then we expect the **residual,** the amount by which the stationarity conditions are violated, to be very small. In that case we can declare $\bar{\boldsymbol{\lambda}}$ to be $\boldsymbol{\lambda}^\star$ and use it for `MULTS` in `/NGC5/`.

In the terminal session below the program discovers multipliers already present in `/NGC5/` because when I coded this problem I solved it analytically, found that $\boldsymbol{\lambda}^\star = [\frac{2}{3}, \frac{2}{3}]^\top$, and listed those values on line 17 of the problem definition file `him24.qp`. Here I let the program compute the multipliers numerically and it found exactly the same values, yielding a residual of zero. This problem is convex, so the existence of these multipliers means the catalog `XR` is definitely the optimal point.

```
unix[26] prepare him24.qp kktcheck
using problem directory /home/mike/NGC
using program directory /home/mike/bin/obj
linking of kktcheck succeeded
unix[27] kktcheck
Himmelblau 24
N= 2  MI= 2  ME= 0  M= 2
Source of X:
  1  optimal point
  2  starting point, midpoint of bounds
  3  the keyboard; entered by the user
  4  a file containing performance measurement records
Source of X [1]:
X
  1.0000000000000000D+00  1.0000000000000000D+00
multipliers are already given
them
  6.6666666666666663D-01  6.6666666666666663D-01
Want to check them? no
Find new multipliers yes
them
  6.6666666666666663D-01  6.6666666666666663D-01
Want to check them? yes
Sum of absolute row deviations is 0.000000D+00

Try another point? no
unix[28]
```

At the top in the terminal session on the next page I study the test problem numbered 71 in [81], which when I ran the program did not have catalog KKT multipliers. There are `M=10` constraints altogether so there are 10 multipliers; the last of them corresponds to the equality constraint and turned out to be positive. The catalog optimal point for this problem is exact[11] but `feastest` finds that it violates the functional inequality constraint by $\approx 8 \times 10^{-8}$. The solution published in [81] violates this constraint by $\approx 8 \times 10^{-7}$ so using the convergence tolerance `TEQ` of 1×10^{-7} set in `hs71.f`, `kktcheck` declares that point infeasible.

[10] The first version of this program was written by my colleague Dick Sacher and improved by students Tyge Rugenstein and Lee Quintas.

[11] Google AI finds it precise to 16 digits, but only its first component is exactly represented internally.

```
unix[29] prepare hs71.f kktcheck
using problem directory /home/mike/Workbench/NGC
using program directory /home/mike/bin/obj
linking of kktcheck succeeded
unix[30] kktcheck
Hock & Schittkowski 71
N= 4  MI= 9  ME= 1  M=10
Source of X:
  1  optimal point
  2  starting point, midpoint of bounds
  3  the keyboard; entered by the user
  4  a file containing performance measurement records
Source of X [1]:
Want to see X? no
Find new multipliers yes
Want to see them? yes
them
  1.0878712289081225D+00  0.0000000000000000D+00  0.0000000000000000D+00
  0.0000000000000000D+00  0.0000000000000000D+00  0.0000000000000000D+00
  0.0000000000000000D+00  0.0000000000000000D+00  5.5229366037562333D-01
  1.6146856646761359D-01

Want to check them? yes
Sum of absolute row deviations is 1.483601D-09

Try another point? yes
Source of X:
  1  optimal point
  2  starting point, midpoint of bounds
  3  the keyboard; entered by the user
  4  a file containing performance measurement records
Source of X [1]: 3
Enter  4 values
X(1)...= 1 4.7429994 3.8211503 1.3794082

point is infeasible at TEQ= 1.000000D-07

unix[31] prepare dem1.gp kktcheck
using problem directory /home/mike/Workbench/NGC
using program directory /home/mike/bin/obj
linking of kktcheck succeeded
unix[32] kktcheck
GETPDF attached problem definition file /home/mike/Workbench/NGC/dem1.gp
Dembo 1b
N=12  MI= 3  ME= 0  M= 3
Source of X:
  1  optimal point
  2  starting point, midpoint of bounds
  3  the keyboard; entered by the user
  4  a file containing performance measurement records
Source of X [1]:
Want to see X? no
multipliers are already given
them
  1.0369708103977944D-02  9.1959042896102119D+00  5.6825377443673606D+00
Want to check them? yes
Sum of absolute row deviations is 3.410529D-09
Find new multipliers yes
them
  1.0369708103979339D-02  9.1959042896102350D+00  5.6825377442255247D+00
Want to check them? yes
Sum of absolute row deviations is 3.410271D-09

Try another point? EOF
unix[33]
```

At the bottom in the terminal session on the previous page I used `kktcheck` to find multipliers for `dem1.gp`. At the time I ran the program there were already multipliers in its problem definition file, but the new ones it found yield a smaller sum of absolute row deviations so I updated the file with those.

Appendix 28.7 includes, for each problem that has constraints, the values of its $\lambda_i^\star$.

5.5 Plotting Error versus Effort Curves: `perfplot`

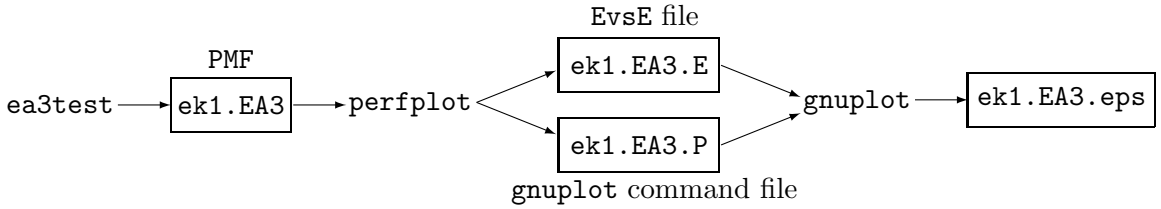
I describe in §26.4.1 a hypothetical program named `perfplot`, which writes a `gnuplot` command file to produce an error-versus-effort graph of the sort illustrated in §26.3.2 and §26.3.4. A practical implementation of this idea in the Workbench environment must differ in several ways from the program suggested there (it cannot be quite a solution to Exercise 26.6.48).

- The shell program of §26.4.2 assumes that each algorithm driver writes an error-versus-effort file, but in practice it is more convenient for it to write a **performance measurements** file or `PMF`. This is accomplished through calls to the `LOGPMR` subroutine described earlier, which writes, for each iteration of the algorithm under test, a **performance measurement record** or `PMR` containing the following quantities.

`PHASE`	an integer indicating whether the iterate is `1`=infeasible or `2`=feasible
`K`	the iteration number, where `K=0` is the starting point
`TALG`	the processor time consumed in algorithm code so far, in seconds
`NFE, NGE, NHE`	the numbers of function, gradient, and Hessian evaluations used so far
`F`	the objective function value
`N`	the number of variables
`X`	the `N` coordinates of the current iterate

  A driver program is responsible for recording only this information about the progress of its algorithm in solving only a single test problem.

- The `perfplot` program reads a single `PMF` and writes two output files, an **error-versus-effort** or `EvsE` file and a **gnuplot command file** telling how the data in the `EvsE` file are to be graphed. If the experiment consists of solving test problem `ek1` with `EA3` then the process of collecting and analyzing performance measurement data looks like this.



- The Workbench environment assumes that the files generated in this process have names that are formed according to certain conventions. Earlier you saw that a problem

definition file has the extension `.f` (if it contains source code) or `.gp`, `.qp`, or `.lp` (if it contains data). As shown above, the name of a `PMF`, `EvsE` file, `gnuplot` command file, or `gnuplot` output file is also constructed according to a rule. The names of files having these and other types that are used in the Workbench environment are obtained in its routines by calling the library subroutine `FNANDU`, whose `man` page includes a complete list along with the FORTRAN I/O unit numbers used for them.

- Two plausible measures of solution error are discussed in §26.3, distance from the optimal point and combined solution error. It is possible to imagine other measures that might be appropriate in special circumstances (e.g., maximum constraint violation, difference between primal and dual objectives) so `perfplot` allows for a user-defined error measure to be used. This is done by replacing the default `user.f` function subprogram, which does nothing, with one supplied by the experimenter. Values that are returned by this routine are plotted without further processing.

- Because of the way in which optimization algorithms converge (see §9.2) it is most revealing to plot the logarithm of the relative distance error or combined solution error. As I mentioned in §26.4.1 this is problematic at points where the solution error is zero. At the first such point that `perfplot` encounters in a `PMF`, it plots a vertical arrow pointing down to the effort axis of the error-versus-effort curve. Subsequent iterates are plotted to reveal the behavior of the algorithm after it has found the optimal point, but, to avoid clutter, if any of them also have zero error the vertical lines leading to them are just drawn down to the axis without arrowheads.

- As envisioned in the shell program of §26.4.2, it might be desirable to run `perfplot` without user interaction. This can be accomplished by invoking the program like this.

```
rm -f [EvsE file] [gnuplot command file]
cat perfplot.scr | perfplot 7=[PMF] 8=[EvsE file] 9=[gnuplot command file] 2> /dev/null
```

  where the appropriate filenames (or shell variables containing them) are substituted for the quantities shown in square brackets and the script file `perfplot.scr` contains two lines. The first line in this script is the numeral `1`, `2`, or `3` designating which error measure to use and the second line is the numeral `1`, `2`, or `3` designating which effort measure to use.

- Because `perfplot` generates an `EvsE` file and a `gnuplot` command file for a single experiment, plotting multiple error-versus-effort curves on a single set of axes requires that the several `gnuplot` command files be merged into one that includes a `plot [EvsE file]` command for each curve. This file should have a name like `ek1..P`, should give its output file a name like `ek1..eps`, and should include `set label` commands at coordinates appropriate to identify the different curves. The precise requirements of such multiple-curve graphs differ enough that creating a merged `gnuplot` command file is usually best done by hand with an editor.

The terminal session below illustrates the use of `perfplot` to draw an error-versus-effort curve showing the performance of `EA3` when it is used to solve the test problem `ek1`.

The program begins by printing the name of the problem and its dimensions. Then it prompts for the name of the `PMF`, suggesting that it should begin with the problem abbreviation `ek1` and end with the short name of the algorithm under test. When we used `prepare` to link `feastest` it put the problem abbreviation in `/NGC1/`, but until `perfplot` reads the `PMF` it has no way of knowing which algorithm was used. To this prompt I responded with the file name `ek1.EA3`.

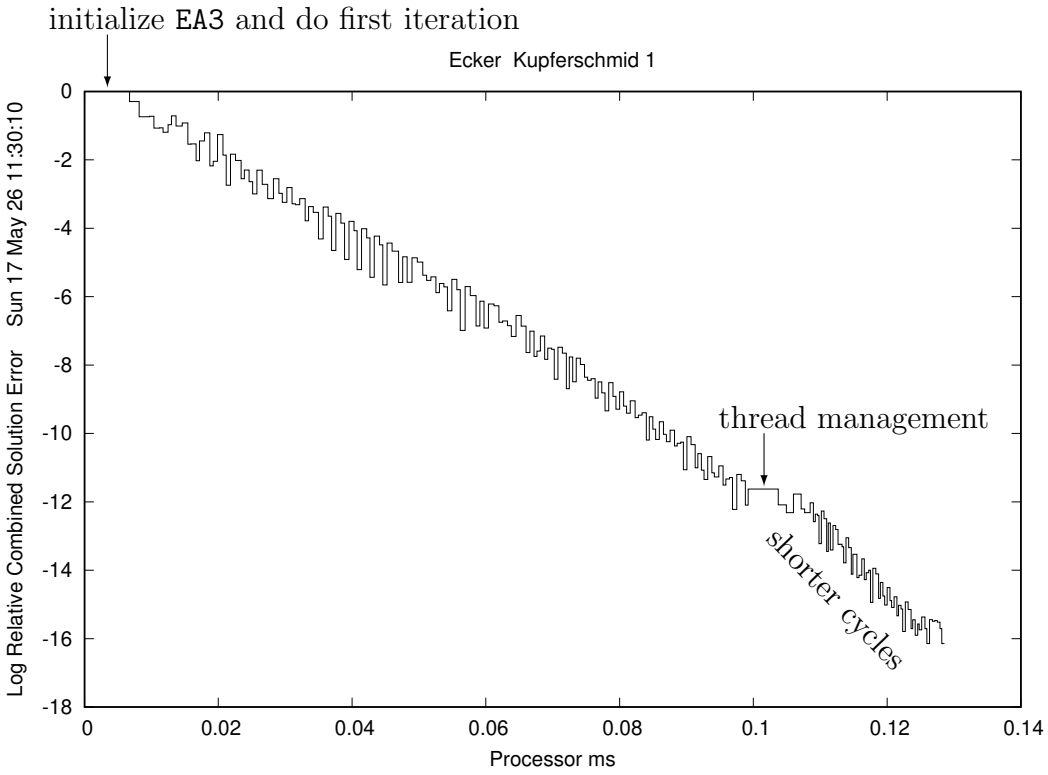
Next the program asks which error measure to use. When `perfplot.o` was made the compilation included the default `user.f` routine, which does nothing, so the menu offers only the choices `1` and `2`; I pressed return to accept the default, which is log relative combined solution error. This `PMF` includes timing measurements, so all three choices of effort measure are offered; if it had been written without timing, choice `1` would have been marked as unavailable. In response to this prompt I also pressed return to accept the default effort measure of processor time.

The program suggests names for the `EvsE` and `gnuplot` command files that it will write, and I accepted both by pressing return in response to the prompts.

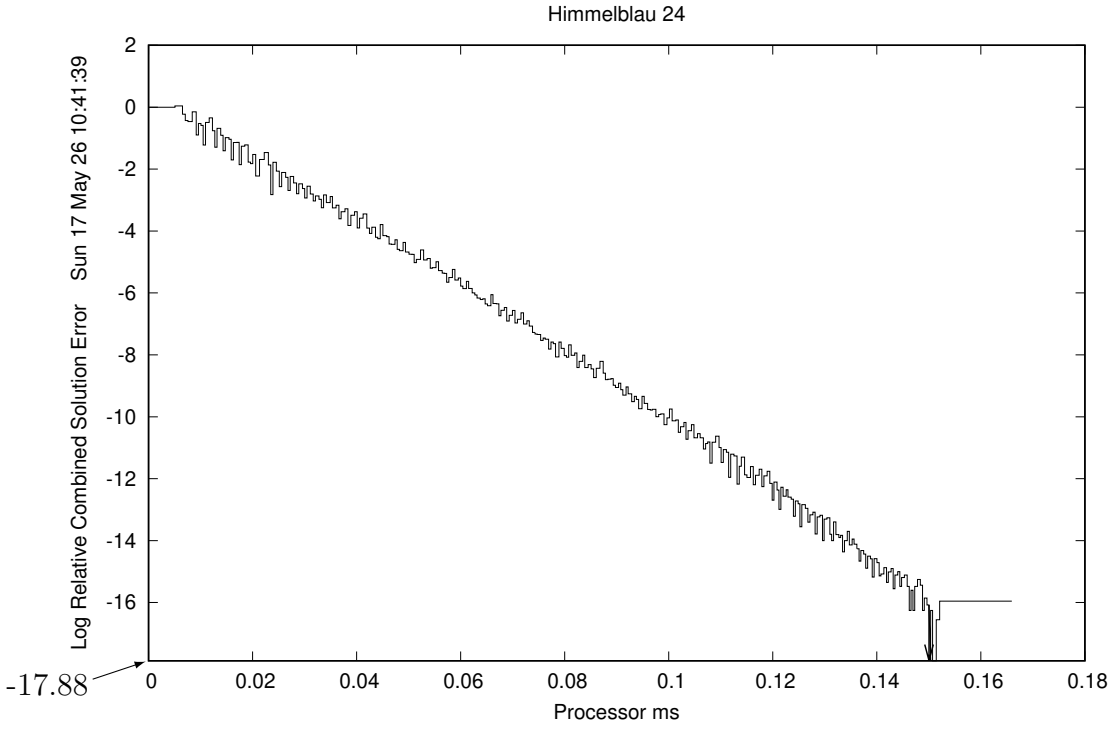
```
unix[34] prepare ek1 perfplot
using problem directory /home/mike/Workbench/NGC
using program directory /home/mike/bin/obj
linking of perfplot succeeded
unix[35] perfplot
Ecker & Kupferschmid 1
N= 2  MI= 3  ME= 0
Name of performance measurements file [ek1.????]: ek1.EA3
Error measure:
  1  log relative combined solution error
  2  log relative distance to optimal point
 (3) user-defined error
Error measure [1]:
Effort measure:
  1  processor time
  2  equivalent function evaluations
  3  iterations
Effort measure [1]:
Name of error-versus-effort file [ek1.EA3.E]:
Name of gnuplot commands file [ek1.EA3.P]:
unix[36] echo "load 'ek1.EA3.P'" | gnuplot
unix[37] gv ek1.EA3.eps
```

When `perfplot` finished (at `[36]`) I sent `gnuplot` a `load` command to use the command file that `perfplot` wrote, which in turn reads the `EvsE` file. When `gnuplot` finished (at `[37]`) I used `gv` to display the `.eps` file. The `.eps` file can also be included in a document; I included it in this one to produce the top picture on the next page.

This graph resembles the one in §26.3.4 but it has some new features. The delay before the first step down in solution error includes the time required to find the starting point and starting ellipsoid from the variable bounds as well as the time `EA3` spent performing its first iteration. The delay later on is probably due to the Intel processor switching the execution thread between cores; it happens in every repetition of the experiment, but at different times. The results graphed in §26.3.4 were obtained on a single-core fixed-speed processor.



After the second pause the iterations take less time, which suggests that the clock rate increased. I made no attempt to prevent these effects in running the experiment, because if the object of measuring processor time is to assess the performance of one algorithm it seems most realistic to accept the conditions that will prevail when the code is actually used.



The him24 problem is also easy for EA3, and its error-versus-effort curve, shown on the bottom of the previous page, resembles that for ek1. For this problem the algorithm finds several iterates that have zero error (it produced the exact solution $\mathbf{x}^\star = [1, 1]^\top$ three times). From the point on the error-versus-effort curve corresponding to the first of them, perfplot has drawn an arrow pointing down to the effort axis (it is not precisely at error level -17 so gnuplot draws no bottom tic mark). During this experiment there do not seem to have been any disturbances to the solution process. Here EA3 produces final iterates that are worse rather than better, and experience with many problems shows that this behavior is typical of the algorithm (the phenomenon is less obvious but also present in its solution of ek1).

The dem1 problem is much harder for EA3 to solve, as can be seen from the error-versus-effort curves on the next page. The algorithm's first step from the fair starting point is to one that is feasible but has an objective value $\sim 10^{14}$. After a while the linear convergence of the ellipsoid algorithm becomes apparent, but the last 168 iterates are infeasible and the solution error increases steadily during the final 36. The combined error measure penalizes infeasibility but the curves we draw using it normally include both feasible and infeasible points, and a point that is slightly infeasible but has a better objective value can have a lower combined error than one that is strictly feasible.

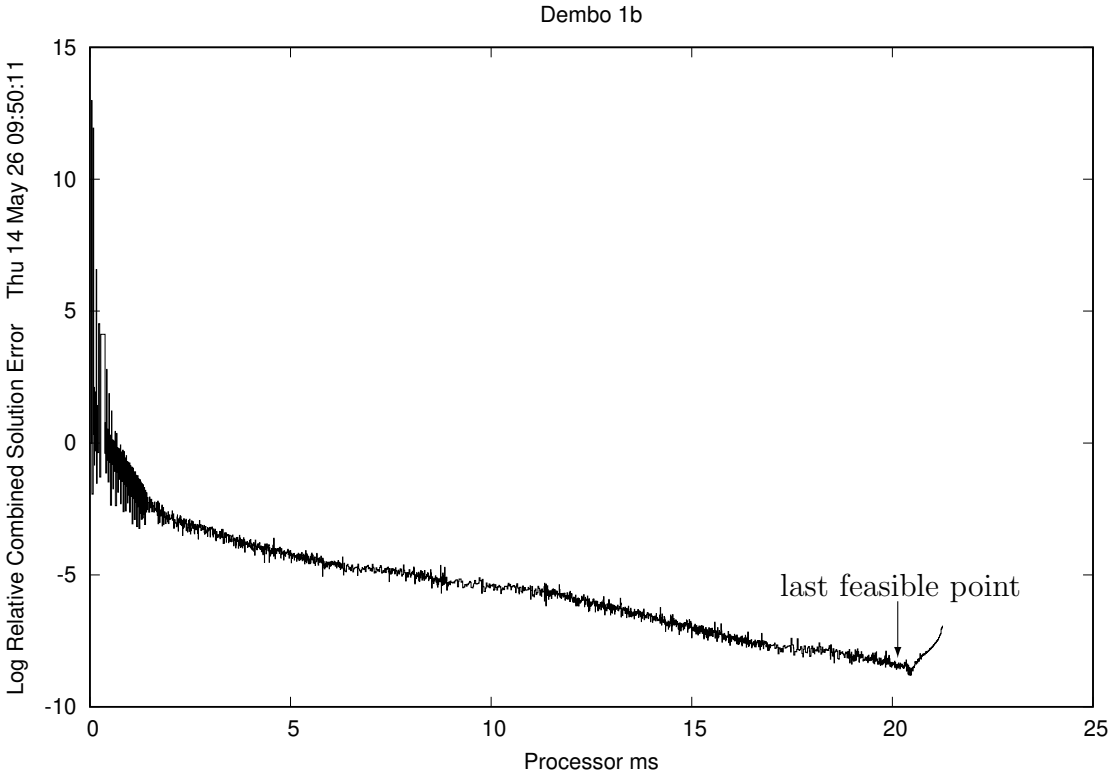
The shell program[12] listed below can be used to extract only the phase-2 points from a PMF (except for the starting point, which must be the midpoint of the bounds and is therefore included even if it is infeasible).

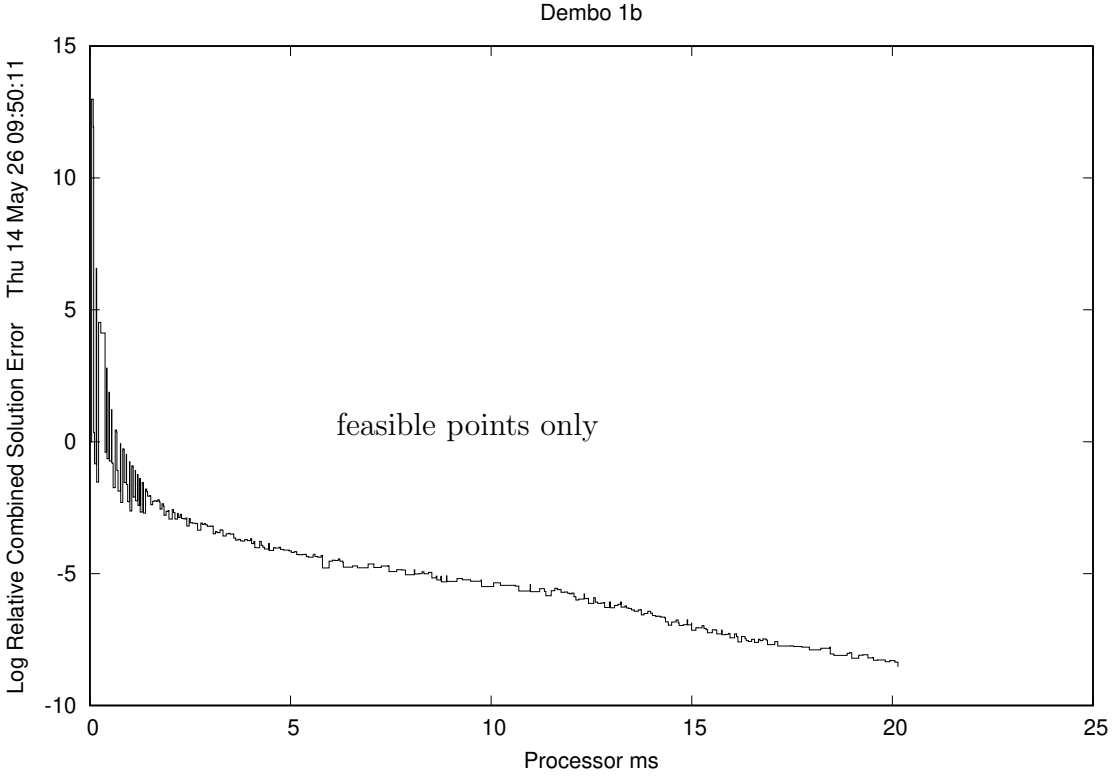
```
#!/bin/sh
pmflen=‘wc -l $1 | cut -f1 -d" "‘
fnabbr=‘echo "$1" | cut -f1 -d"."‘
rm -f $fnabbr.feas
cat $1 | rowcut 1-5 > $fnabbr.feas
cat $1 | rowcut 6-$pmflen | grep "^2" >> $fnabbr.feas
exit 0
```

Recall from the earlier discussion of LOGPMR that a phase-2 point satisfies all of the inequality constraints with zero tolerance and all of the equality constraints to within TEQ. I used this program to generate a file named dem1.feas and then ran perfplot to generate the picture on the bottom of the next page. Doing this eliminates the noise at the end of the solution process, and comparing the bottom picture to the top one reveals that feasible and infeasible iterates are mixed until then. It is another attribute of the ellipsoid algorithm that its current iterate pops into and out of the feasible set as the solution proceeds.

The examples in this Section illustrate some of the virtues and drawbacks of error-versus-effort curves. Tools for performing other kinds of analysis on performance measurement data can be integrated into the Workbench environment in a way similar to perfplot.

[12] This approach can be used to censor performance measurement data in other ways. FORTRAN source code for the rowcut utility is included in util.tar.gz.


Dembo 1b
Log Relative Combined Solution Error Thu 14 May 26 09:50:11
Processor ms
last feasible point
15
10
5
0
-5
-10
0
5
10
15
20
25


Dembo 1b
Log Relative Combined Solution Error Thu 14 May 26 09:50:11
Processor ms
feasible points only
15
10
5
0
-5
-10
0
5
10
15
20
25