Implementation of High Precision Arithmetic in the BFGS Method for Nonsmooth Optimization

Allan Kaku

New York University, Courant Institute of Mathematical Sciences

251 Mercer Street

New York, NY 10012

January 2011

A thesis submitted in partial fulfillment
of the requirements for the degree of
master's of science
Department of Mathematics
New York University
January 2011

Advisor: Michael L. Overton

Abstract

The results of this thesis investigate the claim that the effectiveness of the BFGS variable metric (quasi-Newton) method with an inexact weak Wolfe line search for minimizing nonsmooth, nonconvex functions is limited by machine precision, rather than defects in the algorithm itself. To investigate this question, the algorithm was implemented using double-double precision, i.e. twice as precise as floating-point double. The double and double-double precision implementations were tested on Nesterov's Chebyshev-Rosenbrock functions, which are known to be difficult to solve. Results for these and other test functions for both the normal and high-precision BFGS codes are presented.

Contents

List of Figures

List of Tables

1 Introduction

The applicability of the Broyden-Fletcher-Goldfarb-Shanno (BFGS) method [NW06] is in solving the unconstrained optimization problem

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

BFGS is an iterative method that generates a sequence of points to find a local minimizer x^*, which may not be a global minimizer due to nonconvexity of f. In this thesis, the function f is only assumed to be continuous, and not continuously differentiable. For an introduction to algorithms for solving nonconvex, nonsmooth optimization problems, see [Kiw85].

In work by Lewis and Overton [LO10] and Skajaa [Ska10], BFGS and the limited memory variation LBFGS were shown to work well in solving nonsmooth test problems, with the primary limitation apparently being machine precision. By implementing a high-precision version of the BFGS solver, this thesis investigates the claim that the algorithms are unable to solve difficult problems because of rounding errors, rather than because of a breakdown in the algorithm itself.

Section 2 will give a brief discussion of the BFGS method. In section 3, a description of double-double high-precision arithmetic is provided, as developed by David Bailey in [Bai10], and distributed by Bailey and Xiaoye (Sherry) Li. Section 4 details the implementation process of incorporating double-double into the BFGS code, using the C code written by Skajaa [Ska10], explaining the changes made and their implications for the results. Section 5 describes the test functions that were used to compare the double and double-double versions, including Nesterov's Chebyshev-Rosenbrock functions. Finally, section 6 gives the results of the tests, comparing the correctness and accuracy of the outputs, as well as the time taken.

A website[1] is being developed with the BFGS code used in this thesis, which includes both the double precision version written by Anders Skajaa and the new double-double version of the code. The double-double version of BFGS uses C++, in order to allow the use of classes with function and operator overloading. In addition, the test problems used to verify the algorithms and the Nesterov functions are available in both double and double-double precision.

2 BFGS

BFGS is the most popular quasi-Newton update formula. Specifically, we will focus on the class of line-search methods that use the search direction defined by

$$d^{(j)} = -C_j \nabla f(x^{(j)})$$

[1] http://cims.nyu.edu/~aak357/

where C_j, an approximation to the inverse Hessian $\nabla^2 f(x^j)$, is updated by a quasi-Newton update formula with certain properties. The BFGS update formula is

$$C_{j+1} = (I - p_j s^{(j)}(y^{(j)})^T) C_j (I - p_j y^{(j)}(s^{(j)})^T) + p_j s^{(j)}(s^{(j)})^T$$

where

$$p_j = ((y^{(j)})^T s^{(j)})^{-1}$$

and

$$s^{(j)} = x^{(j+1)} - x^{(j)}$$
$$y^{(j)} = \nabla f(x^{(j+1)}) - \nabla f(x^{(j)}).$$

When f is continuously differentiable, the BFGS method exhibits superlinear convergence after enough iterations, under standard regularity assumptions. See Nocedal and Wright [NW06] for more details.

The Limited-Memory BFGS method, or LBFGS, is less computationally intensive, requiring $\mathcal{O}(mn)$ floating point operations per update rather than the $\mathcal{O}(n^2)$ floating point operations required for BFGS, where for the first m iterations, LBFGS and BFGS generate the same search directions. The best choice of m is problem-dependent, and the method may require more iterations to converge than BFGS. Tests using LBFGS were not completed in this thesis; however similar experimentation is planned for the future.

In dealing with the nonsmooth case, BFGS typically succeeds in finding a local minimizer, despite being developed for smooth functions. However, the right line search must be used, namely, a weak Wolfe line search [LO10, Ska10]. The rate of convergence in the nonsmooth case is typically linear, not superlinear.

In our tests, the methods begin with a random starting point, unless otherwise specified. A set of points are generated using the `rand` and overloaded `rand` function in C/C++ and normalizing the output between $[-1, 1]$. Thus, the starting point $x^{(0)} \in \mathbb{R}^n$ is drawn randomly from the uniform distribution on $[-1, 1]^n$.

3 High Precision Arithmetic

High precision computing has existed for many years, and as such there are several implementations and variations in different computing languages. In MATLAB, for instance, the Symbolic Math Toolbox contains a function called Variable Precision Arithmetic, or `vpa`, which allows users to specify the number of digits of accuracy they desire[2]. MATLAB automatically ensures that the results are precise to at least the number specified. While this allows high precision computing, it is slow due to the method by which the extra precision computations are performed, as well as the general slower nature of MATLAB in executing code.

[2] http://www.mathworks.com/help/toolbox/symbolic/vpa.html

In other programming languages, many high precision libraries have been created. They can be split into two distinct groups, based on how the precision is represented. One group uses a multiple-digit format, which stores a sequence of digits or bits with a single exponent. The group we will focus on is a multiple-component format, which stores values as an unevaluated sum of normal precision numbers. The advantage of this second representation is that calculations are done much faster. Computations are done on a component level, and then combined at the end to gain the extra precision. Thus, calculations are on the order of magnitude of normal machine precision. This idea was apparently originally suggested by Priest [Pri91]. Two examples of this multiple-component format are double-double and quad-double precison floating point arithmetic, implemented by Hida, Li, and Bailey [LHB01]. As fixed precision libraries, both run faster than other arbitrary precision libraries. We will be focusing on the double-double library.

3.1 Double-double

A double-double number is the sum of two IEEE double format numbers [Ove01]. The double-double number (a_0, a_1) represents the exact sum $a = a_0 + a_1$. The first number, a_0, is the most significant component, with a_1 providing the extra precision. Since a 64 bit IEEE double format number uses 1 bit for the sign, 11 bits for the exponent and 52 bits for the significand, and since there is an additional "hidden bit" for normalized numbers as explained in [Ove01], it effectively has 53 bits of precision, or approximately 16 decimal digits. An example of a double-double number would be $\pi + e * 10^{-16}$, namely $3.14159265358979351029082622918449e+00$, which has 106 bits of precision or approximately 32 digits. In fact, because of the binary nature of the representation, the number $1 + 2^{-1000}$ can be stored *exactly* as a double-double number, since both the numbers 1 and 2^{-1000} are exact floating point double format numbers. Although one might say that such a number has more than 106 bits of precision, this is somewhat artificial as most of the internal zero bits are not stored, and if arithmetic were done using such numbers, the least significant bits would be lost rapidly. Thus, double-double numbers effectively have 106 bits of precision, or about 32 digits, somewhat less than if they were stored using a 128-bit floating point word, but the advantage is that arithmetic on double-doubles can be done rapidly using standard double-format hardware.

This representation was implemented in C++ by creating a new class, overloading the arithmetic and comparative operators and standard mathematical functions [LHB01]. In addition, several constants and random number generators were implemented. The source code and documentation is all available open source at http://crd.lbl.gov/~dhbailey/mpdist/. The necessary libraries and header files to use double-double are already packaged with BFGS in the high-precision version of the code on the companion website to this thesis.

The algorithms to execute the operations between double-doubles is complex, and for a complete explanation, consult [LDB+02]. The double-double addition algorithm

```
void ddadd(double dda[2], double ddb[2], double ddc[2])
{
    /* Compute double-double = double-double + double-double
       (ddc = dda + ddb). */
    /* dda[0] represents high-order word, dda[1] represents
              low-order word. */
    double bv;
    double s1, s2, t1, t2;

    /* Add two high-order words. */
    s1 = dda[0] + ddb[0];
    bv = s1 - dda[0];
    s2 = ((ddb[0] - bv) + (dda[0] - (s1 - bv)));

    /* Add two low-order words. */
    t1 = dda[1] + ddb[1];
    bv = t1 - dda[1];
    t2 = ((ddb[1] - bv) + (dda[1] - (t1 - bv)));

    s2 += t1;

    /* Renormalize (s1, s2)  to  (t1, s2) */
    t1 = s1 + s2;
    s2 = s2 - (t1 - s1);

    t2 += s2;

    /* Renormalize (t1, t2)  */
    ddc[0] = t1 + t2;
    ddc[1] = t2 - (ddc[0] - t1);
}
```

Figure 3.1: *Algorithm for double-double addition.*

shown in Figure 3.1 requires 20 floating-point operations, and therefore runs about an order of magnitude slower than double arithmetic.

4 Implementation

We began with a MATLAB code for BFGS developed M. Overton as a part of the HANSO (Hybrid Algorithm for Nonsmooth Optimization) package[3]. The process of incorporating the double-double class into the source code for BFGS went through several iterations. These alterations to the original MATLAB code are detailed below, as well as their impact on the algorithm's behavior and results.

4.1 MATLAB to C

The first translation to the original MATLAB code to C was done by A. Skajaa. This initial translation of the HANSO package into C resulted in a much faster running version, although it was still partly dependent on MATLAB to run. While the BFGS code and its required functions were translated to C, the HANSO code still ran through MATLAB, using MEX as an interface to call the C code. Furthermore, the test functions were also called through MATLAB. In order to properly implement the

[3] http://www.cs.nyu.edu/faculty/overton/software/hanso/

double-double C++ library, a stand-alone version BFGS, and eventually HANSO, would need to be created.

Taking just the BFGS source code, the program was stripped down to only its essential components. This included implementing a function pointer to the various test functions, now translated into C. To simplify the program and make it more portable, it was necessary to download and dynamically link the necessary libraries and header files from CLAPACK[4] and CBLAS[5] that are required to do the matrix and vector operations. An executable was also created in order to verify the correct translation of the test functions and alterations made to the source code, as well perform the tests for this thesis. A similar translation can be, but has not been yet, implemented in the Gradient Sampling code, which is another component of HANSO.

In addition, the test functions were translated from MATLAB to C, with the T functions translated by A. Skajaa, and the F functions and Nesterov functions translated by the author, who also took responsibility for the debugging of all the code.

4.2 Elimination of CBLAS and CLAPACK

Although we now had a version of BFGS in C, the double-double library could not yet be added. A majority of the major calculations within BFGS were performed, for efficiency and speed, using the optimized algebraic operations provided by CBLAS and CLAPACK. While useful in both its ease of implementation and its recognition as a standard in numerical computing, it took away the control over the calculations, which was necessary in adding double-double. Although Li has been developing XBLAS, an Extended and Mixed precision BLAS standard [LDB+02], it does not yet include all the necessary operations for BFGS. Thus, in order to fully implement double-double precision in all the calculations, the function calls to CBLAS and CLAPACK were replaced with hard-coded operations that executed the necessary calculations. While resulting in slower runtimes, as these codes were not at all optimized, they paved the way for the use of double-double precision. In addition, these changes simplified the program and made it more portable by eliminating the need for the CBLAS and CLAPACK libraries and header files.

In addition, the QP stopping criterion in BFGS [LO10] was removed. The main purpose of the criterion was to stop the code if the solution was only minimally improved with further iterations, thus reducing the running time of the program. However, with the anticipation of using higher precision, allowing slight improvements on each iteration would be required in order for the extra precision to affect the output. Therefore, the QP stopping criterion was commented out of the code in order to test the increased accuracy that double-double provides. It is recommended it remains excluded when running any functions using double-double precision.

[4] http://www.netlib.org/clapack/
[5] http://www.netlib.org/blas/

4.3 C to C++ and overload functions

While C and C++ code are for the most part completely compatible with one another, they differ most importantly in the functionality of classes in C++. Li's double-double library is primarily written in C++ using a special class called dd_real to represent double-double numbers, but also contains low-level C implementations of the operations, without the use of classes. Rather than creating a new class, the C version stores each number in an array of size two, which represents the double-double number, as in Figure 3.1 on page 9. Implementing this complicated method of arrays and low-order functions, while possible, made the code unnecessarily long and convoluted. A simpler approach was to first convert the BFGS C source code to C++. A few changes were made in order for this to be possible, namely substituting the use of the `new` operator in lieu of `malloc`, as well as the changing of a few header files and print statements. In general, the source code was not changed in any significant way during the translation.

Once compiled in C++, the double-double libraries[6] were linked and the necessary header files added. Because of the use of classes and overloaded functions, changing the precision of the necessary variables was as simple as declaring them of type dd_real instead of double. Once a few initial tests were run on simple functions, and the program was verified as operating correctly, more comprehensive tests were made on less trivial functions. All tests were performed using the same random starting points for the double and double-double versions in order to create meaningful comparisons.

5 Test Functions

In order to compare the performance of the double (D) precision code to the double-double (DD) precision code, the algorithms were implemented on several test functions.

5.1 F Functions

The first set of test functions are nine problems from [HMM04], denoted F1-F9, and are all nonsmooth at the minimizer. The first five F1-F5 are convex and the other four F6-F9 are nonconvex. They are defined as follows.

F1: Generalization of MAXQ

$$f(x) = \max_{1 \le i \le n} x_i^2$$

[6] Actually, the quad-double libraries were used, which includes the double-double class.

F2: Generalization of MAXHILB

$$f(x) = \max_{1 \le i \le n} \left| \sum_{j=1}^{n} \frac{x_j}{i+j-1} \right|$$

F3: Chained LQ

$$f(x) = \sum_{i=1}^{n-1} \max \left\{ -x_i - x_{i+1}, -x_i - x_{i+1} + (x_i^2 + x_{i+1}^2 - 1) \right\}$$

F4: Chained CB3 I

$$f(x) = \sum_{i=1}^{n-1} \max \left\{ x_i^4 + x_{i+1}^2, (2 - x_i)^2 + (2 - x_{i+1})^2, 2e^{-x_i + x_{i+1}} \right\}$$

F5: Chained CB3 II

$$f(x) = \max \left\{ \sum_{i=1}^{n-1} (x_i^4 + x_{i+1}^2), \sum_{i=1}^{n-1} ((2 - x_i)^2 + (2 - x_{i+1})^2), \sum_{i=1}^{n-1} (2e^{-x_i + x_{i+1}}) \right\}$$

F6: Number of Active Faces

$$f(x) = \max_{1 \le i \le n} \left\{ g\left(-\sum_{j=1}^{n} x_j \right), g(x_i) \right\}, \quad \text{where } g(y) = \ln(|y| + 1)$$

F7: Nonsmooth Generalization of Brown Function 2

$$f(x) = \sum_{i=1}^{n-1} \left(|x_i|^{x_{i+1}^2 + 1} + |x_{i+1}|^{x_i^2 + 1} \right)$$

F8: Chained Mifflin 2

$$f(x) = \sum_{i=1}^{n-1} \left(-x_i + 2\left(x_i^2 + x_{i+1}^2 - 1\right) + \frac{7}{4} \left| x_i^2 + x_{i+1}^2 - 1 \right| \right)$$

F9: Chained Crescent I

$$f(x) = \max\left\{\sum_{i=1}^{n-1}\left(x_i^2 + (x_{i+1}-1)^2 + x_{i+1} - 1\right), \sum_{i=1}^{n-1}\left(-x_i^2 - (x_{i+1}-1)^2 + x_{i+1} + 1\right)\right\}$$

Through experiments, it has been determined that, apparently, none of these problems have multiple local minima for $n \leq 200$ [Ska10]. All functions were tested with $n = 10, 50, 200$ variables using random starting points for 10 runs on each code.

5.2 T Functions

The second set of test problems were five functions taken from [LTS+02], denoted T1-T5. All are again nonsmooth and can be defined with any number of variables. The first two, T1 and T2, are convex. The problems are defined as follows.

T1: Problem 2 from `TEST29`

$$f(x) = \max_{1 \leq i \leq n} |x_i|$$

T2: Problem 5 from `TEST29`

$$f(x) = \sum_{i=1}^{n}\left|\sum_{j=1}^{n}\frac{x_j}{i+j-1}\right|$$

T3: Problem 6 from `TEST29`

$$f(x) = \max_{1 \leq i \leq n} \left|(3-2x_i)\,x_i + 1 - x_{i-1} - x_{i+1}\right|, \quad \text{with } x_0 = x_{n+1} = 0$$

T4: Problem 11 from `TEST29`

$$f(x) = \sum_{k=1}^{2(n-1)} |f_k(x)|$$

where

$f_k(x) = x_i + x_{i+1}\left((1+x_{i+1})\,x_{i+1} - 14\right) - 29 \quad \text{if } \mod(k,2) = 0$

$f_k(x) = x_i + x_{i+1}\left((5-x_{i+1})\,x_{i+1} - 2\right) - 13 \quad \text{if } \mod(k,2) = 1$

$i = \lfloor (k+1)/2 \rfloor$

T5: Problem 22 from `TEST29`

$$f(x) = \max_{1 \le i \le n} \left| 2x_i + \frac{1}{2(n+1)^2} \left(x_i + \frac{i}{n+1} + 1 \right)^3 - x_{i-1} - x_{i+1} \right|$$

All functions were tested for $n = 10, 50, 200$ variables using random starting points for 10 runs on each code.

5.3 Nesterov's Chebyshev-Rosenbrock Functions

The three Nesterov Chebyshev-Rosenbrock (NCR) functions are all very difficult to solve. For reasons that are only partially understood, the functions are exponentially difficult in n. This makes these the most challenging test functions for our double-double BFGS solver.

The smooth function, which we will denote as NCR-S, is:

$$\tilde{f}(x) = \frac{1}{4}(x_1 - 1)^2 + \sum_{i=1}^{n-1} \left(x_{i+1} - 2x_i^2 + 1 \right)^2.$$

The first nonsmooth variation, which we will denote as NCR-NS1, is:

$$\hat{f}(x) = \frac{1}{4}(x_1 - 1)^2 + \sum_{i=1}^{n-1} \left| x_{i+1} - 2x_i^2 + 1 \right|.$$

The second nonsmooth variation, which we will denote as NCR-NS2, is:

$$f(x) = \frac{1}{4}|x_1 - 1| + \sum_{i=1}^{n-1} |x_{i+1} - 2|x_i| + 1|.$$

In all three cases, the only local minimizer is $\bar{x} = [1, 1, 1, ..., 1]^T$.

For the smooth variant, the starting point used is $\hat{x} = [-1, 1, 1, ..., 1]^T$. This point is contained in the manifold

$$M = \{x : x_{i+1} - 2x_i^2 + 1 = 0, \quad i = 1, ..., n-1\}$$

which also contains the minimizer $\bar{x}$. Because $\tilde{f}$ is the sum of a quadratic term and a nonnegative sum whose zero set is the manifold M, minimizing $\tilde{f}$ is equivalent to minimizing its first quadratic term on M, and it is easy to see that the only stationary point of $\tilde{f}$ is the global minimizer $\bar{x}$. Due to the manifold's highly oscillatory nature, BFGS requires many iterations to converge to the minimizer.

The first nonsmooth variation is nondifferentiable at points in M, including the point $\hat{x}$, but as explained in [GO10], it is "partly smooth" with respect to the manifold M

in the sense of [Lew03]. See Figure 5.1 (left) for a contour plot for $n = 2$, where the line segments show the iterates of BFGS from 7 random starting points. Minimizing this function is similarly equivalent to minimizing its first quadratic term on M, and again it has only one stationary point, namely the minimizer $\bar{x}$. The starting point x is chosen randomly from the uniform distribution between -1 and 1.

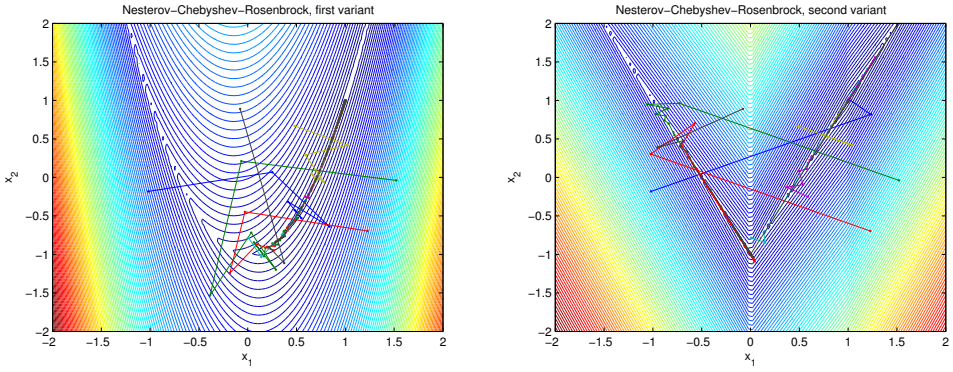

Figure 5.1: *Contour plot for NCR-NS1 (left) and NCR-NS2 (right), n=2. The line segments show the iterates of BFGS initialized from 7 random starting points.*

The second variation is not only nonsmooth, but also non-regular in the sense of Clarke (see [GO10]). Minimizing f is equivalent to minimizing its first term on the set S, where

$$S = \{x : x_{i+1} - 2\,|x_i| + 1 = 0, \quad i = 1, ..., n-1\}.$$

The set S is again highly oscillatory, and contains "corners", resulting in its non-regularity. See Figure 5.1 (right) for a contour plot for $n = 2$, again showing the iterates of BFGS initialized from 7 random starting points. In fact, for a given n, the NCR-NS2 function has 2^{n-1} Clarke stationary points, of which all except the global minimizer $\bar{x}$ are "corners" of S [GO10]. The starting point x is again chosen randomly from the uniform distribution between -1 and 1.

In all cases, the BFGS inverse Hessian approximation matrix was initialized as the identity matrix. NCR-S was tested with $n = 2, ..., 10$ variables, NCR-NS1 was tested with $n = 2, ..., 7$ variables, and NCR-NS2 was tested with $n = 2, ..., 8$ variables. BFGS was intialized at $\hat{x}$ for NCR-S and at randomly generated starting points for the 1000 runs of NCR-NS1 and NCR-NS2.

6 Results and Comparison of Performance

To fully test the capabilities of the BFGS algorithm with higher precision, we terminate the program only when it breaks down because of rounding errors due to

machine precision. That is, all parameters are set such that the termination only occurs when a search direction is not a descent direction ($\nabla f(x^{(j)})^T p^{(j)} < 0$) or a reduction is not obtained in the line search (the limit on the number of bisection or expansion steps, 100, is exceeded). For F1-F4 and F8, $n = 200$, the limit on the number of iterations was set to 10^5, and for NCR-NS1, $n = 6, 7$, the limit on the number of iterations was set to 10^{n-1}, so that the tests would run in a reasonable amount of time. In all other tests, the iteration limit was set large enough that it did not cause termination.

All tests were done using a dual-core 64-bit Intel Pentium Processor SU4100, 1.3 GHz, 2 MB cache, and 3 GB DDR3 memory, running Ubuntu 10.04.

6.1 F Functions Results

The minimum and maximum final values found by the D and DD codes over the 10 randomly generated starting points on the F functions are shown in Tables 1, 2, and 3 for $n = 10, 50$, and 200, respectively. Also shown is the optimal value f^* to double-double precision, when it is known.

		D		DD	
Problem #	f^*	min	max	min	max
F1	0	2.0066500862662096e-92	1.7598318660798305e-75	1.8973357484890915473252577036497e-231	2.2965319420333477537721025563416 3e-187
F2	0	7.4592726093114514e-10	6.0591901898665590e-08	9.3601198295764785329921388501 4954e-16	2.4196201938062616647008546581 4475e-14
F3	-1.2727922061357855439215198517887e+01	-1.2727922061357855e+01	-1.2727922061357855e+01	-1.2727922061357855439215198517886 5e+01	-1.2727922061357855439215198517887 3e+01
F4	18	1.8000000000000007e+01	1.8000000000000039e+01	1.8000000000000000000000000000005e+01	1.8000000000000000000000000000021e+01
F5	18	1.8000000000000000e+01	1.8000000000000007e+01	1.8000000000000000000000000000000e+01	1.8000000000000000000000000000001e+01
F6	0	1.3322676295501871e-15	1.1324274851176532e-14	1.5942199684643182922254712949269 4e-32	1.3674705228786941505300013159431 8e-31
F7	0	2.0264389251848834e-15	9.0434354618988751e-15	6.6778217627730076573265834390018 3e-31	7.0318556039522038976204707072486 2e-31
F8	-	-6.5146142106623506e+00	-6.5146142106776281e+00	-6.5146142106776417143337911290874 3e+00	-6.5146142106776417143337911301330 7e+00
F9	0	-1.1102230246251565e-16	4.4408920985006262e-16	-3.7728666683992139855764567898685 2e-33	1.1444688231244243803984128925534 2e-31

Table 1: *Minimum and maximum final values found by the D and DD codes for the F functions, $n = 10$, and the optimal value.*

		D		DD	
Problem #	f^*	min	max	min	max
F1	0	4.1109297512990274e-78	5.7156919175156493e-72	4.1619304039041066068144981150454 9e-206	1.4443557341931123544149899796 7932e-180
F2	0	9.5770040422354930e-10	2.3555088200918547e-07	7.3216186390741068879597013759407 2e-15	2.5595290629588393452739755739396e-13
F3	-6.9296464556281657391282747486275+01	-6.9296464556281350e+01	-6.9296464556281592e+01	-6.9296464556281657391282747486254 0e+01	-6.9296464556281657391282747486276 7e+01
F4	98	9.8000000000000099e+01	9.8000000000000512e+01	9.8000000000000000000000000000114e+01	9.8000000000000000000000000000183e+01
F5	98	9.7999999999999986e+01	9.8000000000000540e+01	9.7999999999999999999999999999996e+01	9.8000000000000000000000000000154e+01
F6	0	1.1546319456101562e-14	1.9628743075370842e-13	2.3671260788658168521416821333198 5e-31	1.5387256100154726691771300425493 6e-30
F7	0	2.4364638592174621e-14	5.4037965138567655e-14	2.7089208872895293586791885641988 6e-30	7.7109074367775245899713125377729 6e-30
F8	-	-3.4795181409548288e+01	-3.4795181409546011e+01	-3.4795181409547642983703577599499 2e+01	-3.4795181409547642983703577760804 50e+01
F9	0	0	2.8865798640254070e-15	3.8518598887744717061119558851698 7e-33	1.1931136005478926109681783354313 7e-31

Table 2: *Minimum and maximum final values found by the D and DD codes for the F functions, $n = 50$, and the optimal value.*

		D		DD	
Problem #	f^*	min	max	min	max
F1	0	5.2745683191926849e-43	3.8387659508038081e-40	3.7737491967416662386234131993102 0e-43	3.6726547391489082230124031303456 9e-40
F2	0	1.4803212635729202e-09	9.7432916068129066e-08	2.1815668036739972169576944481163 3e-14	4.0875788128871963440369303138117 66e-13
F3	-2.8142849891224591471153605611773e+02	-2.8142849891224233e+02	-2.8142849891224591e+02	-2.8142849891224591471153605611715 5e+02	-2.8142849891224591471153605611759 9e+02
F4	398	3.9800000000000233e+02	3.9800000000000495e+02	3.9800000000000000000000000000093e+02	3.9800000000000000000000000000123e+02
F5	398	3.9800000000000006e+02	3.9800000000000398e+02	3.9800000000000000000000000000001e+02	3.9800000000000000000000000000244e+02
F6	0	6.6391336872582152e-14	1.5241141682042535e-12	8.5498282348741463414042589122906 e-31	6.5663421772668169514638804967808 6e-30
F7	0	2.4244868800304813e-13	3.3815857155322334e-13	2.8462982312040926459180918974949 5e-29	2.1970160346648511916709476817069 3e+01
F8	-	-1.4086070716519524e+02	-1.4086070717278730e+02	-1.4086070717286944204096568730215 6e+02	-1.4086070717286944204096573324482 9e+02
F9	0	0	6.4392935428259079e-15	-6.6270045680149543681248041018133 2e-32	1.5457513733651954956272789671866 e-30

Table 3: *Minimum and maximum final values found by the D and DD codes for the F functions, $n = 200$, and the optimal value.*

For most of the F functions, the D code of BFGS was able to find the minimizer fairly well when $n = 10$, as demonstrated in Skajaa [Ska10]. The DD code did converge to a lower final function value, reflecting the different machine precision, about 10^{-16} for D and 10^{-32} for DD. The exception was for F2, for which the D code only reduced the function value to 10^{-10}, while the DD code was able to reduce the function value to 10^{-16}. The easiest problem for either code to solve was F1, as seen by the final function values being as low as 10^{-92} and 10^{-231} for the D and DD code, respectively.

When $n = 50$, both codes exhibited the same behavior as $n = 10$. For $n = 200$, the problems were more difficult for either code to solve. On F1, both codes reduced the final function value to around 10^{-43}, significantly higher than with fewer variables. Most interesting was the DD code's final function values on F7, ranging from 10^{-29} to 22, while the D code always converged around 10^{-13}. The reason why the DD code sometimes fails in solving F7 for $n = 200$ needs further investigation.

6.2 T Functions Results

The minimum and maximum final values found by the D and DD codes over the 10 randomly generated starting points on the T functions are shown in Tables 4 and 5 on this page, and 6 on the following page for $n = 10, 50$, and 200, respectively. Also shown is the optimal value f^* to double-double precision, when it is known.

		D		DD	
Problem #	f^*	min	max	min	max
T1	0	1.46006581632566 93e-16	8.2340403956125942e-16	1.7319290353081586316872323744 1175e-32	7.9796625542989061100873062063 4931e-32
T2	0	6.7428593524682112e-10	1.2019196891617859e-07	2.4627338486006592512567484559 4735e-15	2.0895915232508262351277832356 5036e-13
T3	0	3.2196467714129540e-15	4.4617856727674143e-01	2.8349688781380111756983995314 8502e-31	4.4617667100439855508856509451 3944e-01
T4	-	1.0605911852062579e+02	1.0605911852062738e+02	1.0605911852062563282380321145 7878e+02	1.0605911852062563282380321145 7957e+02
T5	0	7.2164496600635175e-16	7.5495165674510645e-15	6.1629758220391547297791294162 7187e-32	7.9001646318764414692356215204 8337e-31

Table 4: *Minimum and maximum final values found by the D and DD codes for the T functions, $n = 10$, and the optimal value.*

		D		DD	
Problem #	f^*	min	max	min	max
T1	0	1.4601138701364973e-15	5.3033915333941507e-15	1.0407705240373059788231318846613 5e-31	4.7821314178333594462449245221 8446e-31
T2	0	8.1392709499333860e-09	3.2645285200950946e-07	7.3842204950014258120976255840930 5e-15	1.3269993655297300294675164560031 8e-12
T3	0	5.2624571367232420e-14	5.0598160442741924e-01	1.7194702543489241696083771071398 2e-30	5.0597365136433599065357712915 3149e-01
T4	-	5.6317033620503435e+02	5.8799776162067542e+02	5.6317033620499873913045587092745 7e+02	5.8799776162066962575628236203708 2e+02
T5	0	5.9396931817445875e-15	2.5079938126282286e-13	5.4080112838393582753811860627784 8e-31	1.1043667487105287828595358718370 4e-29

Table 5: *Minimum and maximum final values found by the D and DD codes for the T functions, $n = 50$, and the optimal value.*

		D		DD	
Problem #	f^*	min	max	min	max
T1	0	1.0620278236118440e-14	2.1619668381963696e-14	8.7471451612260629784855574431308 7e-31	1.9610032186937223966575714202999 7e-30
T2	0	1.3242105659618322e-08	3.4949545577428036e-07	7.3602157364363111028663526985472 1e-14	1.2151620298039549752517836826255 7e-12
T3	0	1.4343876061581251e-01	4.6214915356057096e-01	7.7540365900825017861372975066702 6e-02	4.6207239912154168188222470066306 3e-01
T4	-	2.2378539839989403e+03	2.3952676732459868e+03	2.2378539839988736768415726358594 0e+03	2.3952676732458579765123051111530 0e+03
T5	0	1.2307932450994485e-12	1.0875109146546436e-10	4.7080513048512612769465214393254 1e-29	7.2135243843837265939606920404812 2e-28

Table 6: *Minimum and maximum final values found by the D and DD codes for the T functions, $n = 200$, and the optimal value.*

For T1 and T5, the D and DD codes worked as we expected, reducing the final function values close to machine precision, around 10^{-16} for D and 10^{-32} for DD. On T2, the codes reduced the final function value to slightly larger values, but still close to the minimal value. However, the results for T3 and T4 were quite different, with the final function values varying quite widely over the 10 runs. For this reason, we decided to run both codes on T3 and T4 for $n = 10$ and 50 with 1000 random starting points[7], graphing the results with the DD values sorted and plotted with the corresponding D value for the same starting point, as shown in Figures 6.1 and 6.2 on the following page.

It seems that for both T3 and T4, there may be multiple local minima. For T3, we know the global minimum value is 0, and while the method sometimes finds this value in 10 runs for $n = 10$ and $n = 50$, for $n = 200$, 10 runs are not enough. As can be seen in Figure 6.1, 1000 runs of T3 result in several plateaus, suggesting the presence of multiple local minima. For T4, we do not know the global minimum value, which changes with the number of variables. When $n = 10$, all 10 runs on T4 find the same value, and for $n = 50$ and 200, 10 runs are enough to sometimes find what may be the minimum value. However, over 1000 runs, it can be seen in Figure 6.2 on the following page that there are several plateaus, suggesting multiple local minima. Further investigation is required to prove there are in fact other local minima.

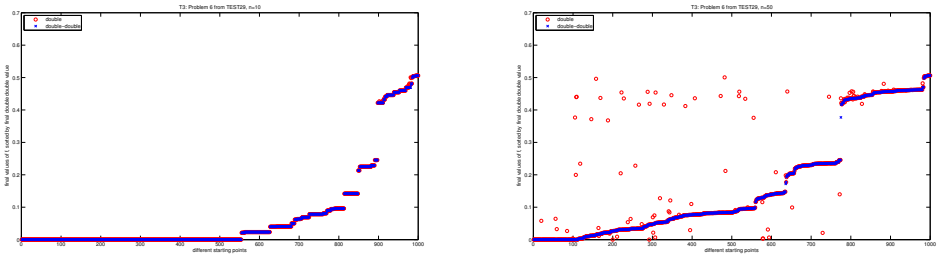

Figure 6.1: *T3 performance plots for $n = 10$ (left) and $n = 50$ (right), with the final DD function values sorted and plotted with the corresponding final D function value for 1000 random starting points. It appears that there are multiple local minima.*

[7] Tests on T3 and T4 for $n = 200$ were not completed because of the extraordinarily long amount of time one run of the DD code takes to run before breakdown due to machine precision.

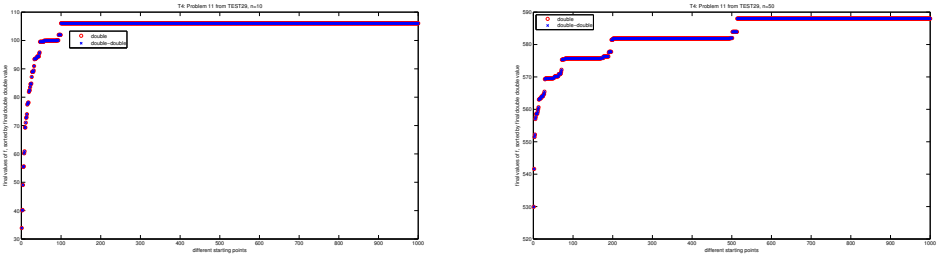

Figure 6.2: *T4 performance plots for $n = 10$ (left) and $n = 50$ (right), with the final DD function values sorted and plotted with the corresponding final D function value for 1000 random starting points. It appears that there are multiple local minima.*

6.3 Nesterov Functions Results

The three Nesterov functions are all very different from one another, and are thus affected differently by the implementation of higher precision.

The smooth function NCR-S is differentiable and we find, for $n = 10$, that using the D code we can reduce the gradient norm to about 10^{-15}, while using the DD code, with only a few more iterations, we can reduce it to about 10^{-161} before breakdown occurs. These gradient norm values correspond to final function values of approximately 10^{-31} and 10^{-323} for D and DD, respectively. As shown for $n = 10$ in Table 7 on the next page, the D and DD codes generate function values that agree to double precision accuracy for many iterations, but eventually they start to diverge. By iteration 1988, they agree to only three digits, and by iteration 55025, they have no significant digits in common, although the order of magnitude is the same for both. The D code terminates at iteration 55036, while the DD code continues running until iteration 55102. The fact that the DD code makes so much improvement to the gradient norm with a relatively small increase in the number of iterations is evidence of superlinear convergence. The fourth and eighth columns show the gradient norm ratios $\frac{\|(\nabla \tilde{f}(x^{(j)})\|}{\|(\nabla \tilde{f}(x^{(j-1)})\|}$, and we see that indeed this ratio converges to zero, as required for superlinear convergence.

Figure 6.3 on page 21 and Figure 6.4 on page 22 shows the final function values found by both the D and DD codes for NCR-NS1 and NCR-NS2, respectively. The final function values found by the DD code have been sorted, and plotted with the corresponding function value the D code finds for the same starting point.

For the first nonsmooth variant NCR-NS1, the DD code overall performs better at converging to the global minimizer than the D code. For $n = 2$, the D and DD codes consistently reduce the final function value to about 10^{-15} and 10^{-30}, respectively, for all 1000 runs. For $n = 3$, the problem is much more difficult and BFGS already has trouble "tracking" the manifold M. As a result, for all starting points the D code is able to reduce the function value only to values in the range 10^{-3} to 10^{-8},

Double				Double-double			
iteration #	f value	gradient norm	gradient norm ratio	iteration #	f value	gradient norm	gradient norm ratio
0	1.0000000000000000e+00	1.00e+00		0	1.00000000000000000000000000000000e+00	1.00e+00	
1	9.9713134765625000e-01	9.76e-01	9.76e-01	1	9.97131347656250000000000000000000e-01	9.76e-01	9.76e-01
2	9.9558682074316829e-01	8.03e-01	8.23e-01	2	9.95586820743168268275978325513396e-01	8.03e-01	8.23e-01
1988	9.7224397108592209e-01	2.31e-02	1.09e+00	1988	9.72437838251788204096169116613658e-01	1.08e-02	2.15e+00
1989	9.7217838105665322e-01	2.25e-02	9.74e-01	1989	9.72430883872208813397005684647731e-01	6.38e-03	5.91e-01
1990	9.7215264554336966e-01	5.84e-02	2.60e+00	1990	9.72404840430861654428482797546735e-01	1.37e-02	2.15e+00
1991	9.7201814908113804e-01	7.78e-02	1.33e+00	1991	9.72387198935592781600786873296769e-01	1.04e-02	7.59e-01
1992	9.7159770041401672e-01	3.30e-02	4.24e-01	1992	9.72373552571879118532169026418070e-01	2.59e-03	2.49e-01
1993	9.7158629322000323e-01	4.25e-02	1.29e+00	1993	9.72368712349502782081137723139568e-01	2.35e-02	9.07e+00
1994	9.7156054349185494e-01	9.06e-03	2.13e-01	1994	9.72353434828202937409674979651386e-01	4.88e-03	2.08e-01
1995	9.7155717855465851e-01	4.81e-03	5.31e-01	1995	9.72320465170916529711941362169960e-01	9.87e-03	2.02e+00
						3.51E-011	
55025	3.8464142897599387e-21	4.54e-11	2.47e+00	55040	1.36016830443403326688507176195698e-21	1.94e-10	5.53e+00
55026	2.5356214550219539e-21	2.62e-10	5.77e+00	55041	8.76081717388940710031385066561499e-22	9.07e-11	4.68e-01
55027	1.5820933700832428e-21	8.55e-11	3.26e-01	55042	2.58982489685111526573726048751922e-22	4.60e-11	5.07e-01
55028	6.2784355786157786e-22	6.01e-11	7.03e-01	55043	7.84836770402995020584598152750246e-23	4.15e-11	9.02e-01
55029	2.5081377338650191e-22	8.74e-11	1.45e+00	55044	1.21985100184319338251598351089778e-23	1.20e-11	2.89e-01
55030	4.5703340141257500e-23	1.22e-11	1.40e-01	55045	5.49455337875079633371246955767164e-24	1.70e-11	1.42e+00
55031	9.6263875386555454e-24	2.51e-11	2.06e+00	55046	1.56333422601228232556369414727610e-25	1.84e-14	1.08e-03
55032	1.1800109413869144e-24	3.09e-12	1.23e-01	55047	1.02987507846577314030418736221096e-26	3.09e-13	1.68e+01
55033	8.7452787152106979e-26	4.50e-13	1.46e-01	55048	6.34700140921867487077260764553929e-29	6.37e-14	2.06e-01
55034	1.2666456058245973e-28	7.67e-14	1.70e-01	55049	4.79600291673943850635842124557142e-31	5.42e-15	8.51e-02
55035	6.1629758220391547e-32	1.83e-15	2.39e-02	55050	6.80255531444838133689387622711984e-35	1.43e-17	2.64e-03
55036	6.1629758220391547e-32	6.48e-15	3.54e+00	55051	2.17490545742709934286578838594657e-38	3.11e-19	2.17e-02
				55052	9.98123840971977145138916247867635e-44	2.48e-21	7.97e-03
				55053	3.95256742643155123146253361021236e-46	1.56e-22	6.29e-02
				55054	1.71957933337100508387652040538676e-47	3.29e-23	2.11e-01
				55055	9.30210226651205099550904426783349e-52	2.46e-25	7.48e-03
				55056	3.48865671761624136653426926686208e-55	4.70e-27	1.91e-02
				55057	7.65910576628363852983449786530361e-60	2.32e-29	4.94e-03
				55058	1.44168300251534811302845390904585e-61	3.30e-30	1.42e-01
				55059	5.11650193786460008497600518330151e-65	5.65e-32	1.71e-02
				55060	4.75534523581623293739021446360475e-67	5.22e-33	9.24e-02
				55061	1.73153691091351950338280925813442e-70	1.14e-34	2.18e-02
				55062	5.50961511707012022526855319175153e-72	2.10e-35	1.84e-01
				55063	4.57581984347376238801132233017522e-75	5.67e-37	2.70e-02
				55064	1.11192140796356968337928611527172e-76	8.60e-38	1.52e-01
				55065	1.13075676088187163362291965632891e-82	9.38e-41	1.09e-03
				55066	3.87532413593686367271638511169429e-86	1.58e-42	1.68e-02
				55067	2.06565695460783815682480358420569e-88	1.06e-43	6.71e-02
				55068	2.12273457334140370095999929245673e-92	1.20e-45	1.13e-02
				55069	3.93595399737738495942440609602142e-96	1.77e-47	1.48e-02
				55070	1.41201933319065215451456872406262e-101	3.09e-50	1.75e-03
				55071	9.70210305370674932782151000024867e-107	7.64e-53	2.47e-03
				55072	1.79563500806363496742421991079174e-112	1.12e-55	1.47e-03
				55073	5.93894871261638218411121350044390e-117	6.05e-58	5.40e-03
				55074	1.02825952229205319490940775355354e-123	2.40e-61	3.97e-04
				55075	9.38652613902593923776992130399695e-130	2.48e-64	1.03e-03
				55076	2.37365131904632234224028663910748e-136	1.21e-67	4.88e-04
				55077	2.87556449140241933593776179039414e-141	4.18e-70	3.45e-03
				55078	6.73913589966158739519868451956139e-150	2.27e-74	5.43e-05
				55079	2.39366099153348236957666796978149e-155	4.08e-77	1.80e-03
				55080	1.79591516857394293733622507747822e-159	3.56e-79	8.73e-03
				55081	2.61739873059889931543308260853175e-166	1.38e-82	3.88e-04
				55082	3.72349878891967750347631374363834e-172	1.55e-85	1.12e-03
				55083	1.84498934861703140797443860575022e-178	1.16e-88	7.48e-04
				55084	3.30509463745221358158184141828516e-184	1.63e-91	1.41e-03
				55085	3.69517250553648843196316867876103e-192	1.61e-95	9.88e-05
				55086	1.37160652884311251229846212189785e-199	2.90e-99	1.80e-04
				55087	3.30198388731871166216586332796718e-209	4.76e-104	1.64e-05
				55088	4.98041929567289835906015000120704e-218	1.93e-108	4.05e-05
				55089	3.85347771371676023719775360073517e-226	1.74e-112	9.02e-05
				55090	6.09746339419566318492243550723581e-234	2.03e-116	1.17e-04
				55091	2.02704047172850065118656344015725e-242	1.15e-120	5.67e-05
				55092	1.16791216498261533584522958264514e-250	9.73e-125	8.46e-05
				55093	7.76478913491329158848942593915328e-260	2.39e-129	2.46e-05
				55094	1.21364827147284398938587392100999e-266	9.74e-133	4.08e-04
				55095	1.36978432195256289924331216545868e-273	3.35e-136	3.44e-04
				55096	1.37973081994056613224180984645614e-280	2.83e-140	8.45e-05
				55097	4.04436751790345677047872488481844e-287	1.92e-143	6.78e-04
				55098	2.32773417327698651522409578511019e-296	1.45e-147	7.55e-05
				55099	6.75549265185909825757444479122310e-305	7.05e-152	4.86e-05
				55100	7.39971014811772129212765303894124e-314	2.25e-156	3.19e-05
				55101	1.48219693752373963252970637860443e-323	3.56e-161	1.58e-05
				55102	1.48219693752373963252970637860443e-323	3.56e-161	1.00e+00

Table 7: *Values of BFGS for double and double-double precision. Superlinear convergence can be seen in the higher iterations, before the program terminates.*

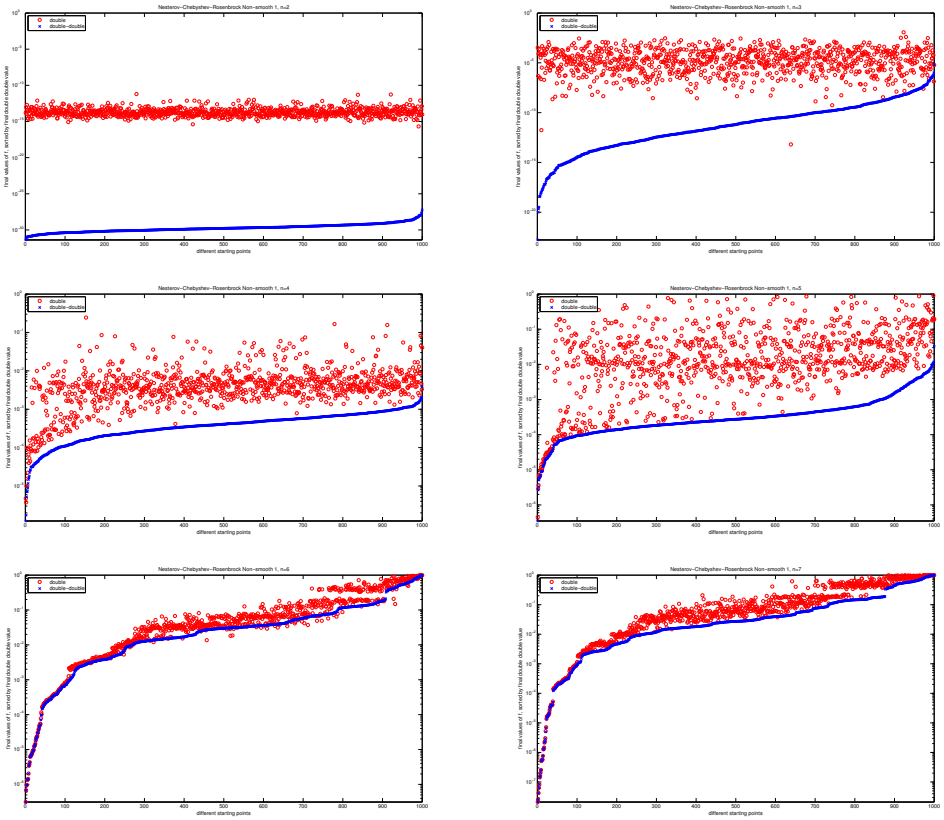

Figure 6.3: *NCR-NS1 performance plots, with the final DD function values sorted and plotted with the corresponding final D function value for 1000 starting points. Top left: Plot for $n = 2$. The DD code reduces the function value to as low as its precision allows. Top right: Plot for $n = 3$. The DD code overall does better than the D code, but with a much larger range. Middle left: Plot for $n = 4$. The DD code again does better than the D code, but final values are much closer. Middle right: Plot for $n = 5$. Both codes do equally well in some runs, and both codes have a large range of final values. Bottom left: Plot for $n = 6$. A limit is placed on the maximum number of iterations for $n \geq 6$. Neither code does very well, with little difference between the two. Bottom right: Plot for $n = 7$. Neither code does very well, with little difference between the two.*

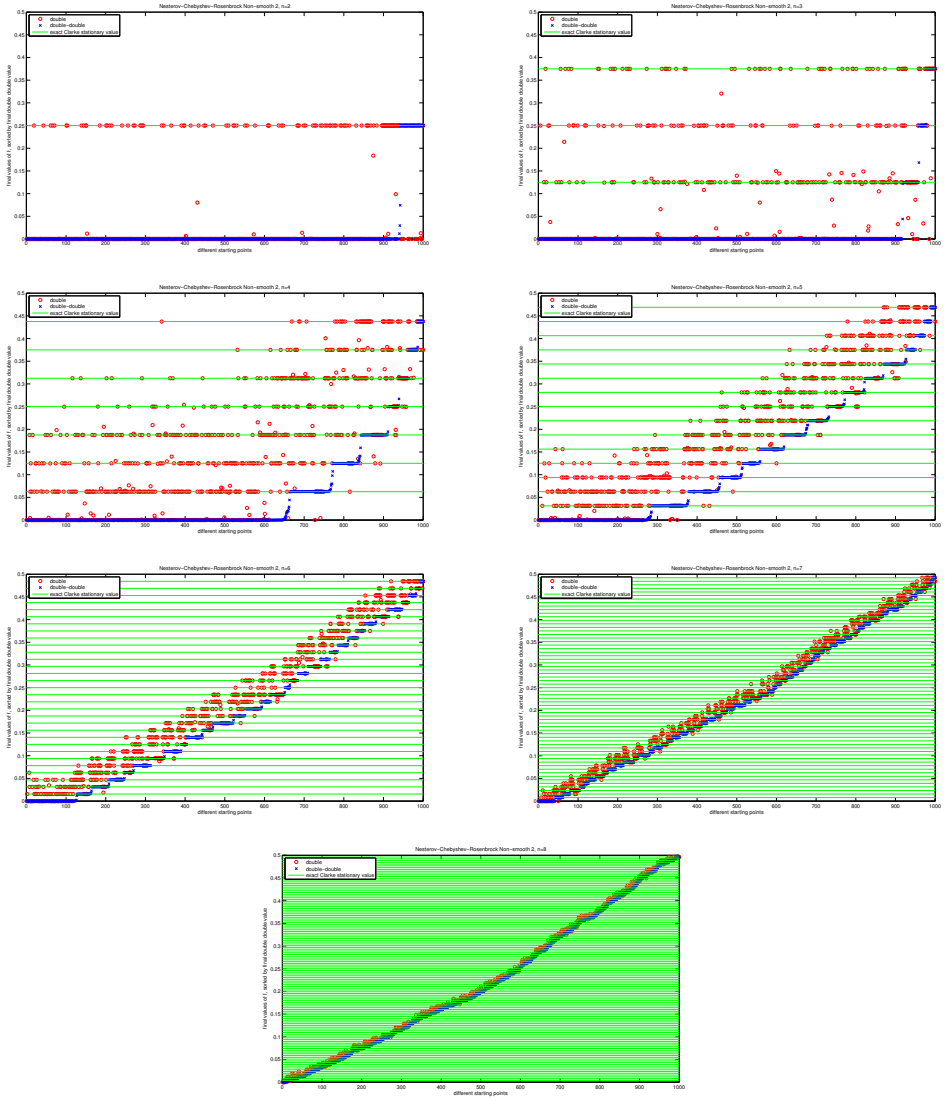

Figure 6.4: *NCR-NS2 performance plots, with the final DD function values sorted and plotted with the corresponding final D function value for 1000 starting points. Top left: Plot for $n = 2$. The DD code finds the minimizer much more often than the D code. Top right: Plot for $n = 3$. The DD code finds the minimizer most of the time, while the D code converges to the other Clarke stationary points more often. Top-Middle left: Plot for $n = 4$. The DD code finds the minimizer rather than the Clarke stationary points much more often than the D code. Top-Middle right: Plot for $n = 5$. The DD code finds a lower stationary point than the D code almost every time, but only finds the minimizer on a quarter of the runs. Bottom-Middle left: Plot for $n = 6$. Neither code finds the minimizer very often, but the DD code still finds a lower stationary point most of the time. Bottom-Middle right: Plot for $n = 7$. Neither code does very well in finding the minimizer a majority of the time, with minimal improvement of the DD code over the D code. Bottom: Plot for $n = 8$. Both codes perform practically the same.*

but the DD code does much better, with final results ranging from 10^{-6} to 10^{-23}. When $n = 4$, both codes do much worse in converging to the global minimizer, but DD still performs better, with the D code final results ranging from 10^{-1} to 10^{-4}, while the DD code reduces the function value to values ranging from 10^{-2} to 10^{-6}. For $n = 5$, the D code final function values vary widely from 1 to 10^{-6}, while the DD code performs slightly better, resulting in a tighter range of function values, 10^{-2} to 10^{-7}. In all the cases for $n = 2, ..., 5$, the DD code does much better than the D code in almost all but a couple runs. The final function values form an "S"-shaped smooth curve, with a long, slowly increasing middle area, presumably reflecting the use of a uniform distribution for the starting points.

When $n = 6$ and $n = 7$, the number of maximum iterations was limited to 10^{n-1}. This was necessary in order to run the tests in a timely manner, as each run of the DD code would run for a couple of minutes before the machine precision causes termination of the code. Under these additional restrictions, the DD code has much less advantage over the D code. More runs of the D code produce the same if not better results. Also, the final function value curve loses its S-shape, looking more like a logarithmic curve. This indicates that although the D and DD codes are able to occasionally reduce the final function value to around 10^{-7}, a majority of the time the algorithm is unable to reduce the final function value below 10^{-3}. When the DD code was allowed to run with as many iterations as needed for $n = 6$, around three to four million iterations were required before the program terminated; however the final function value was still around 10^{-7}. A complete run for NCR-NS1 $n = 7$ was not performed due to the limitations of the machine.

On the second nonsmooth variant NCR-NS2, the DD code does significantly better than the D code. As the plots show, starting from enough starting points BFGS "finds" all the Clarke stationary points, whose f values are marked on the plots by horizontal green lines. Intuitively, BFGS has trouble tracking the set S to the global minimizer because it cannot get past the corners, and as we can see, the algorithm ends up converging to these corners in both the D and DD implementations. For $n = 2$ the DD code reduces the final function value to about 10^{-32} most of the time, where the D version reduces it to about 10^{-16} noticeably less often. In general, both codes do well in approaching the global minimizer under their respective precision. For $n = 3$, the DD code reduces the final function value to about 10^{-31} a large majority of the time, however it also "finds" all the other Clarke stationary points. In contrast, the D code reduces the function value to 10^{-15} only about two-thirds of the time, converging to the other Clarke stationary points much more often.

For $n = 4$ we start to see a big difference in performance between the two codes. The DD code reduces the final function value to 10^{-31} over half of the time, while the D code reduces it to 10^{-15} only about a quarter of the time. The DD code still finds all the other Clarke stationary points, but the higher the corresponding function value, the less likely the DD code is to terminate there. The D code, on the other hand, converges to the other Clarke stationary points almost as often as the global minimizer. For $n = 5$ the DD code reduces the final function value to approximately

10^{-30} about a quarter of the time, but the D code only reaches 10^{-14} a tenth of the time. The DD code finds the other Clarke stationary points decreasingly less often as the function value increases, while the D code finds most of the other Clarke stationary points almost as often as the global minimizer, and rarely converges to a lower final function value than the DD code.

Starting at $n = 6$, neither code does very well in finding the global minimizer. The DD code reduces the final function value to 10^{-30} about a tenth of the time, while the D code reduces it to 10^{-14} around 60 times in 1000 runs. In both codes, all other Clarke stationary points are found about as frequently as the global minimizer. However, on practically every run, the DD code reduces the final function value lower than the D code for the same starting point. For $n = 7$, although the DD code finds a lower final function value than the D code on almost every run, neither code does well in converging to the global minimizer a significant amount of times. By the time $n = 8$, there is almost no difference between the performance of the D and the DD code. Neither code successfully reduces the final function value to the global minimizer any more often than to any of the other Clarke stationary values.

On each test, the DD version is able to get past the corners to a lower point, possibly even the global minimizer, more often than the D code. However, occasionally the D code is able to reduce the final function value lower than the DD code, which we hypothesize happens because the D code gets lucky with rounding in the right direction, lower than the DD code does. However, this phenomenon occurs so seldom, it has almost no statistical significance.

To better demonstrate the ability of the DD code to reduce the final function value and get past the corners, where the D code could not, we looked at the function values for each iteration during some of the runs where a lower Clarke stationary point or the global minimizer was obtained by the DD code. These iteration plots for NCR-NS2 are shown in Figure 6.5 on the following page. In the graph where $n = 3$, although the DD code does not reach the global minimizer, it demonstrates its ability to get past the first corner and reach a lower Clarke stationary value. In the case where $n = 5$, the D code terminates after only 215 iterations at a Clarke stationary value, while the DD code continues on past several more corners, eventually reaching and terminating at the global minimizer. Finally, in the graph where $n = 7$, the D code again terminates very quickly after only 305 iterations, stopping at a Clarke stationary value. The DD code, however, continues on for several tens of thousands more iterations, eventually reaching a final function value of 10^{-30}, where the code finally terminates.

In general, the DD runs for many more iterations than the D code before the program terminates. Consequentially, the DD code takes much longer to run. However, because of the huge effect different starting points have on the algorithm's ability and speed at finding the minimizer, it is difficult to find meaningful statistics on the time and number of iterations required. Some of the DD code runs took much longer than the corresponding D code, because of the longer time it takes to perform each

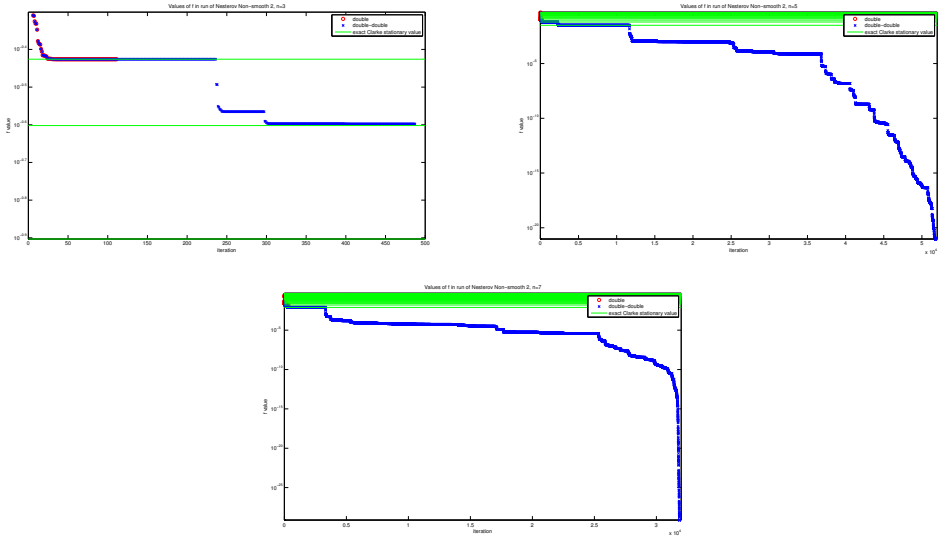

Figure 6.5: *Function values after each iteration of several runs of NCR-NS2. Top left: Plot for* $n = 3$*. The DD code is able to find a* smaller *stationary point than the D code after several more iterations beyond the D code termination. Top right: Plot for* $n = 5$*. The DD code is able to get past several more corners than the D code. Bottom: Plot for* $n = 7$*. The DD code gets past many more corners than the D code.*

floating point operation in double-double precision, the larger number of iterations the DD code required to converge, and the longer time spent in the linesearch in each iteration. For example, in one run of NCR-NS1, $n = 5$, the D code took 7841 iterations, took an average of 2.54 expansion and/or bisection steps per line search, and took 0.04 seconds to run. The DD code took 342,830 iterations, took an average of 3.14 expansion and/or bisection steps per line search, and took 20.54 seconds to run. In addition, the D code has at most 25 expansion and/or bisection steps per line search, while the DD code took at most 72 expansion and/or bisection steps per line search. The DD code took approximately 44 times more iterations, but ran for about 514 times longer before terminating.

7 Conclusion

Using the BFGS code written in C by Skajaa [Ska10], we were able to implement double-double precision and test the hypothesis by Lewis and Overton [LO10] that BFGS is limited by machine precision, rather than the algorithm itself. In general, the DD code was able to reduce the final function values closer to the minimal value than the D code. In many cases, both codes were able to reduce the function essentially to the minimum value, 10^{-16} for the D code and 10^{-32} for the DD code.

Depending on the specific problem and the starting points chosen, the DD code was sometimes able to reduce the final function value significantly farther than the D code, particularly in the case of getting past Clarke stationary points in NCR-NS2.

As we saw on the Nesterov functions, the DD code performed much better overall than the D code for all three variations. For the smooth version, the DD code reduced the final function value much lower than the D code, 10^{-323} compared to 10^{-32}, without taking an exorbitant amount of time before terminating under machine precision. On the first nonsmooth variation, the DD code proved much better at reducing the final function value better than the D code for a small number of variables. For $n = 6$ and larger, neither code performed well, and the DD version only did slightly better than the D code. However, the DD code took much longer to run, partly because of the extra iterations taken, and partly because each linesearch required more bisections and expansions before a sufficient stepsize was found.

For the second nonsmooth variation of Nesterov's function, the DD code proved much better at finding the minimizer and getting past the Clarke stationary points than the D code. When $n = 2, ..., 5$ the DD code found the minimizer more often than the other stationary points, and in general converged to a lower function value than the corresponding D code. For $n = 6$ and larger, neither code does particularly well in finding the minimizer more than the other stationary points; however the DD code still does slightly better. By $n = 8$, however, there is almost no difference in the final function values of the two codes. With few exceptions, the double-double precision code proved much better at finding the minimizer and getting past the Clarke stationary points. However, for a larger number of variables and more difficult problems, much longer time is needed for the double-double precision code to run, and in some cases, an unrealistic amount of time is required before the code breaks down because of machine precision.

Acknowledgements

I would like to thank my advisor, Professor Overton, for all the hard work he put in to helping me complete this thesis.

I would also like to thank Anders Skajaa for the critical role he played in translating the majority of the BFGS code into C, making my work far less than it would have been otherwise.

I also want to thank Andreas Kloeckner and Ben Stifler for the significant programming help they provided me on various occassions.

A big thanks also to Esteban Tabak for being my second reader on this thesis.

Finally, I would like to thank the following people for their small, but significant contributions:

Rachael Bailine
Evan Herring
Edwin Marte
Smeet Merchant
Ethan Price-Livingston
Brian Sampson
Serge Yegi

References

[Bai10] D.H. Bailey. A Fortran-90 based multiprecision system. *ACM Transactions on Mathematical Software*, 21(4):379–387, December 2010.

[GO10] M. Gurbuzbalaban and M.L. Overton. On Nesterov's nonsmooth Chebyshev-Rosenbrock functions. *Submitted to J. Nonlinear Analysis: Theory, Methods, Applications*, December 2010.

[HMM04] M. Haarala, K. Miettinen, and M.M. Makela. New limited memory bundle method for large-scale nonsmooth optimization. *Optimization Methods and Software*, 19(6):673–692, 2004.

[Kiw85] K.C. Kiwiel. *Methods of Descent for Nondifferentiable Optimization. Lecture Notes in Mathematics 1133*. Springer Verlag, 1985.

[LDB+02] X. Li, J. Demmel, D. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, A. Kapur, M.C. Martin, B. Thompson, and D.J. Yoo. Design, implementation, and testing of extended and mixed precision BLAS. ACM Trans. Math. Software, 2002.

[Lew03] A.S. Lewis. Active sets, nonsmoothness and sensitivity. *SIAM Journal on Optimization*, 13:702–725, 2003.

[LHB01] X. Li, Y. Hida, and D.H. Bailey. Algorithms for quad-double precision floating point arithmetic. In *15th IEEE Symposium on Clomputer Arithmetic*, pages 155–162, June 2001.

[LO10] A.S. Lewis and M.L. Overton. *Nonsmooth Optimization via Quasi-Newton Methods*. August 2010.

[LTS+02] L. Luksan, M. Tuma, M. Siska, J. Vlcek, and N. Ramesova. UFO 2002. Interactive system for universal functional optimization. Research report, Academcy of Sciences of the Czech Republic, 2002.

[NW06] J. Nocedal and S. Wright. *Numerical Optimization*. Springer, 2nd edition, 2006.

[Ove01] M.L. Overton. *Numerical Computing with IEEE Arithmetic*. SIAM, 2001.

[Pri91] D.M. Priest. Algorithms for arbitrary precision floating point arithmetic. In P. Kornerup and D. Matula, editors, *Proceedings of the 10th Symposium on Computer Arithmetic*, pages 132–143, Piscataway, NJ, 1991. IEEE Computer Society Press.

[Ska10] A. Skajaa. Limited memory BFGS for nonsmooth optimization. Master's thesis, Courant Institute of Mathematical Science, New York University, January 2010. http://www2.imm.dtu.dk/ andsk/nsosite/webfiles/index.shtml.