Overview of Nashlib and its Implementations

John C Nash, retired professor, University of Ottawa Peter Olsen, retired ??

11/01/2021

Contents

Abstract

The repository https://github.com/pcolsen/Nash-Compact-Numerical-Methods is a collection of implementations of the algorithms from Nash (1979), which was written by one of us (JN) in the mid 1970s.

The present collection originated with a query from the other author (PO) concerning a possible Python implementation based on the Pascal codes in the second edition of the original book (Nash (1990)).

From email interchanges, the repository and the idea of gathering different implementations and ensuring workable example codes arose. We hope that these will be useful in various ways:

- as a focus for comparison of programming languages and coding styles for numerical computations
- as a source of didactic examples and exercises for students
- as a resource for workers needing to embed numerical computations within their application programs.

TO DO

The Nashlib project has been an ongoing effort since the mid-1970s, and there are always other ideas and implementations to try. Some of these are as follows.

- in this document, documentation of how to run the codes under Microsoft Windows, in particular, Windows 10
- in this document, documentation of how to run the codes under Apple Mac OS. Particular versions may or may not be relevant.
- in this document, documentation of how to run the codes under Android and/or Chrome operating systems, with possible version notes.
- codes in C and C++
- codes in Python. A list of those still to do ??
- how to time programs???

History

Nashlib began with a desire on the part of John Nash to be able to carry out some speculative computations at Agriculture Canada. After completing his D.Phil. at Oxford in June of 1972, ostensibly in quantum mechanics but with a large computational component, he had a Post-Doc in the Chemistry Department of the University of Alberta with Fraser Birss in the Theoretical Chemistry group. In the Autumn of 1972, he started a job search in a very tight market. At an interview with the Public Service of Canada, he met Tom Kerr of Agriculture Canada who – with likely a fine disregard of many rules and policies related to hiring in the Public Service – arranged an interview in Ottawa at the end of November 1972. John joined Ag Can in February 1973, and quickly discovered that computations generally needed to be attached to a project and be given a budget code. Programs were run on punched cards with remote job entry to either DataCrown (whose mainframe was in Toronto), or else Computel, who had a Univac 1108 in the basement of Ottawa's St. Laurent Shopping Center.

Given the effort and bureaucracy to establish a budget code for speculative work exploring computational algorithms, it turned out to be more attractive to use a Data General NOVA minicomputer to which there were perhaps half a dozen Teletype terminals (10 characters per second, noisy, but with the capability of reading or punching paper tape). One could connect to the DG and obtain a partition (workspace) of between 3K and 7K bytes (1500 to 3500 words) for both program and data. Programs were in a fairly restricted dialect of BASIC. As I recall they were tokenized which reduced storage a little. I requested and was supplied with ear protectors, which were necessary when "saving" a program to punched tape. This was done by LISTing it and turning on the punch.

The Economics Branch also had Hewlett-Packard 9820 and 9830 desktop calculators. These were for producing the many graphs used in the various reports on different sectors of the Canadian agricultural economy, but after hours could be used for experimental computations. The 9830A ran BASIC of a dialect more or less compatible with that of the NOVA, but the programs had to be re-entered from the keyboard at first. Later we acquired a card-reader. But not a punched tape reader! Even later a Tektronix 4051 was acquired (1976). Its BASIC was yet again different. Some designer thought it would be neat to include a FUZZ option that allowed comparisons in program logic to be done approximately. If you were not careful, this would vitiate program logic and lead to strange outputs.

In 1975, Mary Nash was awarded the Nancy Stirling Lambert Scholarship by Blackwell's of Oxford, the booksellers and publishers. This was to carry out a 1 year research Masters' degree in Librarianship. Mary decided to investigate the microfiche and microfilm sector of the publishing industry. In the process, she intereviewed Neville Goodman of Adam Hilger Inc., a 1-man scientific publishing house in Bristol. Neville thought there would be a good market for a book about minicomputer scientific programs and John got a contract. Neville's instinct was correct. The usual print run of books was around 750 at the time. Compact Numerical Methods (CNM) has sold over 5000. Not quite a best seller, but also not a best cellar – a book that lives in the author's basement.

The 1979 edition of CNM has algorithms in pseudo-code (Step and Description). For the Second Edition (with Adam Hilger acquired by the Institute of Physics), a 5.25 inch diskette of programs in Turbo Pascal was provided. In the present project, John has been horrified to realize that the vast majority of the code is to work around difficulties of the MD-DOS operating system. Hopefully the present project will go some way to cleaning up the code, as much of the work-around material is no longer needed.

At the time of release of CNM in 1979, John also prepared a Fortran collection of the codes. In an era before open source and online collections, this was called Nashlib and was sold on magnetic tape. There were, of course, still many BASIC programs around, primarily for North Star BASIC after John built a Horizon computer in the summer of 1978. We named it THINK. This initially had 48K of RAM and two 80K diskette drives, and used a DEC LA-36 printing terminal (30 characters per second). It also came with a kit to build a decimal floating-point unit. Cost: $8000, which came from our own pockets. Henry Wolkowicz of Waterloo was kind enough to offer early access to the new "electronic mail" facility of DARPA and so we got a 300 bps acoustic coupler and John wrote his own drivers in assembly code. (This is something one should avoid doing if at all possible.) THINK eventually got a TV-style monitor, another 8K of RAM (the kit cost $350). some custom, self-designed fuse protection for the very expensive print head of the LA-36, and a few other niceties.

Around 1980, John was engaged by the magazine **Interface Age** to write their MicroMathematician column. For some reason, nobody seemed to pick up on the potentially pejorative connotation of the title. Later, in the mid-1980s, he was Scientific Computing Editor for **Byte**.

While BASIC was John's primary computing language around this time and as the IBM PC and its clones came on stream, there was sufficient incompatibility between BASIC dialects on the many personal computers that it seemed sensible to seek another approach when writing the Second Edition of CNM. By 1987, Turbo Pascal was becoming popular, and John chose to implement in Pascal. Besides the fact it could be acquired cheaply and was widely used, Pascal does allow the presentation of algorithms in an orderly way. In particular, a cardinal rule is that nothing can be used before it is defined, a useful condition for presenting algorithms. Less conveniently, Pascal's authors did not specify the library of functions to be supplied, nor the interface to an Operating System. Moreover, Pascal can be rather verbose.

In recent years (post 2010), John has found R to be a useful programming language for exploring algorithms and for encoding **reference implementations**, that is, expressions of algorithms that will actually run and which record the structure of the algorithm. Reference implementations are not necessarily the most efficient nor the most elegant forms for a particular programming language. They are meant to express the flow and reasoning behind an algorithm, and be easy to follow.

Operation and documentation of this project

To provide for easy collaboration and for storage of the results of this project, one of us set up a Github repository at

https://github.com/pcolsen/Nash-Compact-Numerical-Methods

Github allows for Markdown (http://www.aaronsw.com/weblog/001189 and https://daringfireball.net/projects/markdown/) files to be included in the file repository. We have used this for the `README.md` file and some other introductory or indexing documentation. There is also a wiki associated with each project. We have NOT used the wiki in favour of including a `Documentation` directory within the file repository so that the documentation is carried along with any clone of the repository or download as a Zip archive.

A relatively well-developed resource for documentation is the `knitr` set of tools (Xie (2013)) that are vastly expanded within the RStudio computational ecosystem. See https://rstudio.com/products/rstudio/. In particular, the **Rmarkdown** flavour of the **markdown** markup language allows for LaTeX mathematical expressions to be included inline, BibTeX references to be cited easily, and code in a number of programming languages to be executed, with the results automatically included in the report. Reports can be output in a variety of ways such as HTML, PDF (our choice), or even as Microsoft Word documents. While Rstudio is a commercial operation, they provide a no-charge version of the Rstudio desktop and have committed Rmarkdown and supporting code to open source licenses.

Code is placed in `chunk` blocks that are delimited with three reverse single quotation marks (‘) at the left hand margin. The first of these can (and should) be qualified by various instructions and parameters inside curly braces {}. To display the text

```
some text to display
more text
yet more
```

we preface it with a line having the three back-quotes followed by

```
{r plaintext1, echo=TRUE, eval=FALSE}
```

(It is very difficult to get any line with the codes to display verbatim.)

We finish the text block with a line of just three reverse single quotes in a row. Though we seem to be trying to display an R code chunk, we effectively list the text. Similarly following the three back-quotes with

```
{r code=xfun::read_utf8('../Pascal2021/knitrExample.pas'), echo=TRUE, eval=FALSE}
```

will list the program `knitrExample.pas`. Note we still need to follow this with a line having the three back-quotes. The source code of the present document is in the file `NashlibOverview.Rmd`, and viewing that file in a text editor (or in the Rstudio IDE) will display the details.

Programming language issues in Nashlib implementations

Fortran

Fortran is one of the oldest so-called "high-level" programming languages. One of us first used this language in 1966 (as Fortran II) on an IBM 1620 computer. Fortran has gone through a number of revisions, but a lot of legacy code still runs easily, particularly in the Fortran 1977 dialect (possibly also so-called Fortran IV which is close to this). Especially is some special-purpose structures are avoided, we can often run legacy Fortran using the `gfortran` compiler available for Linux and likely some other platforms.

Note that Fortran indexes arrays from 1 (i.e., the first element is element 1). Moreover, the storage layout of multidimensional arrays is such that the first index varies most rapidly. That is, conventional 2-dimensional matrices are stored in memory in column order, top to bottom. This was important in early programs, where a number of tricks were used to exploit the indexing and storage for speed or code size.

Input / output in Fortran is handled via channels that are numbered. The original Nashlib code uses channel 15 for input ("card reader") and 16 for output ("line printer"). `gfortran` seeks files for these, using 5 and 6 respectively for `STDIN` and `STDOUT`. Thus we have changed to 5 and 6 for the current repository.

Sidebar: **Ringing the Bell"

In the late 1960s when IBM was introducing its multi-user operating systems MFT and MVT, the University of Calgary acquired a System /360 Model 50 and agreed to try the MVT roll-in/roll-out operating system. JN recalls a couple of interesting details:

- the Chief System Programmer for the University (??name?? Norm?) quite literally had his hair go grey over the summer of 1967;
- the operating system had a monitor that would sound an alarm bell if it detected a situation it could not resolve. IBM had, for reasons known only to some sadistic designer, to make this alarm a 12 inch fire bell right behind the main control panel of the machine. It became a badge of honour among programmers to "ring the bell". Given the size of the bell and proximity to the operator, there could be soiled underwear. The Fortran IV of the system used channel 1 for the card reader and channel 3 for the line printer, but I accidentally and undeservedly got the "ring the bell" honour by trying to read from channel 3. Oddly, a couple of years later I worked on an ICL 1906A at Oxford where one could "read" from the line printer and get the contents of the last line printed.

Minor Fortran issues

- `gfortran` does not now accept `DO` loops terminating on an executable statement, so a `CONTINUE` must be provided. Moreover, each loop must have a separate `CONTINUE` statement.
- The "0" format control seems to be ignored, so we have substituted a blank.

BASIC

BASIC was developed in 1964 for the Dartmouth College Time Sharing System by John Kemeny and Thomas Kurtz. One of us (JN) was Canadian representative at the meeting in 1983 that agreed the ISO/IEC 6373:1984 "Data Processing—Programming Languages—Minimal BASIC". Unfortunately, BASIC has been plagued by a multitude of incompatible dialects, though in the 1980s many BASIC interpreters and compilers would execute code that used all but a few of the IBM/Microsoft GWBASIC dialect elements. Microsoft has released the source code in 2020 at https://github.com/microsoft/GW-BASIC, but without any build scripts. However, executables are available, though some claimed sources may be malware. CAUTION! One of us has an old MS-DOS executable `GWBASIC.EXE` that runs quite satisfactorily under the `DOSbox` emulation software available in Linux repositories, but the infrastructure is awkward to use under `knitr` environment used for this document. We make some further comments below in duscussions of how to actually run the Nashlib codes.

Figure 1: IBM System /360 panel and operator station

For the purposes of the present effort, it seems that the Linux `bwbasic` (Bywater BASIC) is close enough to the GWBAIC dialect to be usable, and is, moreover, usable under `knitr` as illustrated below and elsewhare in this repository.

Typically BASIC indexes arrays from 1, but we make no comment on storage layout.

Dialect details

There are a number of ways in which BASIC dialects differ. Here we list SOME of these. There are too many dialects and details for this list to be exhaustive.

North Star BASIC, which was important for JN for the years 1978-1988 used `SQRT` for the square root, whereas many BASICs use `SQR`.

Many BASICs will suppress a line feed if the `PRINT` line ends with a comma (`,`), while others (including `bwbasic`) use a semi-colon (`;`). The comma may or may not cause other effects. In `bwbasic` it appears to add 9 spaces when between fields, but insert a line feed if at the end of the line. For example, if we run

```
10 PRINT "JOHN";
20 PRINT "--JOHN",
30 PRINT "==JOHN"
40 PRINT "JOHN","MARY"
50 QUIT
```

we see

```
#!/bin/bash
bwbasic ../BASIC/printbas.bas
echo "done"
```

```
## Bywater BASIC Interpreter/Shell, version 2.20 patch level 2
## Copyright (c) 1993, Ted A. Campbell
## Copyright (c) 1995-1997, Jon B. Volkoff
##
## JOHN--JOHN
## ==JOHN
## JOHN          MARY
##
## done
```

A particular nuisance of `bwbasic` is that it does NOT print large or small numbers in scientific notation, and seems to ignore the "PRINT USING" language structure. This is, we believe, an extension to the 1984 ISO Minimal BASIC standard (which appears to have been withdrawn), underlining some of the issues with BASIC for use in modern computing.

Machine precison for BASIC

We can compute the machine precision using the `calceps.bas` program derived from Malcolm (1972) and Kahan's PARANOIA (Karpinski (1985)). (JN commissioned the latter article for Byte when he was Scientific Computing Editor.)

```
100 PRINT "calceps.bas -- machine precision etc."
110 GOSUB 7000
120 PRINT  "B9 = ";B9;"  -- a large number"
130 PRINT "E5 = ";E5;"  -- a number for relative equality tests"
140 PRINT  "E9 = ";1e15*E9;"  -- the machine precision*1e+15,  E9=MIN (X : 1+X>1)"
150 PRINT "E6 = ";E6;"  -- the radix of arithmetic"
160 PRINT "J1 = ";J1;"  -- the number of radix digits in mantissa of FP numbers"
170 PRINT "E6^(-J1) * 1E+16=";1E+16*(E6^(-J1))
```

```
180 QUIT
7000 REM ENVRON -- SUBROUTINE TO DETERMINE MACHINE PRECISION
7010 REM                AND SET SOME STANDARD VARIABLES
7020 REM INPUTS:  NONE
7030 REM
7040 REM OUTPUTS:
7050 REM    B9 -- a large number (currently 1E+35)
7060 REM    E5 -- a number for relative equality tests (currently 10)
7070 REM    E9 -- the machine precision,  E9=MIN (X : 1+X>1)
7080 REM    E6 -- the radix of arithmetic
7090 REM    J1 -- the number of radis digits in mantissa of
7100 REM          floating-point numbers
7110 REM
7120 LET D1=1: REM use a variable to avoid constants when double
7130 REM       precision is invoked
7140 LET E5=10: REM arbitrary scaling for additive equality tests
7150 LET B9=1E+35: REM big number, not necessarily biggest possible
7160 LET E6=1: REM initial value for radix
7170 LET E9=1: REM initial value for machine precision
7180 LET E9=E9/2: REM start of loop to decrease estimated machine precision
7190 LET D0=E6+E9: REM force storage of sum into a floating-point scalar
7200 IF D0>E6 THEN 7180: REM repeat reduction while (1+E9) > 1
7210 LET E9=E9*2: REM restore smallest E9 which gives (1+E9) > 1
7220 LET E6=E6+1: REM try different radix values
7230 LET D0=E6+E9
7240 IF D0>E6 THEN 7220: REM until a shift is observed
7250 LET J1=1: REM initial count of radix digits in mantissa
7260 LET E9=1: REM use radix for exact machine precision
7270 LET E9=E9/E6: REM loop while dividing by radix
7280 LET J1=J1+1: REM increment counter
7290 LET D0=D1+E9: REM add tp 1
7300 IF D0>D1 THEN 7270: REM test and repeat until equality
7310 LET E9=E9*E6: REM recover last value of machine precision
7320 LET J1=J1-1: REM and adjust the number of digits
7330 RETURN
```

Running this in `bwbasic` gives the output below.

```
Bywater BASIC Interpreter/Shell, version 2.20 patch level 2

Copyright (c) 1993, Ted A. Campbell

Copyright (c) 1995-1997, Jon B. Volkoff


calceps.bas -- machine precision etc.
B9 =  99999999999999996863366107917975552  -- a large number
E5 =  10  -- a number for relative equality tests
E9 =  0.2220446  -- the machine precision*1e+15,  E9=MIN (X : 1+X>1)
E6 =  2  -- the radix of arithmetic
J1 =  53  -- the number of radix digits in mantissa of FP numbers
E6^(-J1) * 1E+16= 1.110223
```

A slight modification of the program is needed to run under the `bas` interpreter (??ref):

```
100 PRINT "calceps.bas -- machine precision etc."
110 GOSUB 7000
120 PRINT  "B9 = ";B9;"  -- a large number"
130 PRINT "E5 = ";E5;"  -- a number for relative equality tests"
140 PRINT  "E9 = ";1e15*E9;"  -- the machine precision*1e+15,  E9=MIN (X : 1+X>1)"
150 PRINT "E6 = ";E6;"  -- the radix of arithmetic"
160 PRINT "J1 = ";J1;"  -- the number of radix digits in mantissa of FP numbers"
170 PRINT "E6^(-J1) * 1E+16=";1E+16*(E6^(-J1))
180 QUIT
7000 REM ENVRON -- SUBROUTINE TO DETERMINE MACHINE PRECISION
7010 REM                AND SET SOME STANDARD VARIABLES
7020 REM INPUTS:  NONE
7030 REM
7040 REM OUTPUTS:
7050 REM    B9 -- a large number (currently 1E+35)
7060 REM    E5 -- a number for relative equality tests (currently 10)
7070 REM    E9 -- the machine precision,  E9=MIN (X : 1+X>1)
7080 REM    E6 -- the radix of arithmetic
7090 REM    J1 -- the number of radis digits in mantissa of
7100 REM          floating-point numbers
7110 REM
7120 LET D1=1: REM use a variable to avoid constants when double
7130 REM       precision is invoked
7140 LET E5=10: REM arbitrary scaling for additive equality tests
7150 LET B9=1E+35: REM big number, not necessarily biggest possible
7160 LET E6=1: REM initial value for radix
7170 LET E9=1: REM initial value for machine precision
7180 LET E9=E9/2: REM start of loop to decrease estimated machine precision
7190 LET D0=E6+E9: REM force storage of sum into a floating-point scalar
7200 IF D0>E6 THEN 7180: REM repeat reduction while (1+E9) > 1
7210 LET E9=E9*2: REM restore smallest E9 which gives (1+E9) > 1
7220 LET E6=E6+1: REM try different radix values
7230 LET D0=E6+E9
7240 IF D0>E6 THEN 7220: REM until a shift is observed
7250 LET J1=1: REM initial count of radix digits in mantissa
7260 LET E9=1: REM use radix for exact machine precision
7270 LET E9=E9/E6: REM loop while dividing by radix
7280 LET J1=J1+1: REM increment counter
7290 LET D0=D1+E9: REM add tp 1
7300 IF D0>D1 THEN 7270: REM test and repeat until equality
7310 LET E9=E9*E6: REM recover last value of machine precision
7320 LET J1=J1-1: REM and adjust the number of digits
7330 RETURN
```

Running this in `bas` gives the output below.

```
bas ../BASIC/calcepsbas.bas <../BASIC/yes.in
```

```
## calceps.bas -- machine precision etc.
## B9 =  1e+35   -- a large number
## E5 =  10   -- a number for relative equality tests
## E9 =  2.220446e-16   -- the machine precision*1e+15,  E9=MIN (X : 1+X>1)
## E6 =  2   -- the radix of arithmetic
## J1 =  53   -- the number of radix digits in mantissa of FP numbers
## E6^(-J1+1) = 2.220446e-16
```

```
## Quit without saving? (y/n) y
```

Pascal

Pascal is another programming language that suffers from many dialects. In this case, it is less from dialect variation in the language itself, but in the libraries of functions supporting the execution of programs. In specifying the language, Wirth (1971) left these libraries undefined, leading to some considerable chaos in the commands to read or write information to or from programs. Borland Turbo Pascal gained sufficient marked dominance to create a *de facto* common dialect that has been more or less copied by the Free Pascal initiative (Free Pascal Team (n.d.)).

Python

- Issue of indexing arrays from 0.
- Issue of numPy and sciPy infrastructure.
- Issue of array indexing style. [i] [j] vs [i, j]

R

R indexes from 1. It is a very high level language, since there are single commands for many quite complex calculations like linear modeling (least squares solution, with extra computations to estimate standard errors and other ancillary information. R is, in the experience of one of the authors (JN), extremely good for presenting reference implementations of algorithms. In this it is similar to Matlab / Octave.

Others

?? Do we want to consider C (possibly using the Fortran and using f2c as starter?)

Running codes under different operating systems

This part of the Overview document is likely to evolve as we and other workers apply our ideas to different computing platforms. We welcome input and collaboration.

Our principal objective is to be able to take a program in one of the selected programming languages (or dialects) and have it execute correctly and produce acceptable output on any reasonably popular operating system. In the subsections below, we outline how we do this, including the adaptations for different operating environments.

Fortran

Rmarkdown is able to compiler Fortran subroutines and then call them from R. We leave out the question of timing for which this simple example was devised. Also we are assuming that Rmarkdown works for the operating systems likely to be of interest.

```
C Fortran test
      subroutine fexp(n, x)
      double precision x
C  output
      integer n, i
C  input value
      do 10 i=1,n
         x=dexp(dcos(dsin(dble(float(i)))))
  10  continue
      return
      end
```

Now try running it from R.

```
res = .Fortran("fexp", n=100000L, x=0)
str(res)
## List of 2
##  $ n: int 100000
##  $ x: num 2.72
```

What about complete programs? In this case we need to work through the operating system. For JN this is a Linux Mint MATE distribution.

The following is file `knitrExample.f`. It is in the directory `../fortran/` relative to this documentation file (`NashlibOverview.Rmd`). See the source of this document for the mechanism by which the Fortran code is read into our discussion text. Note the spaces before `exfort` – this is the name of the chunk, and NOT the name of the `knitr` programming language engine. Here we want simply to include a chunk of text which is echoed but not evaluated.

```
C  Test a computation
       do 10, i=1,1000
        y=exp(sin(cos(dble(i))))
10     continue
       write(*,100) y
 100   format('0 last value of y = ',1pe16.8)
       end
```

```
## gfortran -fno-optimize-sibling-calls  -fpic  -g -O2 -fdebug-prefix-map=/build/r-base-8T8CYO/r-base-4
## gcc -std=gnu99 -shared -L/usr/lib/R/lib -Wl,-Bsymbolic-functions -Wl,-z,relro -o f84235fcf06ee.so f8
```

We run the code with a command line script (**bash** script). Note that we would ideally like to time the execution, but timing commands tend to be very particular to the programming environment. To keep this document portable, we will leave out the timing except for R.

```
#!/bin/bash
gfortran ../fortran/knitrExample.f
./a.out
```

```
## 0 last value of y =   1.70437825E+00
```

BASIC

Our example code is similar to that for Fortran.

```
10 print "Example for testing BASIC execution"
15 let n=1000
20 for i = 1 to n
30    x = exp(sin(cos(i)))
40 next i
50 print "x=";x
60 quit
70 end
```

And we run it with **bash**. Note that our example above includes a line `quit`. This is NOT general, but seems to be the way to exit from the **bwbasic** interpreter chosen for Linux Mint 20 by the authors.

```
#!/bin/bash
bwbasic ../BASIC/knitrExample.bas
echo "done"
```

```
## Bywater BASIC Interpreter/Shell, version 2.20 patch level 2
## Copyright (c) 1993, Ted A. Campbell
## Copyright (c) 1995-1997, Jon B. Volkoff
##
## Example for testing BASIC execution
## x= 1.7043782
##
## done
```

The use of `quit` is, however, not allowed in the `BAS` interpreter. See `Bas`, by Michael Haardt — http://www.moria.de/~michael/bas/ and (at 2021-1-31) version 2.6 as http://www.moria.de/~michael/bas/bas-2.6.tar.gz which fixes a few bugs. Changing `QUIT` to `STOP` allows the use of lines like

```
bas program4bas.bas
```

We can also use the `SYSTEM` command to exit a program running under `bas` but need to answer `y` from the keyboard or else from a file. See "Machine precision for BASIC" for an example.

Note that the tarball of `bas` must be extracted, then the program built with the commands

```
configure
make
sudo make install
```

which on Linux Mint 20.1 put the program in `/usr/bin/bas`.

It is also possible in Linux to run the old GWBASIC.EXE interpreter in the `DOSBOX` program. To today's eyes, the 25 line "screen" is impossibly tight and the interface extremely clunky.

In the Linux Mint 20.1 repository, there is also `brandy` and `yabasic`. These do not seem to be very compatible with either the dialect of BASIC (Data General / North Star) in which Nashlib codes were run in the 1970s and 1980s. Similarly, I used to find Chipmunk Basic http://www.nicholson.com/rhn/basic/ workable for the codes in Nash and Walker-Smith (1987). However an attempt to load a program on 2021-1-31 failed.

Chipmunk basic is downloadable for several platforms as a pre-compiled binary, and it may be that the file access parts of the code are not set up to mesh with Linux Mint 20.1.

Pascal

Our example code is once again similar to that for Fortran.

```
program paskio(input,output);

var
  n, i: integer;
  x: real;

begin
  n := 1000;
  for i:=1 to n do
  begin
    x := exp(sin(cos(i)));
  end;
  writeln('x=',x);
end.
```

And we run it with **bash** after compiling with **fpc**.

```
#!/bin/bash
fpc ../Pascal2021/knitrExample.pas
./knitrExample
```

Support routines and files for Pascal

The original Turbo Pascal 5 and 5.5 used in Nash (1990) required a great deal of code to interface successfully with the MS-DOS operating system to provide a relatively general software system. Some of this code was to define data structures, while other parts were for handling input and output, timing, and other functionality. Some of that code is included here, particularly for data structures.

Constants and type definitions Many of the lines of code which follow are NOT needed to run the Nashlib Pascal codes. They were included in the 1990 Turbo Pascal 5 collection because it was difficult to keep a record or log of output from the running examples in the MS-DOS operating system. The particular lines of interest concern input (via `infile`) and output (via `confile`).

```
{constype.def ==
  This file contains various definitions and type statements which are
  used throughout the collection of "Compact Numerical Methods".  In many
  cases not all definitions are needed, and users with very tight memory
  constraints may wish to remove some of the lines of this file when
  compiling certain programs.

  Modified for Turbo Pascal 5.0

          Copyright 1988, 1990 J.C.Nash
}
uses Dos, Crt; {Turbo Pascal 5.0 Modules}
{ 1. Interrupt, Unit, Interface, Implementation, Uses are reserved words now.}
{ 2. System,Dos,Crt are standard unit names in Turbo 5.0.}

const
```

```
  big = 1.0E+35;    {a very large number}
  Maxconst = 25;    {Maximum number of constants in data record}
  Maxobs = 100;     {Maximum number of observations in data record}
  Maxparm  = 25;    {Maximum number of parameters to adjust}
  Maxvars = 10;     {Maximum number of variables in data record}
  acctol = 0.0001;  {acceptable point tolerance for minimisation codes}
  maxm = 20;        {Maximum number or rows in a matrix}
  maxn = 20;        {Maximum number of columns in a matrix}
  maxmn = 40;       {maxn+maxm, the number of rows in a working array}
  maxsym = 210;     {maximum number of elements of a symmetric matrix
             which need to be stored = maxm * (maxm + 1)/2 }
  reltest = 10.0;   {a relative size used to check equality of numbers.
             Numbers x and y are considered equal if the
             floating-point representation of reltest+x equals
             that of reltest+y.}
  stepredn = 0.2;   {factor to reduce stepsize in line search}
  yearwrit = 1990;  {year in which file was written}

type
  str2 = string[2];
  rmatrix = array[1..maxm, 1..maxn] of real; {a real matrix}
  wmatrix = array[1..maxmn, 1..maxn] of real; {a working array, formed
                as one real matrix stacked on another}
  smatvec = array[1..maxsym] of real; {a vector to store a symmetric matrix
             as the row-wise expansion of its lower triangle}
  rvector = array[1..maxm] of real;  {a real vector. We will use vectors
             of m elements always. While this is NOT space efficient,
             it simplifies program codes.}
  cgmethodtype= (Fletcher_Reeves,Polak_Ribiere,Beale_Sorenson);
    {three possible forms of the conjugate gradients updating formulae}
  probdata = record
          m     : integer; {number of observations}
          nvar  : integer; {number of variables}
          nconst: integer; {number of constants}
          vconst: array[1..Maxconst] of real;
          Ydata : array[1..Maxobs, 1..Maxvars] of real;
          nlls  : boolean; {true if problem is nonlinear least squares}
        end;
{
  NOTE: Pascal does not let us define the work-space for the function
  within the user-defined code.  This is a weakness of Pascal for this
  type of work.

  The following variables allow us to keep a copy of all screen
  information in a file for some of the codes.  Pascal requires a
  variable (confile in this case) for the file itself.  The string
  variable confname is used for the name of the file.  Similar variables
  allow problem data to be read from the file dfile named dfname.
}
var {global definitions}
  banner     : string[80]; {program name and description}
  confile    : text;       {file for output of console image}
  confname   : string[64]; {a name for confile}
  dfile      : text;       {file for output of console image}
```

```
  dfname     : string[64]; {a name for confile}
  infile     : text;       {an input file (keyboard image)}
  infname    : string[64]; {a name for the input file}
  con        : text;       {the console file}
```

Python

We can run python code in Linux Mint (and likely most Linux distributions) as follows:

```
python3 ../python/A9.py
```

R

R is likely best run within the cross-platform Rstudio IDE, in fact in an Rmarkdown article such as the present document source or `NashlibAlgorithms.Rmd`.

Others

?? Possibly C

Date and Time functions

Though the 1990 Turbo Pascal codes used facilities like the above and Free Pascal supports similar features, we have NOT included timing in the current Nashlib implementations. However, in this part of the Overview document we will show some possible mechanisms that users may employ to time their code. These will, we are fairly certain, need modification for local needs.

There are two main needs:

- a time stamp to identify a particular "run" of a program or a particular version of a file. Note that other identification information, such as the machine used, location, compiler or interpreter, and other details could be important.
- elapsed time for a particular set of tasks.

With a sufficiently detailed pair of timestamps, the elapsed time could be worked out.

In the sub-sections below, we consider some examples which could be used as templates for time-stamping or timing the routines in the current incarnation of Nashlib.

Pascal

The routine `tdstamp.pas` in the original Turbo Pascal MAY work work in Free Pascal, since there is a code unit called `Dos`.

The following short routine shows how to get the machine time using this approach, if needed, e.g., for reporting the execution time of programs, modified from the Free Pascal documentation example for function `GetTime` from the `Dos` class.

```
Program TSTAMP;
 uses Dos;

{ Program to demonstrate the GetTime function. }

Function L0(w:word):string;
var
  s : string;
begin
```

```
  Str(w,s);
  if w<10 then
   L0:='0'+s
  else
   L0:=s;
end;

var
  Hour,Min,Sec,HSec : word;
begin
  GetTime(Hour,Min,Sec,HSec);
  WriteLn('Current time');
  WriteLn(L0(Hour),':',L0(Min),':',L0(Sec),'.',L0(HSec));
end.
```

Here is some output.

```
fpc ../Pascal2021/Timeex3.pas
##
echo "Completed fpc"
../Pascal2021/Timeex3
```

```
## Free Pascal Compiler version 3.0.4+dfsg-23 [2019/11/25] for x86_64
## Copyright (c) 1993-2017 by Florian Klaempfl and others
## Target OS: Linux for x86-64
## Compiling ../Pascal2021/Timeex3.pas
## Linking ../Pascal2021/Timeex3
## /usr/bin/ld.bfd: warning: link.res contains output sections; did you forget -T?
## 23 lines compiled, 0.1 sec
## Completed fpc
## Current time
## 12:04:54.46
```

There are also other time and data functions for Free Pascal in the `sysutils` unit, as documented in

https://www.freepascal.org/docs-html/rtl/sysutils/datetimeroutines.html

??? a more detailed approach Tdex.pas??

Fortran

A very nice exposition of timer use under `gfortran` in Fortran 90 is provided by https://faculty.washington.edu/rjl/uwhpsc-coursera/timing.html. This needs some minor modifications to work with the Fortran 77 dialect for most of Nashlib. The adjusted example code follows.

```
      program timings
      implicit none
      integer, parameter :: ntests = 1
      integer :: n
      real(kind=8) :: t1, t2, elapsed_time, x, y
      integer(kind=8) :: tclock1, tclock2, clock_rate
      integer :: i,j,k,itest

      call system_clock(tclock1)

      n = 54545454
      print *, "Will run exp(sin(cos(dble(i)))) for ",n," loops"
```

```
      call cpu_time(t1)   ! start cpu timer
      do itest=1,ntests
          y = 0.0
          do j = 1,n
            x=exp(sin(cos(dble(j))))
            y=y+x
          enddo
      enddo
      call cpu_time(t2)   ! end cpu timer
      print *, "CPU time =",(t2-t1)
      call system_clock(tclock2, clock_rate)
      elapsed_time = float(tclock2 - tclock1) / float(clock_rate)
      print *, "Elapsed time = ",elapsed_time
      end
```

```
#!/bin/bash
gfortran ../fortran/fortran77timer.f
./a.out
rm a.out
```

```
##  Will run exp(sin(cos(dble(i)))) for     54545454  loops
##  CPU time =   1.9913390000000000
##  Elapsed time =    2.0005614757537842
```

BASIC

Given the diversity of BASIC dialects, it is likely that users will have to research the capabilities of their local installation. However, both `bwbasic` and `bas` use the GWBASIC TIME$ and TIMER functions. Unfortunately, it appears the time resolution is only to 1 second, which is too coarse for many computations. Here is an example program to illustrate their use. Note that `END` is replaced with `QUIT` for `bwbasic` so the script can be run.

```
10 PRINT "bastimer.bas"
30 PRINT DATE$, TIME$
40 LET N=765432
45 LET T1=TIMER()
50 LET Y=0
60 FOR I=1 TO N
70 LET X=EXP(SIN(COS(I)))
80 LET Y=Y+X
90 NEXT I
100 LET T2=TIMER()
110 PRINT "# OF LOOPS=",N,"  ELAPSED SECS = ";T2-T1
120 END
```

We observe a considerable difference in performance of the two BASIC interpreters.

bas

```
#!/bin/bash
bas ../BASIC/bastimer.bas
```

```
## bastimer.bas
## 02-14-2021    12:04:56
## # OF LOOPS=    765432         ELAPSED SECS =  1
```

bwbasic

```
#!/bin/bash
bwbasic ../BASIC/bastimer.bw
```

```
## Bywater BASIC Interpreter/Shell, version 2.20 patch level 2
## Copyright (c) 1993, Ted A. Campbell
## Copyright (c) 1995-1997, Jon B. Volkoff
##
## bastimer.bas
## 02-14-2021   12:04:57
## # OF LOOPS=   765432         ELAPSED SECS =  14
```

R

R provides convenient utilities for timestamps and for timing. `system.time()` times a single script, while `microbenchmark` runs a number of replications of the script and provides a distribution of them. We believe `microbenchmark` should be used if accurate timings are wanted, but `system.time` is sufficient for most purposes, and only requires a single run of the script.

```
cat("R timer for exp(sin(cos)))\n")
```

```
## R timer for exp(sin(cos)))
```

```
# Rtimetest.R
n <- 2345678
timeexsico <- function(n){
  y <- 0.0
  for (i in 1:n){
    x<-exp(sin(cos(i)))
    y<-y+x
  }
  y
}

system.time(timeexsico(n))
```

```
##    user  system elapsed
##   0.345   0.000   0.345
```

```
library(microbenchmark)
microbenchmark(timeexsico(n))
```

```
## Unit: milliseconds
##           expr     min       lq     mean   median       uq     max neval
##  timeexsico(n) 336.249 345.4648 353.1558 351.7457 359.4726 385.445   100
```

```
cat("Done!\n")
```

```
## Done!
```

Python

```
../python/PyTime.py
```

```
## Sum of outputs= 2822086.1328393715
## Total runtime of the program is 1.7831785678863525
```

Data storage and algorithmic considerations

Matrices and Arrays

Mathematics generally talks about matrices (or sometimes tensors), while programmers use arrays for storing data. In designing compact algorithms, JN was often constrained by memory limits that affected the combined needs of both program and working data. Thus some structures used in the Nashlib algorithms may seem quite unlike the matrices that are presented in mathematical treatments of the same computational processes.

Solving linear equations

In matrix notation, we want to find x that solves

$$Ax = b$$

where A is a square matrix of order n and b is the so-called Right Hand Side (RHS). If we have a number of different RHS vectors, they can be consolidated as columns of a matrix B, and the solutions then viewed as the columns of a matrix X, so that

$$AX = B$$

In particular, if B is the identity (unit) matrix of order n, X will be the **inverse** of A, A^{-1}. For a number of reasons, we generally should NOT compute A^{-1} to solve for x via

$$x = A^{-1}b$$

though there are some situations where we may wish the inverse.

Many methods for solving linear equations can be thought of as applying a series of transformations in sequence to the initial equations. If T_k is a transformation matrix of order n, then the equations

$$T_kAX = T_kB$$

stay valid for any such transformation. There are many such transformations:

- row permutations
- row scalings – multiplication by a constant
- elimination operations where one element is zeroed by subtraction of a multiple of another row of the equations
- plane rotations (Givens' rotations), see https://en.wikipedia.org/wiki/Givens_rotation
- reflections (Householder transformations), see https://en.wikipedia.org/wiki/Householder_transformation

The general goal is to get the matrix

$$\prod_k T_kA$$

into a form that makes solution for each x easy. Typically we want an upper or lower triangular matrix that allows of back- or forward-substitution to solve for the elements of x recursively, or else a diagonal matrix, in which case the solution is obtained by division. Note that such "solution" methods are also available via transformations, and if we can obtain

$$\prod_k T_kA = 1_n$$

then the solution(s) are

$$X = \prod_k T_k B$$

If we think of actually carrying out the computation, the obvious approach is to compute the matrix multiplication of T_k and A, then that of T_k and B. However, this uses two sets of code, one for A and one for B. On the other hand, if we use a working array `W` that stores $ A | B $, we just have to extend the column index in the loop over the working array, and use just one set of code.

We will also find that we don't generally apply a full matrix multiplication in each transformation, but strip each such operation to its bare essentials.

In practice, we often break up the work. In Nashlib:

- Algorithm 3 uses Givens' rotations to transform $A|B$ so that the A part of the working array is upper triangular. Then Algorithm 6 is used to back-solve for the solution(s).
- Algorithm 5 uses row-wise elimination with partial-pivoting (row permutations to avoid near division by zero) to get an upper triangular form for the A portion of the working array, then the same Algorithm 6 back transformation.

Back-solution generally can be performed in the space to hold the RHS for a further saving of storage.

Compact storage

In the 1960s and 1970s, when storage was exceedingly tight, there was much interest in avoiding use of unnecessary memory. In particular, we often tried to avoid storing the strict upper (or alternatively lower) triangle of a symmetric matrix. Moreover, there were possibly (though in hindsight perhaps not truly) efficiencies if we could avoid arrays where both row and column indexing was required. Thus, we tried to store matrices in single storage vectors (linear arrays). The IBM 7090 Fortran IV Scientific Subroutine Library used the column-wise indexing of two-dimensional arrays to avoid explicit i and j indices. And in Nashlib, Algorithms 7, 8, and 9 work with the symmetric (and hopefully positive definite) matrix A in linear equations above stored as a vector of $n * (n + 1)/2$ elements taken row-wise, i.e., the elements of such a matrix enter the vector in order (from the 2D matrix)

(1,1), (2,1), (2,2), (3,1), (3,2), ..., (n,1), (n,2), ..., (n,n)

Of course, this now limits our ability to easily append columns of B to those of A. And our code becomes much, much more complicated.

Was this worth the trouble? In retrospect, and as the author, JN thinks not! Sigh.

In the case of Algorithm 9, the use of compact storage is an integral part of the algorithm of Bauer and Reinsch (1971) in Wilkinson, Reinsch, and Bauer (1971).

Linear least squares and storage considerations

Without going into too many details, we will present the linear least squares problem as

$$Ax \tilde{=} b$$

In this case A is an m by n matrix with $m >= n$ and b a vector of lenght m. We write **residuals** as

$$r = Ax - b$$

or as

$$r_1 = b - Ax$$

Then we wish to minimize the sum of squares $r'r$. This problem does not necessarily have a unique solution, but the **minimal length least squares solution** which is the x that has the smallest $x'x$ that also minimizes $r'r$ is unique.

The historically traditional method for solving the linear least squares problem was to form the **normal equations**

$$A'Ax = A'b$$

This was attractive to early computational workers, since while A is m by n, $A'A$ is only n by n. Unfortunately, this **sum of squares and cross-products** (SSCP) matrix can make the solution less reliable, and this is discussed with examples in Nash (1979) and Nash (1990).

Another approach is to form a QR decomposition of A, for example with Givens rotations.

$$A = QR$$

where Q is orthogonal (by construction for plane rotations) and R is upper triangular. We can rewrite our original form of the least squares problem as

$$Q'A = Q'QR = R \tilde{=} Q'b$$

R is an upper triangular matrix R_n stacked on an $m - n$ by n matrix of zeros. But $z = Q'b$ can be thought of as n-vector z_1 stacked on $(m - n)$-vector z_2. It can easily be shown (we won't do so here) that a least squares solution is the rather easily found (by back-substitution) solution of

$$R_n x = z_1$$

and the minimal sum of squares turns out to be the cross-product $z_2' z_2$. Sometimes the elements of z_2 are called **uncorrelated residuals**. The solution for x can actually be formed in the space used to store z_1 as a further storage saving, since back-substitution forms the elements of x in reverse order.

All this is very nice, but how can we use the ideas to both avoid forming the SSCP matrix and keep our storage requirements low?

Let us think of the row-wise application of the Givens transformations, and use a working array that is $n + 1$ by $n + 1$. (We can actually add more columns if we have more than one b vector.)

Suppose we put the first $n + 1$ rows of a merged $A|b$ working matrix into this storage and apply the row-wise Givens transformations until we have an n by n upper triangular matrix in the first n rows and columns of our working array. We further want row $n + 1$ to have n zeros (which is possible by simple transformations) and a single number in the $n + 1$, $n + 1$ position. This is the first element of z_2. We can write it out to external storage if was want to have it available, or else we can begin to accumulate the sum of squares.

We then put row $n + 2$ of $[A|b]$ into the bottom row of our working storage and eliminate the first n columns of this row with Givens transformations. This gives us another element of z_2. Repeat until all the data has been processed.

We can at this point solve for x. Algorithm 4, however, applies the one-sided Jacobi method to get a singular value decomposition of A allowing of a minimal length least squares solution as well as some useful diagnostic information about the condition of our problem. This was also published as Lefkovitch and Nash (1976).

A simplification of the one-sided Jacobi svd storage

Similar to the concatenation of matrices and vectors in the previous section to allow (row or left-hand side) transformations to be applied to multiple columns simultaneously, we can set up the column rotations for the Jacobi-svd (Algorithm 1) to be applied simultaneously to the working array (initially our m by n matrix A) that becomes $B = US$ and the n by n unit matrix that is transformed by rotations to V. This saves some duplication of code.

An extension to Marquardt stabilization or ridge regression

The Marquardt stabilization of the Gauss-Newton equations (Marquardt (1963)) and ridge regression (Hoerl and Kennard (1970)) approximately solve singular or near-singular linear least squares problems

$$Ax \tilde{=} b$$

by modifying the normal equations via the addition of a positive diagonal matrix D^2 (usually a unit matrix. We use squares here to emphasize the positivity and to simplify the exposition below.)

$$(A'A + k^2D^2)x = A'b$$

This can be accomplished without formation of the SSCP $A'A$ by appending kD to the bottom of A when applying transformations to solve the least squares problem. Zeros are added to the bottom of b as needed.

This approach is used in the present nonlinear least squares package `nlsr` (Nash and Murdoch (2019)), but it is NOT used in the Nashlib Algorithm 23. At the time (1974) when this choice was made, the balance of performance and storage seemed to favour the use of the SSCP (which was stored since several tries with different k were often needed) and Choleski decomposition and back-solution. Furthermore, the precision of calculations available was less than half that used by default today. The present use of a QR approach allows for more reliable calculations when $k = 0$ (no Marquardt stabilization or ridge parameter).

Which choices to make?

Clearly there are many options, and for users the fact that different implementations of linear algebraic methods make use of different choices causes quite a burden of care in ensuring data structures and program invocation is carried out correctly. We urge good documentation and clear but simple examples be provided, and hope to follow our own advice in this.

References

Bauer, F. L., and C. Reinsch. 1971. "Inversion of Positive Definite Matrices by the Gauss-Jordan Method." in Wilkinson et al. 1971, pages 45–49.

Free Pascal Team. n.d. *Free Pascal: A 32, 64 and 16 Bit Professional Pascal Compiler.* Fairfax, VA. https://www.freepascal.org/.

Hoerl, A. E., and R. W. Kennard. 1970. "Ridge Regression: Biased Estimation for Nonorthogonal Problems." *Technometrics* 12: 55–67.

Karpinski, Richard. 1985. "Paranoia: A Floating-Point Benchmark," *Byte Magazine* 10 (2): 223–35.

Lefkovitch, L. P., and John C. Nash. 1976. "Principal Components and Regression by Singular Value Decomposition on a Small Computer." *Applied Statistics* 25 (3): 210–16.

Malcolm, Michael A. 1972. "Algorithms to Reveal Properties of Floating-Point Arithmetic." *Commun. ACM* 15 (11): 949–51.

Marquardt, Donald W. 1963. "An Algorithm for Least-Squares Estimation of Nonlinear Parameters." *SIAM Journal on Applied Mathematics* 11 (2): 431–41.

Nash, John C. 1979. *Compact Numerical Methods for Computers : Linear Algebra and Function Minimisation*. Book. Hilger: Bristol.

———. 1990. *Compact Numerical Methods for Computers : Linear Algebra and Function Minimisation, Second Edition*. Book. Institute of Physics : Bristol.

Nash, John C, and Duncan Murdoch. 2019. *Nlsr: Functions for Nonlinear Least Squares Solutions*.

Nash, John C., and Mary Walker-Smith. 1987. *Nonlinear Parameter Estimation: An Integrated System in Basic*. New York: Marcel Dekker.

Wilkinson, J. H., C. Reinsch, and F. L. Bauer. 1971. *Linear Algebra*. Die Grundlehren Der Mathematischen Wissenschaften in Einzeldarstellungen, v. 10. Springer-Verlag.

Wirth, Niklaus. 1971. "The Programming Language Pascal." *Acta Informatica* 1: 35–63.

Xie, Y. 2013. *Dynamic Documents with R and Knitr*. Chapman & Hall/Crc the R Series. Taylor & Francis. https://books.google.ca/books?id=QZwAAAAAQBAJ.