Overview of Nashlib and its Implementations

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

11/01/2021

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.

History

?? give a history of the codes

Documentation of this project

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)). 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.

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. See the `NashlibOverview.Rmd` file which is the source of the present document for 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.

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.

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.

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 in different programming languages

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. In the subsections below, we outline how we do this, possibly with variations 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.

```
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 f397259a20a61.so f3
```

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
```

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
```

Python

We can run python code as follows:

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

```
## [[1 1 1 1]
##  [1 2 2 2]
##  [1 2 3 3]
##  [1 2 3 4]]
## [1, 1, 2, 1, 2, 3, 1, 2, 3, 4]
## [1, 1, 2, 1, 2, 3, 1, 2, 3, 4]
## i  2
## i  3
## i  4
## 4 : [1, 1, 2, 1, 2, 3, -1, -1, -1, 1.0]
## i  2
## i  3
## i  4
## 3 : [1, 1, 2, 0, 0, 2.0, -1, -1, -1, 1.0]
## i  2
## i  3
## i  4
## 2 : [1, 0, 2.0, 0, -1, 2.0, -1, 0, -1, 1.0]
## i  2
## i  3
## i  4
## 1 : [2.0, -1, 2.0, 0, -1, 2.0, 0, 0, -1, 1.0]
## [2.0, -1, 2.0, 0, -1, 2.0, 0, 0, -1, 1.0]
## matrix is of size  4
## [[2.0 -1 0 0]
##  [-1 2.0 -1 0]
##  [0 -1 2.0 -1]
##  [0 0 -1 1.0]]
## [[1 1 1 1]
##  [1 2 2 2]
##  [1 2 3 3]
##  [1 2 3 4]]
```

```
## [[1.0 0.0 0.0 0.0]
##  [0.0 1.0 0.0 0.0]
##  [0.0 0.0 1.0 0.0]
##  [0.0 0.0 0.0 1.0]]
```

?? need to display this!!!

R

Others

?? Possibly C

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

$$\prod_k T_k A$$

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

$$\prod_k T_k A = 1_n$$

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

$$Ax \cong b$$

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.

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.

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*.

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.