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('../pascal/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

Indexes from 1. Very high level. Good for reference implementations.

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 ff7a13321c046.so ff
```

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 ../pascal/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

References

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

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

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

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.