MPResults

April 26, 2020

1 Newton's Method in Multiple Precision: C. T. Kelley

This notebook documents the results in C. T. Kelley, ***Newton's Method in Mixed Precision***, 2020.[8]

As an example we will solve the Chandrasekhar H-equation [2]. This equation, which we describe in detail in the Example section, has a fast $O(N \log(N))$ function evaluation, a Jacobian evaluation that is $O(N^2)$ work analytically and $O(N^2 \log(N))$ with a finite difference. This means that most of the work, if you do things right, is in the LU factorization of the Jacobian.

The difference between double, single, and half precision will be clear in the results from the examples. This notebook has no half-precision computatons. Julia does half-precsion in software and that is very slow.

1.1 Contents

1.2 The Chandrasekhar H-Equation

The example is the mid-point rule discretization of the Chandrasekhar H-equation [2].

$$\mathcal{F}(H)(\mu) = H(\mu) - \left(1 - \frac{c}{2}\int_0^1 \frac{\mu H(\mu)}{\mu + \nu}\, d\nu\right)^{-1} = 0. \tag{1}$$

The nonlinear operator $\mathcal{F}$ is defined on $C[0,1]$, the space of continuous functions on $[0,1]$.

The equation has a well-understood dependence on the parameter c [9], [3]. The equation has unique solutions at $c = 0$ and $c = 1$ and two solutions for $0 < c < 1$. There is a simple fold singularity [5] at $c = 1$. Only one [2], [1] of the two solutions for $0 < c < 1$ is of physical interest and that is the one easiest to find numerically. One must do a continuation computation to find the other one.

The structure of the singularity is preserved if one discretizes the integral with any rule that integrates constants exactly. For the purposes of this paper the composite midpoint rule will suffice.

The N-point composite midpoint rule is

$$\int_0^1 f(\nu)\,d\nu \approx \frac{1}{N}\sum_{j=1}^{N} f(\nu_j) \tag{2}$$

where $\nu_j = (j-1/2)/N$ for $1 \le j \le N$. This rule is second-order accurate for sufficiently smooth functions f. The solution of the integral equation is, however, not smooth enough. $H'(\mu)$ has a logarithmic singularity at $\mu = 0$.

The discrete problem is

$$\mathbf{F}(\mathbf{u})_i \equiv u_i - \left(1 - \frac{c}{2N}\sum_{j=1}^{N} \frac{u_j \mu_i}{\mu_j + \mu_i}\right)^{-1} = 0. \tag{3}$$

One can simplify the approximate integral operator in (**??**) and expose some useful structure. Since

$$\frac{c}{2N}\sum_{j=1}^{N} \frac{u_j \mu_i}{\mu_j + \mu_i} = \frac{c(i-1/2)}{2N}\sum_{j=1}^{N} \frac{u_j}{i+j-1}. \tag{4}$$

hence the approximate integral operator is the product of a diagonal matrix and a Hankel matrix and one can use an FFT to evaluate that operator with $O(N\log(N))$ work [4].

We can express the approximation of the integral operator in matrix form

$$\mathbf{L}(\mathbf{u})_i = \frac{c(i-1/2)}{2N}\sum_{j=1}^{N} \frac{u_j}{i+j-1} \tag{5}$$

and compute the Jacobian analytically as

$$\mathbf{F}'(\mathbf{u}) = \mathbf{I} - \mathrm{diag}(\mathbf{G}(\mathbf{u}))^2 \mathbf{L} \tag{6}$$

where

$$\mathbf{G}(\mathbf{u})_i = \left(1 - \frac{c}{2N}\sum_{j=1}^{N} \frac{u_j \mu_i}{\mu_j + \mu_i}\right)^{-1}. \tag{7}$$

Hence the data for the Jacobian is already available after one computes $\mathbf{F}(\mathbf{u}) = \mathbf{u} - \mathbf{G}(\mathbf{u})$ and the Jacobian can be computed with $O(N^2)$ work. We do that in this example and therefore the only $O(N^3)$ part of the solve is the matrix factorization.

One could also approximate the Jacobian with forward differences. In this case one approximates the jth column $\mathbf{F}'(\mathbf{u})_j$ of the Jacobian with

$$\frac{\mathbf{F}(\mathbf{u} + h\tilde{\mathbf{e}}_j) - \mathbf{F}(\mathbf{u})}{h} \tag{8}$$

where $\tilde{\mathbf{e}}_j$ is a unit vector in the jth coordinate direction and h is a suitable difference increment. If one computes $\mathbf{F}$ in double precision with unit roundoff u_d, then $h = O(\|\mathbf{u}\|\sqrt{u_d})$ is a reasonable choice [6]. Then the error in the Jacobian is $O(\sqrt{u_d}) = O(u_s)$ where u_s is unit roundoff in single precision. The cost of a finite difference Jacobian in this example is $O(N^2\log(N))$ work.

The analysis in [8] suggests that there is no significant difference in the nonlinear iteration from either the choice of analytic or finite difference Jacobians or the choice of single or double precision for the linear solver. This notebook has the data used in that paper to support that assertion. You will be able to duplicate the results and play with the codes.

Half precision is another story and we have those codes for you, too.

1.3 Setting up

You need to install these packages:

- PyPlot
- LinearAlgebra
- JLD2
- Printf
- FFTW
- IJulia (You must have done this already or you would not be looking at this notebook.)
- AbstractFFTs

The directory is a Julia project. So all you should need to do to get going is to

1. Put the directory in your LOAD_PATH. The way to do this is to type

```
push!(LOAD_PATH,"/Users/yourid/whereyouputit/MPResults")
```

at the Julia prompt in the REPL or in a notebook code windown.

2. Now load the modules with

```
using MPResults2019
```

3. Then you can do a simple solve and test that you did it right by typing

```
hout=heqtest()
```

which I will do now. Make sure **MPResults** is in your LOAD_PATH! If you forget to do the push! command, strange things may happen.

Make absolutely sure that you are in the MPResults directory. Then …

[1]:
```
MPRdir=pwd()
push!(LOAD_PATH,MPRdir)
using MPResults
```

1.4 Running the solver

The codes solve the H-equation and plot/tabulate the results in various ways. **heqtest** prints on column of the tables in Chandrasekhar's book. It calls **nsold.jl** . I've shown you the output from the solver but that is not important for now. You get the details on the solver in the next section.

[2]:
```
heqtest()
```

```
0.00e+00      1.00000e+00
5.00e-02      1.04424e+00
```

```
1.00e-01      1.07236e+00
1.50e-01      1.09470e+00
2.00e-01      1.11346e+00
2.50e-01      1.12965e+00
3.00e-01      1.14389e+00
3.50e-01      1.15657e+00
4.00e-01      1.16797e+00
4.50e-01      1.17830e+00
5.00e-01      1.18773e+00
5.50e-01      1.19638e+00
6.00e-01      1.20435e+00
6.50e-01      1.21172e+00
7.00e-01      1.21856e+00
7.50e-01      1.22493e+00
8.00e-01      1.23088e+00
8.50e-01      1.23646e+00
9.00e-01      1.24169e+00
9.50e-01      1.24662e+00
1.00e+00      1.25126e+00



[1.54457e+00, 7.94167e-03, 1.55209e-07, 2.39149e-15, 1.61651e-15, 1.36877e-15,
1.47288e-15, 1.76242e-15, 2.20932e-15, 1.50598e-15, 1.06489e-15]
```

```
[2]: (exactout = (solution = [1.00707e+00; 1.01754e+00; ... ; 1.24989e+00;
     1.25081e+00], ithist = [1.54457e+00, 7.94167e-03, 1.55209e-07, 2.39149e-15,
     1.61651e-15, 1.36877e-15, 1.47288e-15, 1.76242e-15, 2.20932e-15, 1.50598e-15,
     1.06489e-15]), fdout = (solution = [1.00707e+00; 1.01754e+00; ... ; 1.24989e+00;
     1.25081e+00], ithist = [1.54457e+00, 7.94166e-03, 1.55234e-07, 2.02292e-15,
     1.40433e-15, 1.29473e-15, 1.47288e-15, 1.52226e-15, 1.27555e-15, 1.43901e-15,
     1.38667e-15]))
```

heqtest.jl, calls the solver and harvests some iteration statistics. The two columns of numbers are the reults from [2] (page 125). The iteration statistics are from KNL, the solver.

1.5 NSOLD

The solver is *nsold.jl* version .01. Keep in mind that nothing with a version number with a negative exponent field is likely to be very good. nsold.jl is included when you run the MPResults2019.jl module, which you do automatically when you type **using MPResults2019**

nsold.jl is a simple implemention of Newton's method (see [6] and [7]) using an LU factorization of the Jacobian to compute the Newton step with no line search or globalization. The code evaluates and factors the Jacobian at every nonlinear iteration.

1.6 Using nsold.jl

At the level of this notebook, it's pretty simple. Remember that Julia hates to allocate mememory. So your function and Jacobian evaluation routines should expect the calling function to **preallocate** the storage for both the function and Jacobian. Your functions will then use **.=** to put the function and Jacobian where they are supposed to be.

It's worthwhile to look at the help screen.

It's worth asking for help with nsold . . . .

[3]:
```
?nsold
```

```
search: nsold
plotnsold
```

[3]:
```
nsold(x, FS, FPS, F!, J!=diffjac!; rtol=1.e-6, atol=1.e-12,
        maxit=20, dx=1.e-6, pdata=nothing)
```

```
This is Version .01.  Nothing with a version number having a negative exponent
field can be trusted.
```

```
Nonlinear solvers from my books in Julia.  This version has no globalization, no
quasi-Newton methods, and no Newton-Krylov.  The mission here is to duplicate the
mixed precision results in my SIREV-ED submission.
```

2 Inputs:

- `x:  initial iterate`
- `FS: Preallcoated storage for function.  It is an N x 1 column vector`
- `FPS: preallcoated storage for Jacobian.  It is an N x N matrix`
- `F!:  function evaluation, the !  indicates that F! overwrites FS, your preallocated storage for the function.`
- `J!:  Jacobian evaluation, the !  indicates that J! overwrites FPS, your preallocated storage for the Jacobian.  If you leave this out the default is a finite difference Jacobian.`

---

3 keyword arguments (kwargs):

- `rtol and atol:  relative and absolute error tolerances`
- `maxit:  limit on nonlinear iterations`
- `dx:  difference increment in finite-difference derivatives h=dx*norm(x)+1.e-8`

- pdata: precomputed data for the function/Jacobian. Things will go better if you use this rather than hide the data in global variables within the module for your function/Jacobian

---

4 Using nsold.jl

Here are the rules as of June 6, 2019

F! is the nonlinear residual. J! is the Jacobian evaluation.

Put these things in a module and cook it up so that

1. You allocate storage for the function and Jacobian in advance -> in the calling program <- NOT in FS and FPS

FV=F!(FV,x) returns FV=F(x)

FP=J!(FV,FP,x) returns FP=F'(x); (FV,FP, x) must be the argument list, even if FP does not need FV. One reason for this is that the finite-difference Jacobian does and that is the default in the solver.

In the future J! will also be a matrix-vector product and FPS will be the PREALLOCATED (!!) storage for the GMRES(m) Krylov vectors.

Lemme tell ya 'bout precision. I designed this code for full precision functions and linear algebra in any precision you want. You can decleare FPS as Float64, Float32, or Float16 and nsold will do the right thing if YOU do not destroy the declaration in your J! function. I'm amazed that this works so easily.

If the Jacobian is reasonably well conditioned, I can see no reason to do linear algebra in double precision

Don't try to evaluate function and Jacobian all at once because that will cost you a extra function evaluation everytime the line search kicks in.

2. Any precomputed data for functions, Jacobians, matrix-vector products may live in global variables within the module. Don't do that if you can avoid it. Use pdata instead.

4.1 How nsold.jl controls the precision of the Jacobian

You can control the precision of the Jacobian by simply allocating FPS in your favorite precision. So if I have a problem with N=256 unknows I will decalare FP as zeros(N,1) and may delcare FPS as zeros(N,N) or Float32.(zeros(N,N)).

Note the . between Float32 and the paren. This, as is standard Julia practice, applies the conversion to Float32 to everyelement in the array. If you forget the . Julia will complain.

5 The results in the paper

The paper has plots for double, single, and half precsion computations for c=.5, .99, and 1.0. The half precision results take a very long time to get. On my computer (2019 iMac; 8 cores, 64GB of memoroy) the half precision compute time was over two weeks. Kids, don't try this at home.

The data for the paper are in the cleverly named directory Data_From_Paper

cd to the directory MPResults and from that directory run

data_harvest("Data_From_Paper/MP_Data_")

at the julia prompt you will generate all the tables and plots in the paper.

If you have the time and patience you can also generate the data with data_populate.jl. This create three directories named Mixed_Precision_c=? and you can see for yourself. Look at that file to see opportunities to edit the time-cosumng jobs out. If you eliminate the half precision work and the larger dimensions, the code will run in a short time.

data_populate(c; half=''no'',level=p) does double and single preicsion solves for 1024×2^k point grids for k=0, ... p-1 . Set half=''yes'' to make the computatio take much longer.

Here is a simple example of using data_populate and the plotter code plot_nsold.jl. I'm only using the 1024, 2048 and 4096 point grids. The plot in the paper uses more levels. This is part of Figure 1 in the paper.

Look at the source to data_populate.jl and plotnsold.jl and you'll see how I did this. These codes only mange files, plots, and tables. There is nothing really exciting here. You don't need to know much Julia to understand this, but you do need to know something and I can't help with that.

To begin with, I will create a directory called Data_Test to put all this stuff. I will cd to that directory.

[4]:
```julia
cd(MPRdir)
Home4mp="Data_Test"
try (mkdir(Home4mp))
catch
end
cd(Home4mp)
pwd()
```

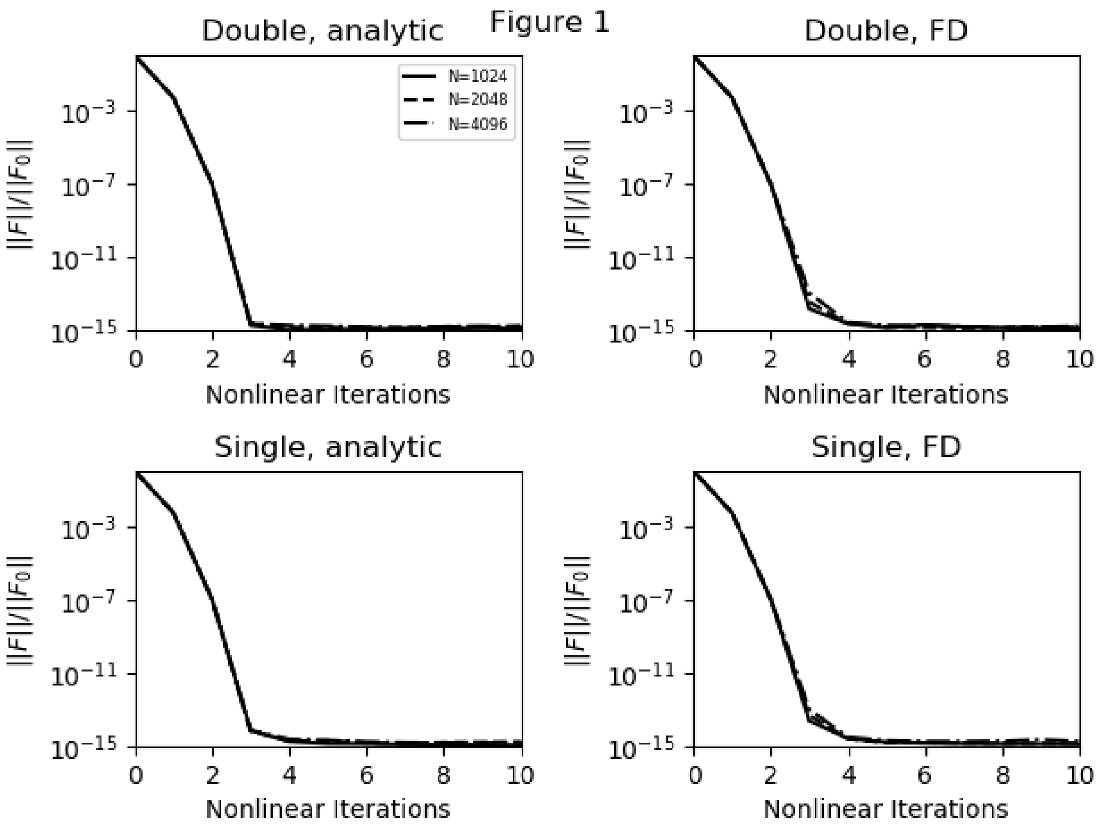
[4]:
```
"/Users/ctk/Dropbox/Julia/dev/MPResults/Data_Test"
```

Now I'll run data_populate to create a subdirectory with the data. cd to that directory and run plotnsold. That's how I created the plots in the paper with all values of c and all the problem sizes. The half precision computation took two weeks.

plotnsold(‘‘no’’,c,10,3;title) makes the four-plot with only double and single (no half).

Even with this small problem, data_populate takes a while. Be patient. Once it’s done the plots will appear pretty rapidly.

[5]:
```
using PyPlot
data_populate(.5;half="no",level=3)
# after data_populate's done, the plots happen fast
cd("Mixed_Precision_c=5")
figtitle="Figure 1"
plotnsold("no",.5,10,3;bigtitle=figtitle)
```



[ ]:
```
```

References

[1] I. W. BUSBRIDGE, *The Mathematics of Radiative Transfer*, no. 50 in Cambridge Tracts, Cambridge Univ. Press, Cambridge, 1960.

[2] S. CHANDRASEKHAR, *Radiative Transfer*, Dover, New York, 1960.

[3] D. W. DECKER AND C. T. KELLEY, *Newton's method at singular points I*, SIAM J. Numer. Anal., 17 (1980), pp. 66-70.

[4] G. H. GOLUB AND C. G. VANLOAN, *Matrix Computations*, Johns Hopkins studies in the mathematical sciences, Johns Hopkins University Press, Baltimore, 3 ed., 1996.

[5] H. B. KELLER, *Lectures on Numerical Methods in Bifurcation Theory*, Tata Institute of Fundamental Research, Lectures on Mathematics and Physics, Springer-Verlag, New York, 1987.

[6] C. T. KELLEY, *Iterative Methods for Linear and Nonlinear Equations*, no. 16 in Frontiers in Applied Mathematics, SIAM, Philadelphia, 1995.

[7] ———, *Solving Nonlinear Equations with Newton's Method*, no. 1 in Fundamentals of Algorithms, SIAM, Philadelphia, 2003.

[8] ———, *Newton's method in mixed precision*, 2020. in preparation.

[9] T. W. MULLIKIN, *Some probability distributions for neutron transport in a half space*, J. Appl. Prob., 5 (1968), pp. 357-374.