MPResults

December 8, 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.[7]

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 computations. 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 [11], [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 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 [7] 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 with **Pkg**. I assume you know how to do that.

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

To render the LaTeX you'll need to run the first markdown cell in the notebook. That sets up the commands you need to render the LaTeX correctly.

The directory is a Julia project. So all you should need to do to get going is to run the first code cell in this notebook. That cell has one line

```
include("mprnote.jl")
```

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

```
hout=heqtest()
```

which I will do in the next code cell. Now ...

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

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.

[12]:
```
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, 2.67377e-15, 1.53837e-15, 1.35064e-15,
1.52226e-15, 1.42178e-15, 1.47288e-15, 1.42178e-15, 1.33227e-15]
```

```
[12]: (exactout = (solution = [1.00707e+00, 1.01754e+00, 1.02622e+00, 1.03392e+00,
      1.04094e+00, 1.04744e+00, 1.05352e+00, 1.05925e+00, 1.06469e+00, 1.06986e+00  ...
      1.24220e+00, 1.24320e+00, 1.24419e+00, 1.24517e+00, 1.24614e+00, 1.24709e+00,
      1.24804e+00, 1.24897e+00, 1.24989e+00, 1.25081e+00], functionval = [0.00000e+00,
      0.00000e+00, 0.00000e+00, 0.00000e+00, 0.00000e+00, 0.00000e+00, 0.00000e+00,
      0.00000e+00, 0.00000e+00, 0.00000e+00  ...  0.00000e+00, 0.00000e+00, 4.
       ↪44089e-16,
      0.00000e+00, 0.00000e+00, -2.22045e-16, 0.00000e+00, 0.00000e+00, 0.00000e+00,
      0.00000e+00], history = [1.54457e+00, 7.94167e-03, 1.55209e-07, 2.67377e-15,
      1.53837e-15, 1.35064e-15, 1.52226e-15, 1.42178e-15, 1.47288e-15, 1.42178e-15,
      1.33227e-15], stats = (ifun = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], ijac = [0, 1,
      1, 1, 1, 1, 1, 1, 1, 1, 1], iarm = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]), idid =
      false, errcode = 10), fdout = (solution = [1.00707e+00, 1.01754e+00,
      1.02622e+00, 1.03392e+00, 1.04094e+00, 1.04744e+00, 1.05352e+00, 1.05925e+00,
      1.06469e+00, 1.06986e+00  ...  1.24220e+00, 1.24320e+00, 1.24419e+00, 1.
       ↪24517e+00,
      1.24614e+00, 1.24709e+00, 1.24804e+00, 1.24897e+00, 1.24989e+00, 1.25081e+00],
      functionval = [0.00000e+00, 0.00000e+00, 0.00000e+00, 0.00000e+00, 0.00000e+00,
      0.00000e+00, 0.00000e+00, 0.00000e+00, 0.00000e+00, 0.00000e+00  ...  0.
       ↪00000e+00,
      0.00000e+00, 4.44089e-16, 0.00000e+00, 0.00000e+00, -2.22045e-16, 0.00000e+00,
      0.00000e+00, 0.00000e+00, 0.00000e+00], history = [1.54457e+00, 7.94166e-03,
      1.55239e-07, 2.24254e-15, 1.58572e-15, 1.53837e-15, 1.47288e-15, 1.53837e-15,
      1.67640e-15, 1.57009e-15, 1.23629e-15], stats = (ifun = [1, 1, 1, 1, 1, 1, 1, 1,
      1, 1, 1], ijac = [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], iarm = [0, 0, 0, 0, 0, 0, 0,
      0, 0, 0, 0]), idid = false, errcode = 10))
```

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 nsold.jl, the solver.

1.5 NSOL

The solver is `nsol` from my pacakge **SIAMFANLEquations.jl** [10]. That pacackage and an **IJulia Notebook** [8]
support my upcoming book **Solving Nonlinear Equations with Iterative Methods: Solvers and Examples in Julia** [9] The solver and the H-equation example are both from that package. When you run the first code cell you are set up.

1.6 Using nsol.jl

At the level of this notebook, it's pretty simple. Remember that Julia hates to allocate memory. 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.

[13]:
```
?nsol
```

```
search: nsol
nsolsc
transcode
transpose
Transpose
transpose!
DimensionMismatch
```

[13]:
```
nsol(F!, x0, FS, FPS, J!=diffjac!; rtol=1.e-6, atol=1.e-12,
           maxit=20, solver="newton", sham=5, armmax=10, resdec=.1,
           dx = 1.e-7, armfix=false,
           pdata = nothing, jfact = klfact,
           printerr = true, keepsolhist = false, stagnationok=false)
```

```
)

C. T. Kelley, 2020

Julia versions of the nonlinear solvers from my SIAM books.  Herewith:  nsol

You must allocate storage for the function and Jacobian in advance -> in the
calling program <- ie.  in FS and FPS

Inputs:

  • F!:  function evaluation, the !  indicates that F! overwrites FS, your
    preallocated storage for the function.

    So FS=F!(FS,x) or FS=F!(FS,x,pdata) returns FS=F(x)

  • x0:  initial iterate

  • FS: Preallocated storage for function.  It is an N x 1 column vector
```

- FPS: preallocated storage for Jacobian. It is an N x N matrix
- 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.

  So, FP=J!(FP,FS,x) or FP=J!(FP,FS,x,pdata) returns FP=F'(x).

  (FP,FS, x) must be the argument list, even if FP does not need FS. One reason for this is that the finite-difference Jacobian does and that is the default in the solver.
- Precision: Lemme tell ya 'bout precision. I designed this code for full precision functions and linear algebra in any precision you want. You can declare FPS as Float64, Float32, or Float16 and nsol 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, you can cut the cost of Jacobian factorization and storage in half with no loss. For large dense Jacobians and inexpensive functions, this is a good deal.

  BUT ... There is very limited support for direct sparse solvers in anything other than Float64. I recommend that you only use Float64 with direct sparse solvers unless you really know what you're doing. I have a couple examples in the notebook, but watch out.

Keyword Arguments (kwargs):

- rtol and atol: relative and absolute error tolerances
- maxit: limit on nonlinear iterations

solver: default = "newton"

Your choices are "newton" or "chord". However, you have sham at your disposal only if you chose newton. "chord" will keep using the initial derivative until the iterate converges, uses the iteration budget, or the line search fails. It is not the same as sham=Inf, which is smarter.

sham: default = 5 (ie Newton)

This is the Shamanskii method. If sham=1, you have Newton. The iteration updates the derivative every sham iterations. The convergence rate has local q-order sham+1 if you only count iterations where you update the derivative. You need not provide your own derivative function to use this option. sham=Inf is chord only if chord is converging well.

I made sham=1 the default for scalar equations. For systems I'm more aggressive and want to invest as little energy in linear algebra as possible. So the default is sham=5.

armmax: upper bound on step size reductions in line search

resdec: default = .1

This is the target value for residual reduction. The default value is .1. In the old MATLAB codes it was .5. I only turn Shamanskii on if the residuals are decreasing rapidly, at least a factor of resdec, and the line search is quiescent. If you want to eliminate resdec from the method (you don't) then set resdec = 1.0 and you will never hear from it again.

dx: default = 1.e-7

difference increment in finite-difference derivatives h=dx*norm(x,Inf)+1.e-8

armfix: default = false

The default is a parabolic line search (ie false). Set to true and the step size will be fixed at .5. Don't do this unless you are doing experiments for research.

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

jfact: default = klfact (tries to figure out best choice)

If your Jacobian has any special structure, please set jfact to the correct choice for a factorization.

I use jfact when I call PrepareJac! to evaluate the Jacobian (using your J!) and factor it. The default is to use klfact (an internal function) to do something reasonable. For general matrices, klfact picks lu! to compute an LU factorization and share storage with the Jacobian. You may change LU to something else by, for example, setting jfact = cholseky! if your Jacobian is spd.

klfact knows about banded matrices and picks qr. You should, however RTFM, allocate the extra two upper bands, and use jfact=qr! to override klfact.

If you give me something that klfact does not know how to dispatch on, then nothing happens. I just return the original Jacobian matrix and nsol will use backslash to compute the Newton step. I know that this is probably not optimal in your situation, so it is good to pick something else, like jfact = lu.

Please do not mess with the line that calls PrepareJac!.

```
FPF = PrepareJac!(FPS, FS, x, ItRules)
```

FPF is not the same as FPS (the storage you allocate for the Jacobian) for a reason. FPF and FPS do not have the same type, even though they share storage. So, FPS=PrepareJac!(FPS, FS, ...) will break things.

printerr: default = true

I print a helpful message when the solver fails. To suppress that message set printerr to false.

keepsolhist: default = false

Set this to true to get the history of the iteration in the output tuple. This is on by default for scalar equations and off for systems. Only turn it on if you have use for the data, which can get REALLY LARGE.

stagnationok: default = false

Set this to true if you want to disable the line search and either observe divergence or stagnation. This is only useful for research or writing a book.

Output:

A named tuple (solution, functionval, history, stats, idid, errcode, solhist) where

solution = converged result functionval = F(solution) history = the vector of residual norms (||F(x)||) for the iteration stats = named tuple of the history of (ifun, ijac, iarm), the number of functions/derivatives/steplength reductions at each iteration.

I do not count the function values for a finite-difference derivative because they count toward a Jacobian evaluation.

idid=true if the iteration succeeded and false if not.

errcode = 0 if if the iteration succeeded = -1 if the initial iterate satisfies the termination criteria = 10 if no convergence after maxit iterations = 1 if the line search failed

solhist:

This is the entire history of the iteration if you've set keepsolhist=true

solhist is an N x K array where N is the length of x and K is the number of iteration + 1. So, for scalar equations, it's a row vector.

---

2 Examples

World's easiest problem example. Test 64 and 32 bit Jacobians. No meaningful difference in the residual histories or the converged solutions.

```
 julia> function f!(fv,x)
       fv[1]=x[1] + sin(x[2])
       fv[2]=cos(x[1]+x[2])
       end
f (generic function with 1 method)

julia> x=ones(2,); fv=zeros(2,); jv=zeros(2,2); jv32=zeros(Float32,2,2);
julia> nout=nsol(f!,x,fv,jv; sham=1);
julia> nout32=nsol(f!,x,fv,jv32; sham=1);
julia> [nout.history nout32.history]
5×2 Array{Float64,2}:
 1.88791e+00  1.88791e+00
 2.43119e-01  2.43119e-01
```

```
 1.19231e-02  1.19231e-02
 1.03266e-05  1.03262e-05
 1.46416e-11  1.43548e-11

julia> [nout.solution nout.solution - nout32.solution]
2×2 Array{Float64,2}:
 -7.39085e-01  -5.42899e-14
  2.30988e+00   3.49498e-13
```

H-equation example. I'm taking the sham=5 default here, so the convergence is not quadratic. The good news is that we evaluate the Jacobian only once.

```
julia> n=16; x0=ones(n,); FV=ones(n,); JV=ones(n,n);
julia> hdata=heqinit(x0, .5);
julia> hout=nsol(heqf!,x0,FV,JV;pdata=hdata);
julia> hout.history
4-element Array{Float64,1}:
 6.17376e-01
 3.17810e-03
 2.75227e-05
 2.35817e-07
```

2.1 How nsol.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 unknowns I will declare FP as zeros(N,1) and may declare 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.

3 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 memory) 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()
```

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 creates binary files with the iteration histories and you can see for yourself how long it takes. You can reduce the number of grid levels and **turn half precision off**. I will turn half precision off in the example in this notebook. That means that the code will run in a reasonable amount of time instead of the **two weeks** it needs for the half precision results.

`data_populate(c; half=false,level=p)` does double and single preicsion solves for 1024×2^k point grids for k=0, ... p-1 . Set `half=true` to make the computations 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 **PlotData.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.

[14]:
```
cd(MPRdir)
Home4mp="Data_Test"
try (mkdir(Home4mp))
catch
end
cd(Home4mp)
pwd()
```

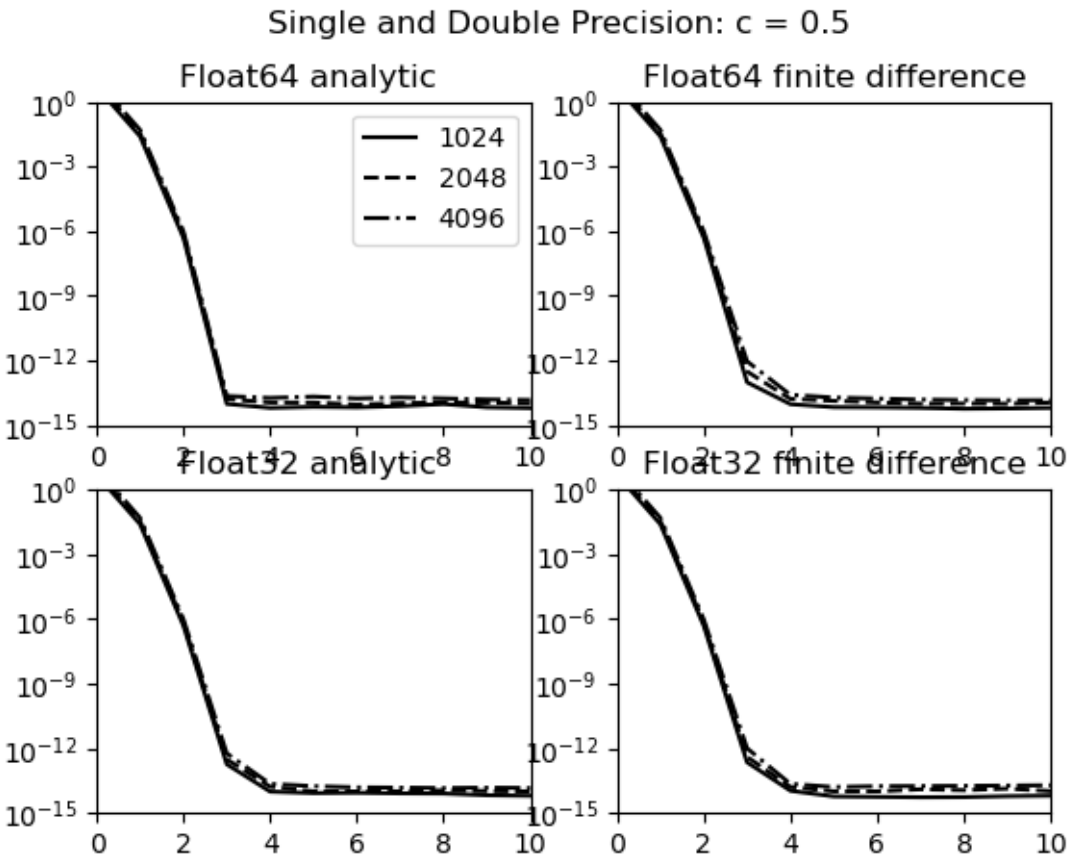
[14]:
```
"/Users/ctk/Dropbox/Julia/dev/MPResults/Data_Test"
```

Now I'll run **data_populate** to create a subdirectory with the data. cd to that directory and run **PlotData**. 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.

PlotData(.5, level=3) makes the four-plot with only double and single (no half).

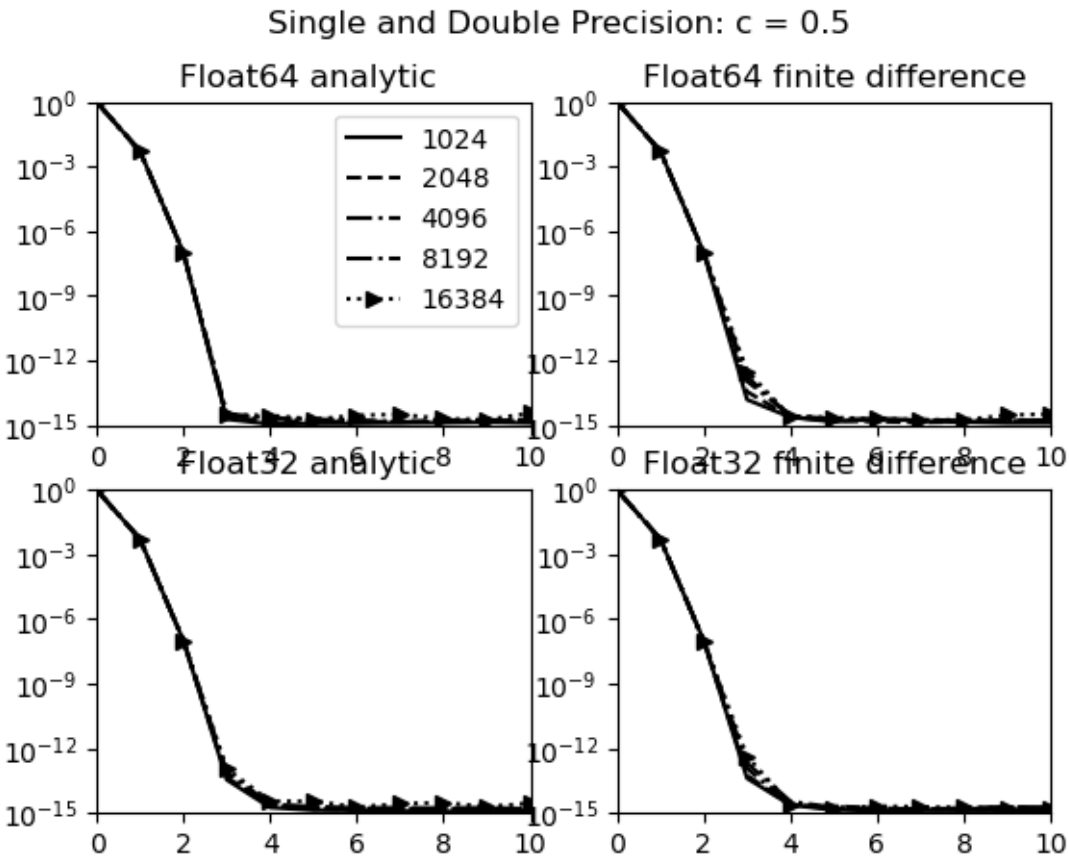
Even with this small problem, **data_populate** takes a while. Be patient. Once it's done the plots will appear pretty rapidly.

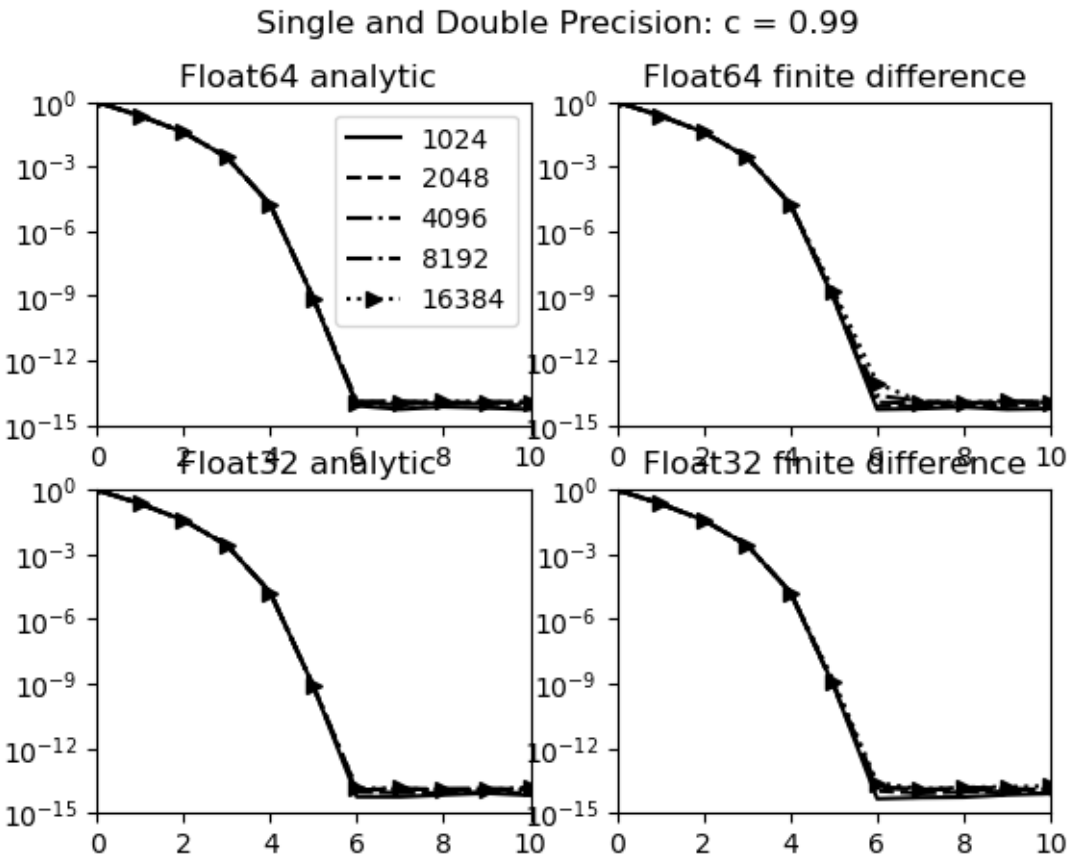
[15]:
```
using PyPlot
data_populate(.5; level=3)
PlotData(.5; level=3);
```

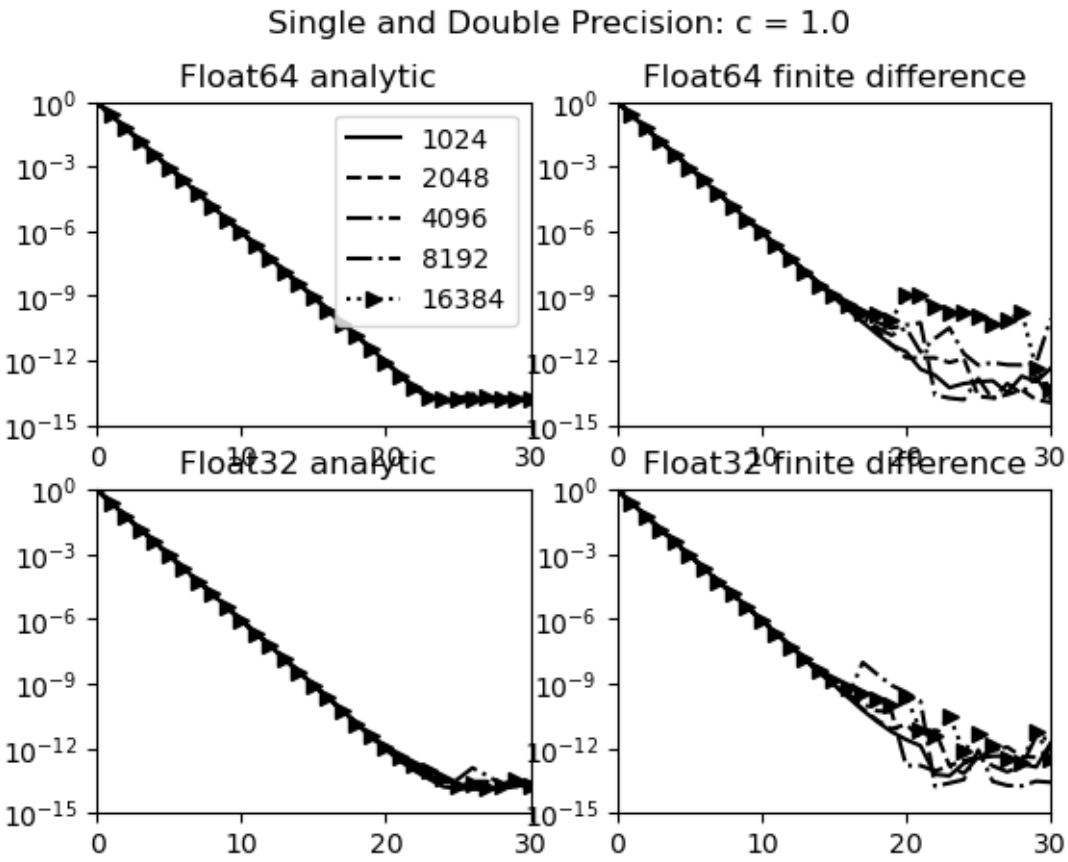


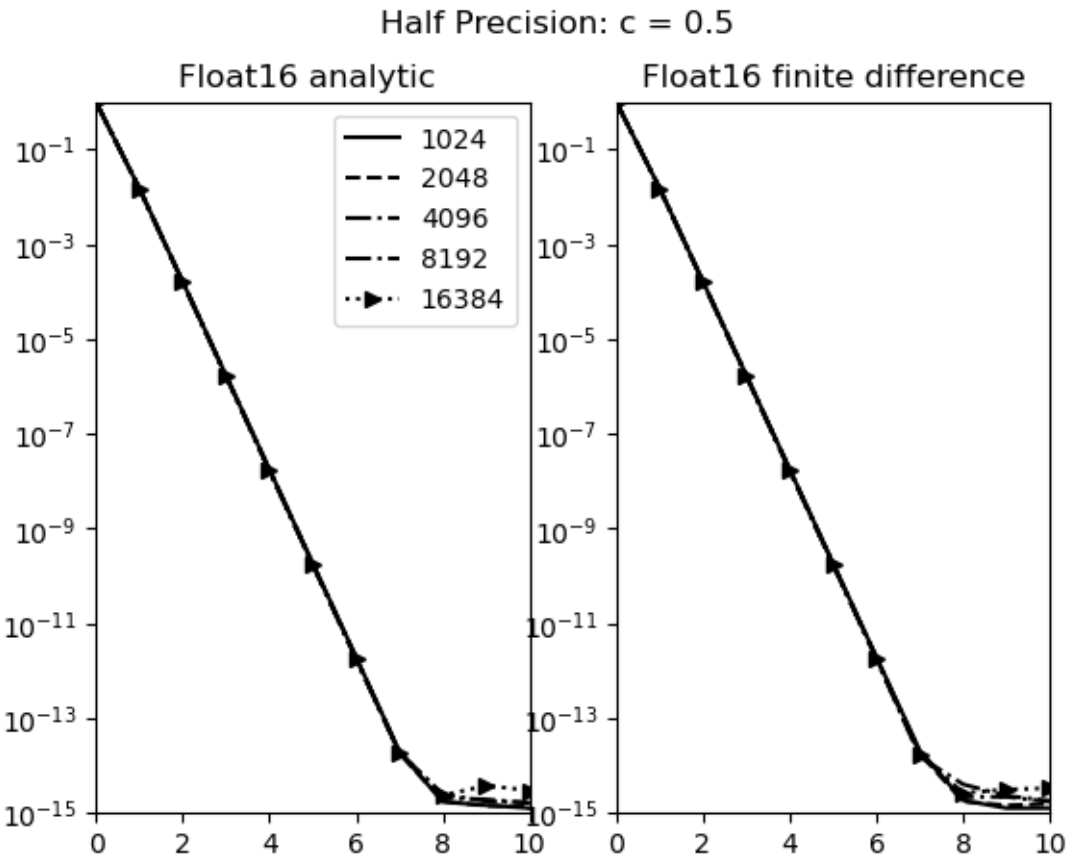
Finally, I will duplicate the tables and plots in the paper with the precomputed data in the Complete_Data directory. I'll cd to that directory and make the plots with **data_harvest.jl**.

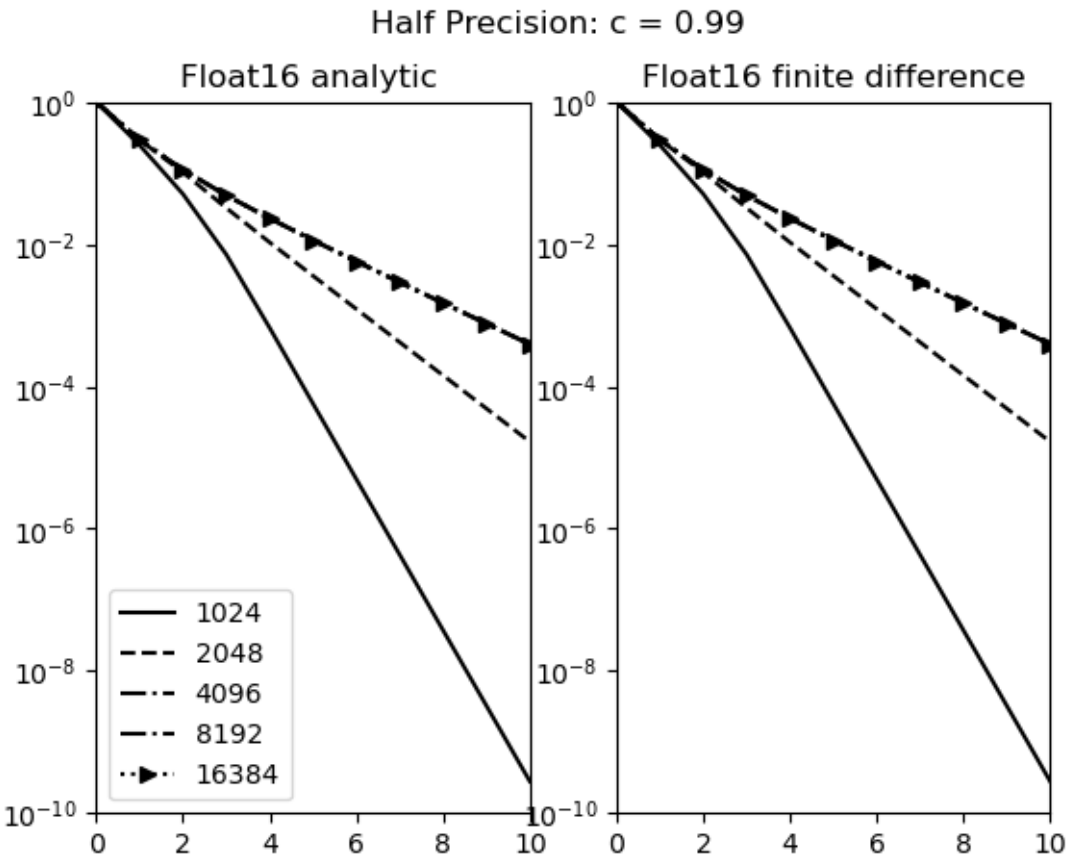
[16]:
```
cd(MPRdir)
cd("Complete_Data")
data_harvest()
```

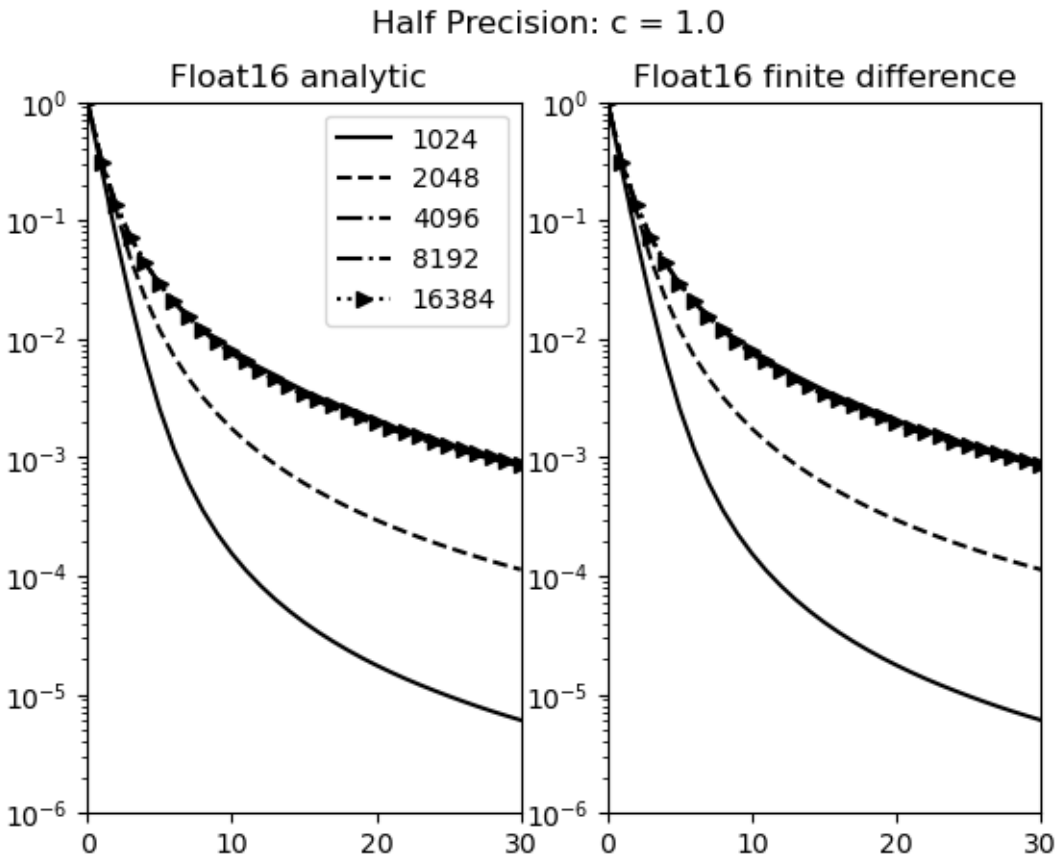

Single and Double Precision: c = 0.5
Float64 analytic
Float64 finite difference
Float32 analytic
Float32 finite difference
1024
2048
4096
8192
16384


Single and Double Precision: c = 0.99
Float64 analytic
Float64 finite difference
Float32 analytic
Float32 finite difference
1024
2048
4096
8192
16384


Single and Double Precision: c = 1.0
Float64 analytic
Float64 finite difference
Float32 analytic
Float32 finite difference
1024
2048
4096
8192
16384


Half Precision: c = 0.5
Float16 analytic
Float16 finite difference
1024
2048
4096
8192
16384
10^{-1}
10^{-3}
10^{-5}
10^{-7}
10^{-9}
10^{-11}
10^{-13}
10^{-15}
0
2
4
6
8
10


Half Precision: c = 0.99
Float16 analytic
Float16 finite difference
1024
2048
4096
8192
16384



The three LaTeX tables are pretty vivid demonstrations that using a half-precision Jacobian is a poor idea. The columns are ratios of successive residual norms. Those ratios are supposed to go to zero if the convergence is q-superlinear. In the half precision case, it is not. You can also see that from the plots at the bottom. I make the tables with three calls to **MakeTable.jl** which is contained in the file **TableData.jl**.

[17]:
```
MakeTable(.5)
```

```
\begin{tabular}{llllll}
        n &     1024 &     2048 &     4096 &     8192 &    16384 \\
\hline
1 & 1.51548e-02 & 1.50894e-02 & 1.50458e-02 & 1.50240e-02 & 1.50131e-02   \\
2 & 1.07397e-02 & 1.06615e-02 & 1.06107e-02 & 1.05853e-02 & 1.05726e-02   \\
3 & 1.04864e-02 & 1.04099e-02 & 1.03607e-02 & 1.03361e-02 & 1.03238e-02   \\
4 & 1.04514e-02 & 1.03766e-02 & 1.03284e-02 & 1.03043e-02 & 1.02922e-02   \\
5 & 1.04464e-02 & 1.03721e-02 & 1.03239e-02 & 1.03001e-02 & 1.02881e-02   \\
6 & 1.04530e-02 & 1.03772e-02 & 1.03263e-02 & 1.02969e-02 & 1.02816e-02   \\
7 & 1.01305e-02 & 9.79786e-03 & 1.05554e-02 & 1.05224e-02 & 1.06917e-02   \\
8 & 8.73158e-02 & 9.79760e-02 & 1.09900e-01 & 1.31948e-01 & 1.24336e-01   \\
9 & 8.46876e-01 & 7.99843e-01 & 9.21406e-01 & 7.13501e-01 & 1.60623e+00   \\
```

```
10 & 8.73518e-01 & 9.65146e-01 & 8.40415e-01 & 9.29831e-01 & 7.88260e-01   \\
\hline
\end{tabular}
```

[18]:
```
MakeTable(.99)
```

```
\begin{tabular}{llllll}
        n &     1024 &     2048 &     4096 &     8192 &    16384 \\
\hline
1 & 2.47566e-01 & 2.93830e-01 & 2.94145e-01 & 2.94107e-01 & 2.94088e-01   \\
2 & 2.05600e-01 & 3.48351e-01 & 3.77559e-01 & 3.77649e-01 & 3.77610e-01   \\
3 & 1.40806e-01 & 3.16201e-01 & 4.38451e-01 & 4.38762e-01 & 4.38723e-01   \\
4 & 9.31148e-02 & 3.28139e-01 & 4.73701e-01 & 4.73983e-01 & 4.73942e-01   \\
5 & 8.59359e-02 & 3.38481e-01 & 4.92930e-01 & 4.93227e-01 & 4.93190e-01   \\
6 & 8.65219e-02 & 3.40729e-01 & 5.03119e-01 & 5.03432e-01 & 5.03398e-01   \\
7 & 8.66721e-02 & 3.41551e-01 & 5.08435e-01 & 5.08762e-01 & 5.08726e-01   \\
8 & 8.66935e-02 & 3.41710e-01 & 5.11188e-01 & 5.11524e-01 & 5.11487e-01   \\
9 & 8.66950e-02 & 3.41756e-01 & 5.12609e-01 & 5.12950e-01 & 5.12912e-01   \\
10 & 8.66960e-02 & 3.41833e-01 & 5.13340e-01 & 5.13685e-01 & 5.13646e-01   \\
\hline
\end{tabular}
```

[19]:
```
MakeTable(1.0)
```

```
\begin{tabular}{llllll}
        n &     1024 &     2048 &     4096 &     8192 &    16384 \\
\hline
1 & 2.65171e-01 & 3.13548e-01 & 3.13867e-01 & 3.13831e-01 & 3.13812e-01   \\
2 & 2.70831e-01 & 3.84700e-01 & 4.29466e-01 & 4.29666e-01 & 4.29633e-01   \\
3 & 2.89581e-01 & 3.89214e-01 & 5.32231e-01 & 5.32478e-01 & 5.32453e-01   \\
4 & 3.23427e-01 & 4.77575e-01 & 6.09958e-01 & 6.10101e-01 & 6.10123e-01   \\
5 & 3.82508e-01 & 5.32789e-01 & 6.62135e-01 & 6.68348e-01 & 6.68369e-01   \\
6 & 4.54440e-01 & 5.98778e-01 & 7.04067e-01 & 7.12635e-01 & 7.12645e-01   \\
7 & 5.22841e-01 & 6.50593e-01 & 7.41734e-01 & 7.47025e-01 & 7.47031e-01   \\
8 & 5.84470e-01 & 6.91184e-01 & 7.70896e-01 & 7.74339e-01 & 7.74343e-01   \\
9 & 6.36332e-01 & 7.24333e-01 & 7.94007e-01 & 7.96482e-01 & 7.96485e-01   \\
10 & 6.81255e-01 & 7.51888e-01 & 8.12841e-01 & 8.14760e-01 & 8.14762e-01   \\
11 & 7.16247e-01 & 7.75225e-01 & 8.28523e-01 & 8.30083e-01 & 8.30085e-01   \\
12 & 7.46058e-01 & 7.94854e-01 & 8.41796e-01 & 8.43103e-01 & 8.43105e-01   \\
13 & 7.69637e-01 & 8.11687e-01 & 8.53176e-01 & 8.54296e-01 & 8.54298e-01   \\
14 & 7.92049e-01 & 8.26179e-01 & 8.63042e-01 & 8.64018e-01 & 8.64019e-01   \\
15 & 8.07371e-01 & 8.38661e-01 & 8.71677e-01 & 8.72536e-01 & 8.72537e-01   \\
16 & 8.23999e-01 & 8.49675e-01 & 8.79296e-01 & 8.80061e-01 & 8.80062e-01   \\
17 & 8.37356e-01 & 8.59263e-01 & 8.86068e-01 & 8.86754e-01 & 8.86755e-01   \\
18 & 8.47186e-01 & 8.67783e-01 & 8.92127e-01 & 8.92746e-01 & 8.92746e-01   \\
19 & 8.57336e-01 & 8.75404e-01 & 8.97578e-01 & 8.98140e-01 & 8.98141e-01   \\
20 & 8.66801e-01 & 8.82174e-01 & 9.02509e-01 & 9.03016e-01 & 9.03022e-01   \\
21 & 8.74347e-01 & 8.88296e-01 & 9.06990e-01 & 9.07453e-01 & 9.07460e-01   \\
```

```
22 & 8.81390e-01 & 8.93883e-01 & 9.11080e-01 & 9.11505e-01 & 9.11511e-01   \\
23 & 8.88034e-01 & 8.98900e-01 & 9.14828e-01 & 9.15219e-01 & 9.15225e-01   \\
24 & 8.93824e-01 & 9.03511e-01 & 9.18275e-01 & 9.18636e-01 & 9.18642e-01   \\
25 & 8.98283e-01 & 9.07706e-01 & 9.21455e-01 & 9.21789e-01 & 9.21795e-01   \\
26 & 9.02619e-01 & 9.11584e-01 & 9.24398e-01 & 9.24708e-01 & 9.24714e-01   \\
27 & 9.06563e-01 & 9.15125e-01 & 9.27130e-01 & 9.27418e-01 & 9.27424e-01   \\
28 & 9.10556e-01 & 9.18424e-01 & 9.29671e-01 & 9.29941e-01 & 9.29946e-01   \\
29 & 9.15203e-01 & 9.21451e-01 & 9.32043e-01 & 9.32295e-01 & 9.32300e-01   \\
30 & 9.18229e-01 & 9.24343e-01 & 9.34260e-01 & 9.34496e-01 & 9.34501e-01   \\
\hline
\end{tabular}
```

Finally, I will cd to the directory that contains the notebook.

[20]:

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

[8] ——, *Notebook for Solving Nonlinear Equations with Iterative Methods: Solvers and Examples in Julia*, 2020. IJulia Notebook.

[9] ——, *SIAMFANLEquations.jl*, 2020. Julia Package.

[10] ——, *Solving Nonlinear Equations with Iterative Methods: Solvers and Examples in Julia*, 2020. Unpublished book ms, under contract with SIAM.

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