NDA-QMC Results

April 6, 2021

Contents

1 Using the notebook

The first cell in this notebook is an invisible markdown cell with LaTeX commands. The second is a code cell that sets up the packages you'll need to load. It's best to do a **run all** before experimenting with the solvers.

2 The equation

This note book compares various solvers for the problem in the Garcia/Siewert paper [1]. My formulation of the transport problem is taken from [7]. The equation for the angular flux ψ is

$$\mu \frac{\partial \psi}{\partial x}(x,\mu) + \Sigma_t(x)\psi(x,\mu) = \frac{1}{2}\left[\Sigma_s(x)\int_{-1}^{1}\psi(x,\mu')\,d\mu' + q(x)\right] \text{ for } 0 \le x \le \tau$$

The boundary conditions are

$$\psi(0,\mu) = \psi_l(\mu), \mu > 0; \psi(\tau,\mu) = \psi_r(\mu), \mu < 0.$$

The notation is

- ψ is intensity of radiation or angular flux at point x at angle $\cos^{-1}(\mu)$
- $\phi = \phi(x) = \int_{-1}^{1} \psi(x,\mu)\, d\mu$ is the scalar flux, the 0^{th} angular moment of the angular flux. - $\tau < \infty$, length of the spatial domain. - $\Sigma_s \in C([0,\tau])$ is the scattering cross section at x - $\Sigma_t \in C([0,\tau])$ is the total cross section at x - ψ_l and ψ_r are incoming intensities at the bounds - $q \in C([0,\tau])$ is the fixed source

2.1 Discretization

The discretization is plain vanilla S_N with diamond differencing. I'm storing fluxes (and later currents) at cell edges to make the boundary conditions for NDA a bit easier to deal with.

I'm using double Gaussian quadratures for the angular mesh. This means that the angles are N-point Gaussian quadratures on (-1,0) and on (0,1) for a total of 2N angles. The function **sn_angles.jl** in the /src directory sets up the angular mesh. It uses the package **FastGaussQuadrature.jl**. We will denote the weights and notes of the angluar mesh by w_j and μ_j for $j = 1, ..., 2N$.

I'm using a uniform spatial mesh $\{x_i\}_{i=1}^{N_x}$ where

$$x_i = (i-1)dx \text{ and } dx = \tau/(N_x - 1).$$

The function **sn_init.jl** in /src sets up all the mesh data and boundary conditions. It builds a Julia named tuple for me to pass around to the solvers.

The disretization approximates the angular flux $\psi_i^j \approx \psi(x_i, \mu_j)$ and the scalar flux

$$\phi_i \approx \phi(x_i).$$

The two approximations are related (in my cell-edge oriented code) by

$$\text{Flux Equation: } \phi_i = \sum_{j=1}^{2N} \psi_i^j w_j$$

The discrete equation is, with source $S_i = \Sigma_s(x_i) f_i + q(s_i)$

$$\begin{array}{ll} \text{Transport Sweep:} & \\ \mu_j \frac{\psi_i^j - \psi_{i-1}^j}{dx} + \Sigma_t(x_i + dx/2)\frac{\psi_i^j + \psi_{i-1}^j}{2} & = \frac{S_i + S_{i-1}}{2} \text{ for } \mu_j > 0 \text{ and } \psi_1^j = \psi(0,\mu_j) \\ \mu_j \frac{\psi_i^j - \psi_{i+1}^j}{dx} + \frac{\psi_i^j + \psi_{i+1}^j}{2} & = \frac{S_i + S_{i+1}}{2} \text{ for } \mu_j < 0 \text{ and } \psi_{N_x}^j = \psi(\tau,\mu_j) \end{array}$$

So, source iteration begins with an initial iterate for the scalar flux ϕ, computes the discrete scalar flux with the equations **Transport_Sweep**, treating **f** as input data, and updates the scalar flux with the formula **Flux_Equation**. This can also be viewed as a linear equation for ϕ

$$\phi - \mathbf{K}\phi = f$$

and solved with a Krylov method like GMRES. The right side of the integral equation depends on the source q and the boundary data.

2.2 The Garcia-Siewert Example

In this example

$$\tau = 5, \Sigma_s(x) = \omega_0 e^{-x/s}, \Sigma_t(x) = 1, q(x) = 0, \psi_l(\mu) = 1, \psi_r(\mu) = 0.$$

3 Solvers

The linear and nonlinear solvers come from my Julia package SIAMFANLEQ.jl [3]. The documentation for these codes is in the Juila notebooks that accompany the package [2]. All of this is part of a book project [4]. The citations for these things may change as the book gets closer to publication. Please ask me before citing this stuff.

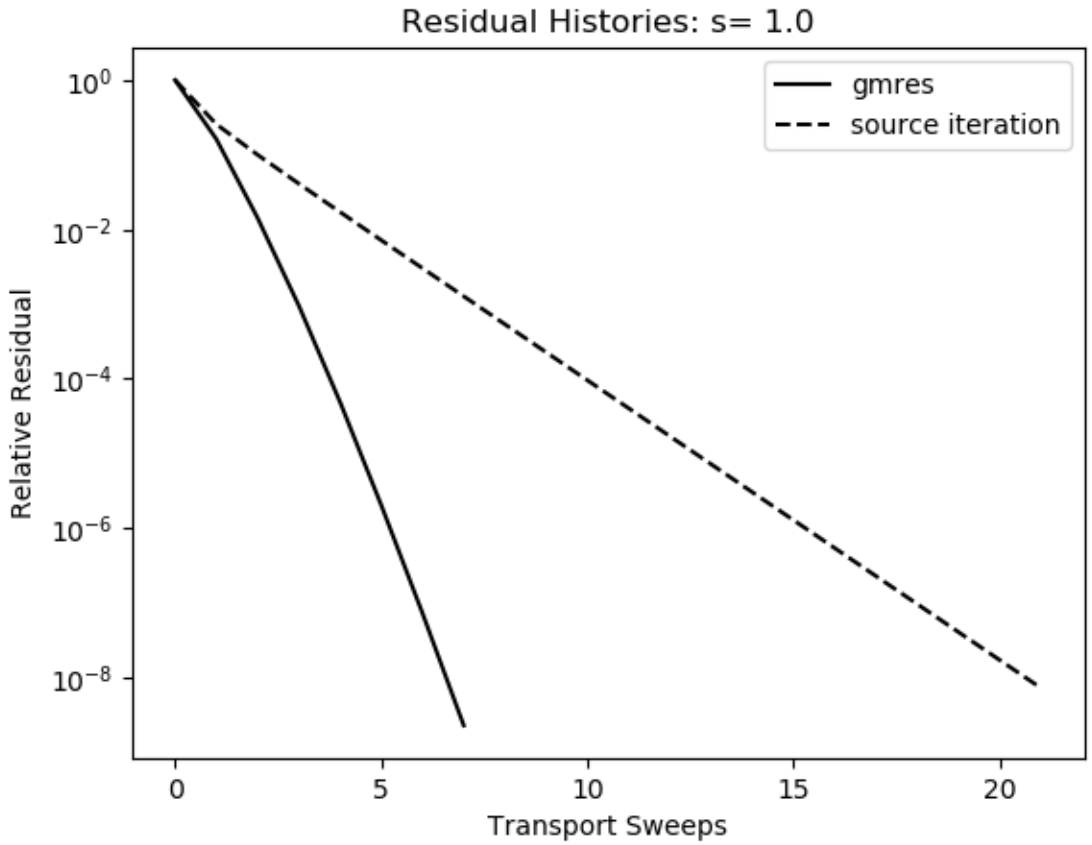
The important solvers for this notebook are the GMRES linear solver **kl_gmres.jl** and the Newton-Krylov nonlinear solver **nsoli.jl**.

3.1 GMRES and Source Iteration

The function `compare.jl` in the **src** directory runs the source iteration and GMRES, plots the residual histories, and tabulates the exit disbributions for comparision with Tables 1 and 2 in the paper. As a sanity check I print the L^∞ norm of the differences in the results from the two solvers.

Here is a run for the first columns in the tables.

[41]:
```
compare(1.0)
```



```
 mu         I(0,-mu)        I(tau,mu)
 0.05     5.89670e-01     6.07486e-06
 0.10     5.31120e-01     6.92514e-06
 0.20     4.43280e-01     9.64229e-06
 0.30     3.80307e-01     1.62338e-05
 0.40     3.32965e-01     4.38575e-05
 0.50     2.96091e-01     1.69371e-04
 0.60     2.66564e-01     5.73462e-04
 0.70     2.42390e-01     1.51281e-03
 0.80     2.22235e-01     3.24369e-03
 0.90     2.05175e-01     5.96035e-03
 1.00     1.90547e-01     9.77122e-03
Norm of result difference = 9.32897e-09
```

The functions all use precomputed data for the angles, spatial mesh, storage allocationf the angular flux, and various problem paramaters. The precomputed data lives in a named tuple `sn_data` which I create with `ns_init.jl`.

The iterations themselves use a transport sweep to do both source iteration and the liner residual computation for GMRES. In both cases a function `flux_map!` takes an input flux, does the transport sweep, and then takes the zeroth moment to return and output flux. The transport sweep is `transport_sweep.jl` it computes the angular flux from a given input flux and boundary conditions.

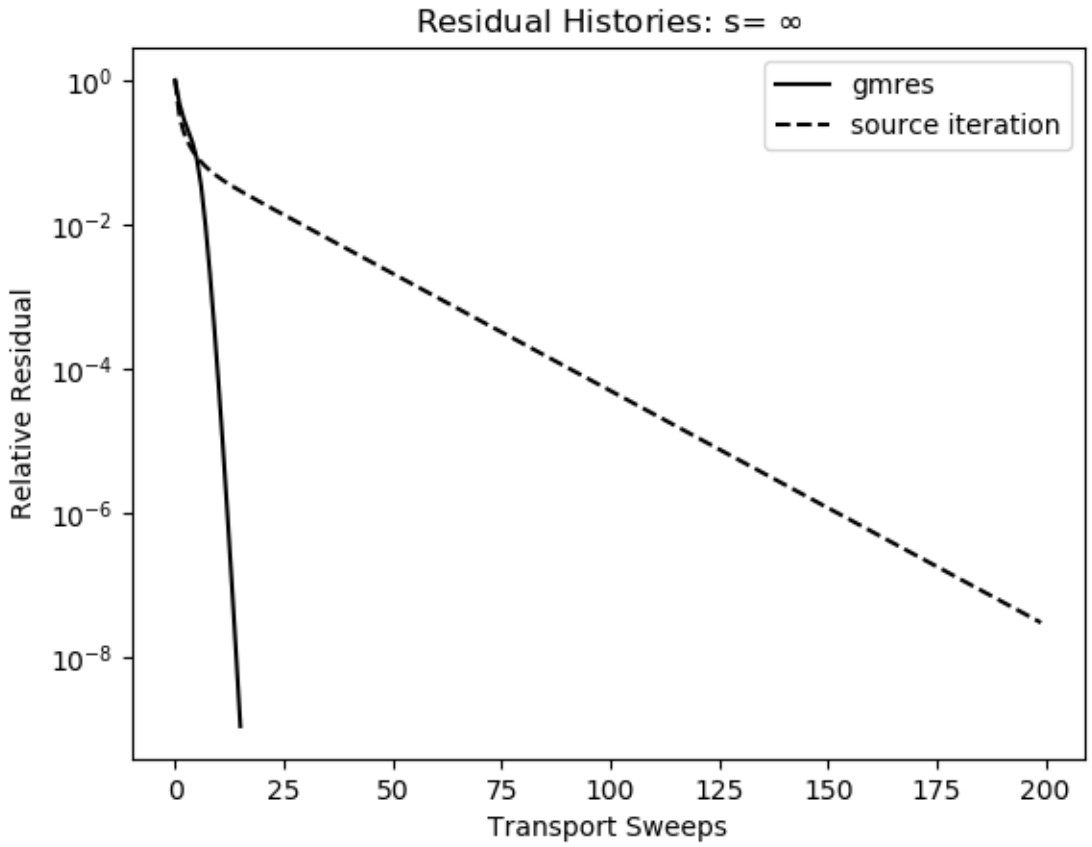
Here is function for the source iteration solver. It's pretty simple if you believe `flux_map.jl` does what I say it does. I think the QMC version of `flux_map.jl` would fit in with this pretty easiliy.

```
"""
 source_iteration(sn_data,s,tol)
 Source iteration example script for transport equation.

 This is one of the test cases from

 Radiative transfer in finite inhomogeneous plane-parallel atmospheres
 by Garcia and Siewert
 JQSRT (27), 1982 pp 141-148.

"""
function source_iteration(sn_data,s,tol=1.e-8)
    nx = 2001
    #
    # precomputed data
    #
    angles=sn_data.angles
    weihts=sn_data.weights
    itt = 0
    delflux = 1
```

```
    phi = zeros(nx)
    flux = zeros(nx)
    reshist = []
    while itt < 200 && delflux > tol
        flux = flux_map!(flux, sn_data)
        delflux = norm(flux - phi, Inf)
        itt = itt + 1
        push!(reshist, delflux)
        phi .= flux
    end
    #
    # Tabulate the exit distributions to check results.
    #
    return ( flux = flux, history= reshist)
end
```

Here are the results for the final column.

[42]:
```
compare(Inf)
```




```
mu          I(0,-mu)          I(tau,mu)
```

```
 0.05     8.97799e-01     1.02201e-01
 0.10     8.87836e-01     1.12164e-01
 0.20     8.69581e-01     1.30419e-01
 0.30     8.52299e-01     1.47701e-01
 0.40     8.35503e-01     1.64497e-01
 0.50     8.18996e-01     1.81004e-01
 0.60     8.02676e-01     1.97324e-01
 0.70     7.86493e-01     2.13507e-01
 0.80     7.70429e-01     2.29571e-01
 0.90     7.54496e-01     2.45504e-01
 1.00     7.38721e-01     2.61279e-01
Norm of result difference = 3.97115e-07
```

3.2 NDA

NDA (nonlinear diffusion acceleration) turns the linear problem into a nonlinear one. One gets an equation for the (**low-order**) flux f in terms of the moments of ψ. It is easier for me to explain this for the continuous problem. Be warned that my **boundary conditions are not standard**.

I'm taking this from [7], [6], and [5].

Given a flux $\phi(x)$ we begin by solving a **high-order** problem for an angular flux

$$\mu \frac{\partial \psi^{HO}}{\partial x} + \Sigma_t \psi^{HO}(x,\mu) = \frac{1}{2}\left[\Sigma_s \phi^{LO}(x) + q(x)\right],$$

with the same boundary conditions we used in the original problem.

The next step is to compute high-order fluxes

$$f^{HO}(x) = \int_{-1}^{1} \psi(x,\mu')\, d\mu'$$

and currents

$$J^{HO}(x) = \int_{-1}^{1} \psi(x,\mu')\mu'\, d\mu'.$$

We use these to compute

$$\hat{D} = \frac{J^{HO} + \frac{1}{3}\frac{d\phi^{HO}}{dx}}{\phi^{HO}},$$

If ϕ is the solution to the **low-order** problem,

$$\frac{d}{dx}\left[-\frac{1}{3\Sigma_t}\frac{d\phi}{dx} + \hat{D}\phi\right] + (\Sigma_t - \Sigma_s)\phi = q,$$

then ϕ is the scalar flux and we have solved the transport equation. The boundary conditions are tricky if we store fluxes and currents at cell centers. If we have cell edge fluxes, then boundary conditions

$$\phi(0) = \phi^{HO}(0), \phi(\tau) = \phi^{HO}(\tau)$$

work well.

The low-order prolbem is nonlinear because ϕ^{HO}, J^{HO} and $\hat{D}$ depend on ϕ. We can make this explicit (writing $\hat{D}(\phi)$ and that helps when it's time to write code. Write the nonlinear equation as $F(\phi) = 0$ where

$$F(\phi) = \frac{d}{dx}\left[-\frac{1}{3\Sigma_t}\frac{d\phi}{dx} + \hat{D}(\phi)\phi\right] + (\Sigma_t - \Sigma_s)\phi - q.$$

We can solve this equation efficiently with Newton-GMRES if we have a good preconditioner. We use fast solver for the high-order term in the low-order equation. The preconditioner in [7], [6], [5], is a bit more complicated.

We can also formulate the low-order problem as a fixed point problem by viewing the solution of the low-order problem ϕ^{LO} as a transformation of ϕ. So given ϕ^{LO} compute the high-order flux, current, and $\hat{D} = \hat{D}(\phi^{LO})$. Then solve

$$\frac{d}{dx}\left[-\frac{1}{3\Sigma_t}\frac{d\phi}{dx} + \hat{D}(\phi^{LO})\phi\right] + (\Sigma_t - \Sigma_s)\phi = q,$$

For ϕ. This is a **linear** equation for ϕ because ϕ^{LO} is input. Express this as

$$\phi = G(\phi^{LO})$$

We have solved the problem if $\phi = \phi^{LO}$. So a Picare or fixed point iteration is

$$\phi_{n+1} = G(\phi_n).$$

One can also solve

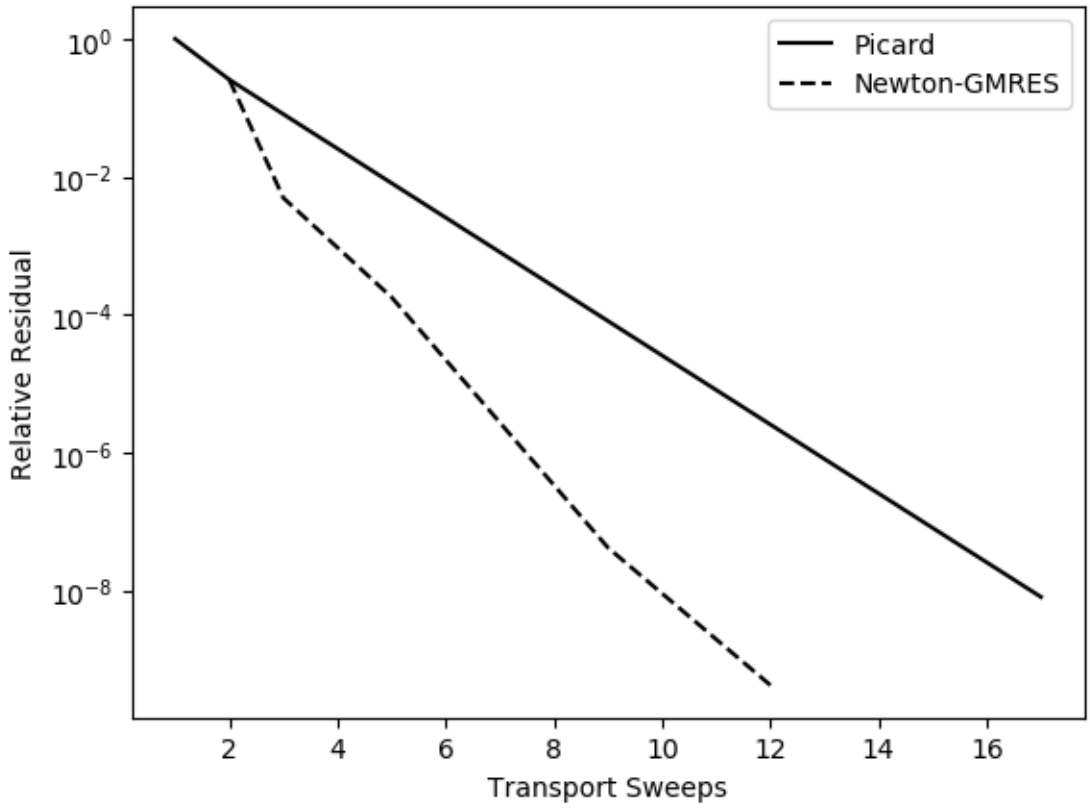
$$F(\phi) = \phi - G(\phi)$$

with a Newton-Krylov method. This is **not what we did** in [7] because MC is not deterministic and we had to use a formula for the derivative of $\hat{D}$ with respect to ϕ. Using that formula to differentiate G would be a real pain. We can do this for QMC and use a finite-difference Jacobian-vector product. The advantage is that the conditioning of the linear system is better because we precondition when we solve low-order problem.

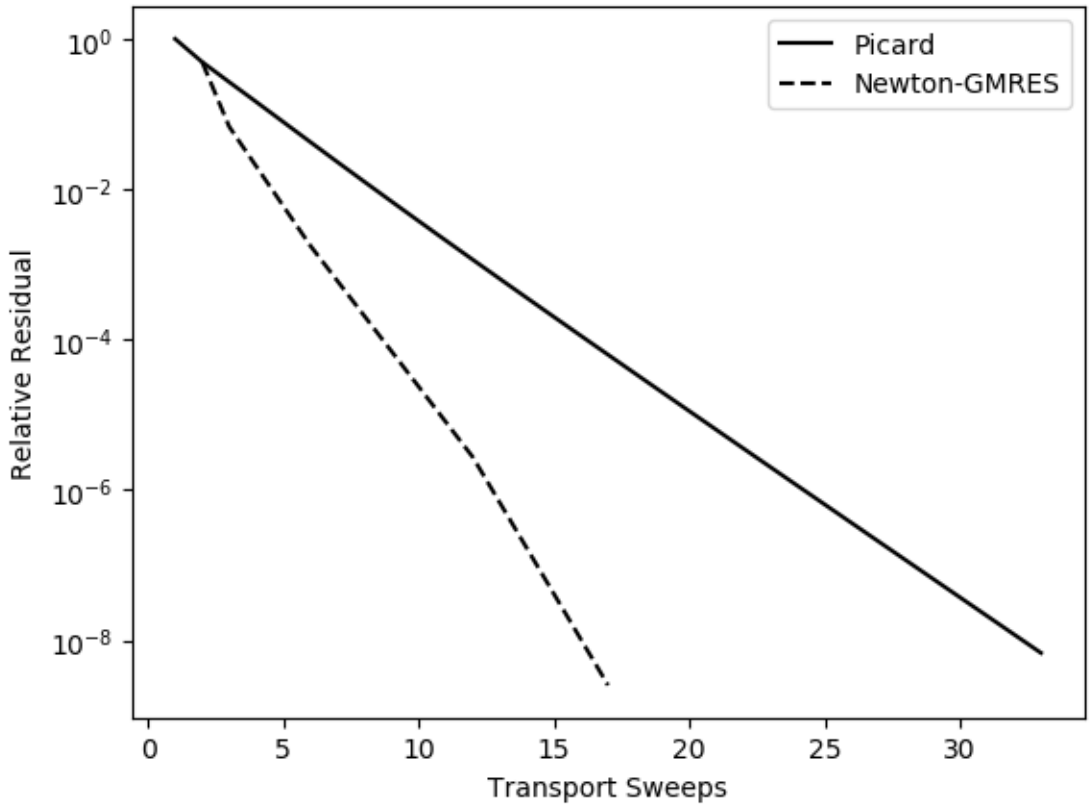
One subtle point for Newton is that we take on Picard iteration before starting Newton. The reason we do this is that the initial iterate $\phi = 0$ is really bad and Newton, while converging, takes many iterations to fix that. A single Picard sorts that out.

The plots below compare the two approaches. The limited data in this note book indicate that GMRES is best for the $s = 1$ case and NDA-Newton is best for $s = \infty$. The iteration histories overlap at the start because the first step for both is that single Picard. Note that I'm not counting Newton iterations but transport sweeps.

[43]:
```
# Run for s=1
nda_compare(1.0);
```



[44]:
```
# and for s=Inf
nda_compare(Inf);
```



4 QMC

4.1 Validation and calibration study

I'll compare the results from the SN computation to what I get from Sam's QMC code. My SN results are for a very fine spatial mesh and fine enough angular mesh. They are good to at least six figures and I will regard them as exact for this study.

I will use the SN results for the table in the Garcia-Siewert paper and get results from QMC in the following way

- For a give N and Nx I will get cell average fluxes from Sam's code.
- I will use the same code to generate the tables that I used for the SN fluxes. That code is **src/sn_tabulate.jl**
- I will take the two 11 x 2 arrays of results DataSN and DataQMC and compoute componentwise relative error with

```
Derr = (DataSN-DataQMC)./DataSN
```

- I report the value `norm(Derr,Inf)` to measure the difference.

Here one example. I have put the results from SN in two data files and can get the data by

```
DataSN = readdata(s)
```

where `s=1` or `s=Inf`. I modified Sams **qmc_test.jl** code to give me the tables. The new code is **ctk_qmc_test.jl** and that version takes N and Nx as input. I've also moved the plotting to a separate function in that file.

[45]:
```
s=1.0; N=10^3; Nx=100;
# Data from SN
DataSN=readdata(s);
# Results from QMC
DataQMC=ctk_qmc_test(N, Nx;s=1.0)
# Results
[DataSN DataQMC]
```

[45]:
```
11×4 Matrix{Float64}:
 5.89664e-01  6.07488e-06  5.68048e-01  5.85641e-06
 5.31120e-01  6.92516e-06  5.20166e-01  6.66916e-06
 4.43280e-01  9.64232e-06  4.40493e-01  9.25509e-06
 3.80306e-01  1.62339e-05  3.80323e-01  1.54463e-05
 3.32964e-01  4.38580e-05  3.34162e-01  4.09862e-05
 2.96090e-01  1.69372e-04  2.97832e-01  1.57435e-04
 2.66563e-01  5.73465e-04  2.68558e-01  5.36144e-04
 2.42390e-01  1.51282e-03  2.44491e-01  1.42477e-03
 2.22235e-01  3.24369e-03  2.24365e-01  3.07475e-03
 2.05174e-01  5.96036e-03  2.07291e-01  5.68048e-03
 1.90546e-01  9.77123e-03  1.92626e-01  9.35417e-03
```

The two columns on the left are the SN data and the ones on the right are the QMC results. It looks like there's about one figure accuracy. To find out we'll look at the component wise relative errors and take the max.

[46]:
```
Derr = (DataSN-DataQMC)./DataSN
```

[46]:
```
11×2 Matrix{Float64}:
  3.66585e-02  3.59626e-02
  2.06238e-02  3.69657e-02
  6.28623e-03  4.01595e-02
 -4.38495e-05  4.85123e-02
 -3.59715e-03  6.54811e-02
 -5.88258e-03  7.04775e-02
 -7.48276e-03  6.50794e-02
 -8.66934e-03  5.82031e-02
 -9.58627e-03  5.20846e-02
 -1.03172e-02  4.69561e-02
 -1.09141e-02  4.26823e-02
```

Looks like one to two figures. Taking the max gives me

[47]:
```
norm(Derr, Inf)
```

[47]: 7.04775e-02

So a bit better than one figure is what it is. The final part of this study is to make tables of these errors for varying N and Nx. The tables make complete sense, as I will explain after I show the results for `s=1.0`. The code I use for this is the **qmc_vs_sn** function in the file **src/validate.jl**. This thing takes a while to run and I'm not comopletely sure why. The next section on benchmarking explains my confusion.

[48]:
```
qmc_vs_sn();
```

```
Nx\N :      1000         2000         4000         8000
  50 :  1.40799e-01  1.36020e-01  1.34429e-01  1.35653e-01
 100 :  7.04174e-02  6.56657e-02  6.48763e-02  6.50648e-02
 200 :  4.74375e-02  3.26008e-02  3.19790e-02  3.20454e-02
 400 :  5.14380e-02  1.99563e-02  1.59583e-02  1.60570e-02
 800 :  5.42732e-02  2.20369e-02  1.30132e-02  8.19996e-03
1600 :  5.66738e-02  2.31124e-02  1.45774e-02  4.41259e-03
```

To understand the table we should first think of the idealized situation where the cell averages are from the **exact** scalar flux. In that case all the collumns would be the same because $N = \infty$ in that case. The row error would decay by a factor of **four** as Nx doubles because the fixed source problem I solve to make the tables is essentially a second order integration rule. But when you look at the **last** column, you see only first order convergence. **Why is that?** I may have a bug in my code, but the same table made for SN looks exactly like second order convergence.

For the earlier columns, there are not enough samples to fully populate the cells for the larger values of Nx, so the error stagnates. You see this on row three of the first column, for example.

If you look at the rows, you'll see the O(1/N) error reduction from QMC. The last row (Nx=1600) does this well. The row just above (Nx = 800) shows that there is no benefit in increasing N once the fixed Nx has done the integration as well as it can. The Nx = 800 row shows reductios by roughly a factor as N –> 2N until N = 4000, then it stops because you've extracted all Nx=800 is able to give.

So the tables make some sense, but there are still things I do not understand.

The case $s = \infty$ is the same. Here's the table.

[49]:
```
qmc_vs_sn(;s=Inf);
```

```
Nx\N :      1000         2000         4000         8000
  50 :  5.67112e-02  2.21674e-02  1.18319e-02  1.27520e-02
 100 :  5.38208e-02  2.13868e-02  1.15210e-02  6.25819e-03
 200 :  5.42957e-02  2.18667e-02  1.26402e-02  4.56562e-03
 400 :  5.12239e-02  2.05227e-02  1.31853e-02  4.52620e-03
 800 :  9.17873e-02  1.76236e-02  1.42885e-02  4.13352e-03
1600 :  1.22759e+28  2.03980e-02  1.39896e-02  3.98528e-03
```

The $s = \infty$ case is a harder problem and we need larger values of N to make progress. That last entry in column 1 tells me that I can't make sense of the results for the low value of N. The bottom line on this section is that things look reasonable, but ...

Why does the error look first order in 1/Nx? This is what would happen if I left something out of a sum or missed and index in loop. But when I use the cell average fluxes from SN, I get second order and when I use the QMC cell averages fluxes (and change nothing else!) I get first order. Have I missed something?

4.2 Timings

I use benchmark tools to see how the cost of QMC varies with N and Nx. The timings seem to tell me that the compute time is $O(N \times Nx)$. I do not understand why it's the product and not the sum? **Can someone explain this to me?** In the experiment I increase N by a factor of two and the time goes up by that factor. Same results with Nx, so it looks like the time is proportional to the product.

[50]:
```
N=1000; Nx=50;
println("Base Case")
@btime ctk_qmc_test($N,$Nx);
N=2000; Nx=50;
println("Double N")
@btime ctk_qmc_test($N,$Nx);
N=1000; Nx=100;
println("Double Nx")
@btime ctk_qmc_test($N,$Nx);
```

```
Base Case
  46.579 ms (110368 allocations: 21.22 MiB)
Double N
  97.090 ms (220368 allocations: 42.37 MiB)
Double Nx
  86.876 ms (110370 allocations: 29.66 MiB)
```

4.3 QMC questions

So I've got a few questions about these results. Most of this is coming from my ignorance of the details of QMC.

- I expected the errors to be $O(1/N) + O((1/Nx)^2)$, with very different prefactors in the O-terms, of course. Instead I see $O(1/N) + O(1/Nx)$. Is there theory for this?
- The timings tell me that the cost of the computation is $O(N \times Nx)$. I expected $O(N)$ with Nx haveing very little effect. Why is this?
- Sam, the benchmark runs tell me QMC is allocating a lot. Can you fix this by, for example, preallocating some storage in the initialization. You might also look at the performance part of the Julia documentation to see if things like `@inbounds` and `@simd` might help the code run faster. The relevand part of the Julia manual is at https://docs.julialang.org/en/v1/manual/performance-tips/

4.4 QMC and NDA

What we need from QMC are

- cell average fluxes and currents, which I think we have and
- cell average spatial derivatives for the high-order flux.

As you can see from [7], it was not entirely trivial to get the spatial derivatives.

Any ideas? I am ready to try a hack job once I have your QMC code.

What I need from QMC is something that takes the cell average low-order flux and returns cell average high-order fluxes and currents. I'm pretty sure I can conform to the way you want me to tell you how many cells and the number of samples.

4.5 Appendix: Docstrings for kl_gmres and nosli

Nothing exciting here. I'll just fire up the help screens.

[51]:
```
?kl_gmres
```

[51]:
```
klgmres(x0, b, atv, V, eta, ptv=nothing; klstore=zeros(1,1); orth = "cgs2",
side="right", lmaxit=-1, pdata=nothing)

Gmres linear solver.  Handles preconditioning and restarts.  Uses gmres_base which
is completely oblivious to these things.

The deal is

Input:

x0:  initial iterate, this is usually zero for nonlinear solvers

b:  right hand side (duh!)

atv:  matrix-vector product which depends on precomputed data pdta I expect you
to use pdata most or all of the time, so it is not an optional argument, even if
it's nothing (at least for now).  If your mat-vec is just A*v, you have to write a
function where A is the precomputed data.  API for atv is av=atv(v,pdata)

V: Preallocated n x K array for the Krylov vectors.  I store the initial normalized
residual in column 1, so you have at most K-1 iterations before gmres_base returns
a failure.  kl_gmres will handle the restarts and, if lmaxit > 0, keep going until
you hit lmaxit GMRES iterations.

eta:  Termination happens when ||b - Ax|| <= eta || b ||

ptv:  preconditioner-vector product, which will also use pdata.  The default is
nothing, which is no preconditioning at all.  API for ptv is px=ptv(x,pdata)

Keyword arguments

klstore:  You may at some point have the option of giving me some room for the
vectors gmres needs to store copies of x0 and b, which I will not overwrite and
a couple of vectors I use in the iteration.  If you're only doing a linear solve,
it does no harm to let me allocate those vectores in klgmres.  If the solver is
inside a loop, you should allocate this storage.  nsoli and ptscoli allocate this
without your having to do anything.  Right now I'm not clear on an efficient way to
do this.
```

pdata: precomputed data. The default is nothing, but that ain't gonna work well for nonlinear equations.

orth: your choice of the wise default, classical Gram-Schmidt twice, or something slower and less stable. Those are classical once (really bad) or a couple variants of modified Gram-Schmidt. mgs2 is what I used in my old matlab codes. Not terrible, but far from great.

side: left or right preconditioning. The default is "right".

lmaxit: maximum number of linear iterations. The default is -1, which means that the maximum number of linear iterations is K-1, which is all V will allow without restarts. If lmaxit > K-1, then the iteration will restart until you consume lmaxit iterations or terminate successfully.

Other parameters on the way.

Output:

A named tuple (sol, reshist, lits, idid)

where

sol= final result reshist = residual norm history lits = number of iterations idid = status of the iteration true -> converged false -> failed to converge

Examples: In these examples you have the matrix and use

```
function atv(x, A)
    return A * x
end
```

to compute the matvec.

Three dimensional problem. Will converge in the correct three iterations only if you orthogonalize with CGS twice.

```
julia> function atv(x, A)
           return A * x
       end
atv (generic function with 1 method)

julia> A = [0.001 0 0; 0 0.0011 0; 0 0 1.e4];

julia> V = zeros(3, 10); b = [1.0; 1.0; 1.0]; x0 = zeros(3);

julia> gout = kl_gmres(x0, b, atv, V, 1.e-10; pdata = A);

julia> gout.reshist
4-element Array{Float64,1}:
 1.73205e+00
 1.41421e+00
 6.72673e-02
 1.97712e-34
```

```
julia> norm(b - A*gout.sol,Inf)
1.28536e-10
```

Integral equation. Notice that pdata has the kernel of the `operator and we do the matvec directly. Just like the previous example. We put the grid information and, for this artifical example, the solution in the precoputed data.`

```
julia> function integop(u, pdata)
           K = pdata.K
           return u - K * u
       end
integop (generic function with 1 method)

julia> function integopinit(n)
           h = 1 / n
           X = collect(0.5*h:h:1.0-0.5*h)
           K = [ker(x, y) for x in X, y in X]
           K .*= h
           sol = [usol(x) for x in X]
           f = sol - K * sol
           pdata = (K = K, xe = sol, f = f)
           return pdata
       end
integopinit (generic function with 1 method)

julia> function usol(x)
           return exp.(x) .* log.(2.0 * x .+ 1.0)
       end
usol (generic function with 1 method)

julia> function ker(x, y)
           ker = 0.1 * sin(x + exp(y))
       end
ker (generic function with 1 method)

julia> n=100; pdata = integopinit(n); ue = pdata.xe; f=pdata.f;

julia> u0 = zeros(size(f)); V = zeros(n, 20); V32=zeros(Float32,n,20);

julia> gout = kl_gmres(u0, f, integop, V, 1.e-10; pdata = pdata);

julia> gout32 = kl_gmres(u0, f, integop, V32, 1.e-10; pdata = pdata);

julia> [norm(gout.sol-ue,Inf) norm(gout32.sol-ue,Inf)]
1×2 Array{Float64,2}:
 4.44089e-16  2.93700e-07
```

```
julia> [gout.reshist gout32.reshist]
4×2 Array{Float64,2}:
 1.48252e+01  1.48252e+01
 5.52337e-01  5.52337e-01
 1.77741e-03  1.77742e-03
 1.29876e-19  8.73568e-11
```

[52]:
```
?nsoli
```

```
search: nsoli
nda_nsoli
nsol
nsolsc
```

[52]:

```
nsoli(F!, x0, FS, FPS, Jvec=dirder; rtol=1.e-6, atol=1.e-12,
           maxit=20, lmaxit=-1, lsolver="gmres", eta=.1,
           fixedeta=true, Pvec=nothing, pside="right",
           armmax=10, dx = 1.e-7, armfix=false, pdata = nothing,
           printerr = true, keepsolhist = false, stagnationok=false)
```

)

C. T. Kelley, 2021

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

You must allocate storage for the function and the Krylov basis 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 the Krylov basis. It is an N x m matrix where you plan to take at most m-1 GMRES iterations before a restart.

- Jvec: Jacobian vector product, If you leave this out the default is a finite difference directional derivative.

  So, FP=Jvec(v,FS,x) or FP=Jvec(v,FS,x,pdata) returns FP=F'(x) v.

  (v, FS, x) or (v, FS, x, pdata) must be the argument list, even if FP does not need FS. One reason for this is that the finite-difference derivative 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 or Float32 and nsoli will do the right thing. Float16 support is there, but not working well.

If the Jacobian is reasonably well conditioned, you can cut the cost of orthogonalization and storage (for GMRES) in half with no loss. There is no benefit if your linear solver is not GMRES or if othogonalization and storage of the Krylov vectors is only a small part of the cost of the computation. So if your preconditioner is good and you only need a few Krylovs/Newton, reduced precision won't help you much.

Keyword Arguments (kwargs):

rtol and atol: relative and absolute error tolerances

maxit: limit on nonlinear iterations

lmaxit: limit on linear iterations. If lmaxit > m-1, where FPS has m columns, and you need more than m-1 linear iterations, then GMRES will restart.

The default is -1. This means that you'll take m-1 iterations, where size(V) = (n,m), and get no restarts.

lsolver: the linear solver, default = "gmres"

Your choices will be "gmres" or "bicgstab". However, gmres is the only option for now.

eta and fixed eta: eta > 0 or there's an error

The linear solver terminates when ||F'(x)s + F(x) || <= etag || F(x) ||

where

etag = eta if fixedeta=true

etag = Eisenstat-Walker as implemented in book if fixedeta=false

The default, which may change, is eta=.1, fixedeta=true

Pvec: Preconditioner-vector product. The rules are similar to Jvec So, Pv=Pvec(v,x) or Pv=Pvec(v,x,pdata) returns P(x) v where P(x) is the preconditioner. You must use x as an input even if your preconditioner does not depend on x

pside: apply preconditioner on pside, default = "right". I do not recommend "left". See Chapter 3 for the story on this.

armmax: upper bound on step size reductions in line search

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

If you use pdata in any of F!, Jvec, or Pvec, you must use in in all of them.

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, ikfail), the number of functions/Jacobian-vector prods/steplength reductions/linear solver failures at each iteration. Linear solver failures DO NOT mean that the nonlinear solver will fail. You should look at this stat if, for example, the line search fails. Increasing the size of FPS and/or lmaxit might solve the problem.

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

- 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.
```

```
Examples:  Simple 2D problem.  You should get the same results as for nsol.jl
because GMRES will solve the equation for the step exactly in two iterations.
Finite difference Jacobians and analytic Jacobian-vector products for full
precision and finite difference Jacobian-vector products for single precision.
```

```
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> function JVec(v, fv, x)
       jvec=zeros(2,);
       p=-sin(x[1]+x[2])
       jvec[1]=v[1]+cos(x[2])*v[2]
       jvec[2]=p*(v[1]+v[2])
       return jvec
       end
JVec (generic function with 1 method)

julia> x0=ones(2,); fv=zeros(2,); jv=zeros(2,2); jv32=zeros(Float32,2,2);

julia> jvs=zeros(2,3); jvs32=zeros(Float32,2,3);

julia> nout=nsol(f!,x0,fv,jv; sham=1);

julia> kout=nsoli(f!,x0,fv,jvs,JVec; fixedeta=true, eta=.1, lmaxit=2);

julia> kout32=nsoli(f!,x0,fv,jvs32; fixedeta=true, eta=.1, lmaxit=2);

julia> [nout.history kout.history kout32.history]
5×3 Array{Float64,2}:
 1.88791e+00  1.88791e+00  1.88791e+00
 2.43119e-01  2.43120e-01  2.43119e-01
 1.19231e-02  1.19231e-02  1.19231e-02
 1.03266e-05  1.03261e-05  1.03273e-05
 1.46416e-11  1.40862e-11  1.45457e-11
```

[ ]:

References

[1] R.D.M. Garcia and C.E. Siewert, *Radiative transfer in finite inhomogeneous plane-parallel atmospheres*, J. Quant. Spectrosc. Radiat. Transfer, 27 (1982),

pp. 141-148.

[2] C. T. KELLEY, *Notebook for Solving Nonlinear Equations with Iterative Methods: Solvers and Examples in Julia.* https://github.com/ctkelley/NotebookSIAMFANL, 2020. IJulia Notebook.

[3] ——, *SIAMFANLEquations.jl.* https://github.com/ctkelley/SIAMFANLEquations.jl, 2020. Julia Package.

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

[5] D. A. KNOLL, H. PARK, AND K. SMITH, *A new look at nonlinear acceleration*, Nuclear Science and Engineering, 99 (2008), pp. 332-334.

[6] ——, *Application of the Jacobian-free Newton-Krylov method to nonlinear acceleration of transport source iteration in slab geometry*, Nuclear Science and Engineering, 167 (2011), pp. 122-132.

[7] JEFF WILLERT, C. T. KELLEY, D. A. KNOLL, AND H. K. PARK, *Hybrid deterministic/Monte Carlo neutronics*, SIAM J. Sci. Comp., 35 (2013), pp. S62-S83.