An Efficient Implementation of Sparse Approximate Inverses and Sparse Approximate Maps in Python

Mark Latvakoski

December 18, 2023

Abstract

This Project explains the process of converting a set of Matlab code to python. The Matlab code includes a method to recycle preconditioners by creating maps between a preconditioner and the next iteration of an iterative method. This project would like to implement a working and efficient version of this code in Python, so we can see the code running in a parallel environment, among other conditions python supports

1 Intro to SAMs

Well conditioned matrices are the key to quick solutions of many iterative methods. Such methods solve systems of linear equations. Applications of solving linear systems are widespread, and include many mathematical, and physical applications, including the flow of groundwater.[5] Data taken here will likely not be well conditioned. This ill-conditioned data can cause large numbers of iterations in iterative methods, or depending on the method, cause them not to converge. Preconditioners solve this issue. Through a few matrix multiplications, preconditioners change an ill conditioned matrix into an approximately equivalent well conditioned matrix.[2] However, preconditioners have a cost. Finding a good preconditioner can be more costly than the amount of time it saves. A good practice to prevent this cost is conditioner recycling where we use one preconditioner for several iterations

of the method by mapping one preconditioner to the next.[2]

There are several benefits to using maps to recycle preconditioners:

1. Maps can be approximated. To a certain extent, gaining some error by discarding values in the map below a certain threshold of importance can save a lot of time. Discarding these values makes a more sparse map, which will take less time to use to map an ill conditioned matrix to a previous preconditioner. This also makes them optimal for storage in efficient matrix storage formats such as coordinate format.

2. calculation of these maps have calculations with parts that can be done in parallel. This is very useful for saving even more time in each iteration.

2 Python, Pytorch introduction

The purpose of this project is to recreate the method described above [2] in Python. Python has several packages such as Tensorflow, Keras, and Pytorch. Pytorch is a Machine Learning framework that has ideal features for the preconditioner recycling described above.[7] This includes:

1. Pytorch has built in Linear algebra operations under the linalg package due to the packages need to work with tensors through the progression of a neural network. These are similar to familiar NumPy linalg operations. This is useful because the Process of computing a SAM involves solving a small linear system. We can use Pytorch's built in lstsq to solve this system[7]

2. To support the creation of Neural Networks, Pytorch has extensive parallel processing capabilities. Pytorch works with NIVIDA's Cuda or AMD's ROCm to support use of the system's GPU for its fast arithmetic throughput and large parallel processing capability.[6] This is an important feature of the package as it will allow expensive computations to run faster.

I'm familiar with python packages NumPy and SciPy. They contain extensive support for numerical computing in python, and look like they will be easy to translate into similar methods offered by packages with parallel computing

support such as PyTorch or CuPy. Therefore the implementation of the python code will start using NumPy and SciPy, then upgrade as needed.

3 SAMs in MATLAB

The developers of the SAM preconditioner recycling algorithm have shared the equivalent MATLAB code that shows the creation of the maps from an input system. The core functionality of the code is here, and this process is

```
for j = 1:n
    G(1:nnz_LS(j),1:nnz_M(j)) = Jac(nz_LS{j},nz_M{j});
    M(1:nnz_M(j)) = G(1:nnz_LS(j),1:nnz_M(j))\J0(nz_LS{j},j);
    rowM(cntrM+1:cntrM+nnz_M(j)) = nz_M{j};
    colM(cntrM+1:cntrM+nnz_M(j)) = j;
    valM(cntrM+1:cntrM+nnz_M(j)) = M(1:nnz_M(j));
    cntrM = cntrM+nnz_M(j);
end
% Assemble the map
MM = sparse(rowM(1:cntrM),colM(1:cntrM),valM(1:cntrM));
```

done every time a map is calculated. The code shows a few steps. First, the input matrix needs to be broken down into smaller pieces. Then a system is solved for each of these pieces. The solution to this system is then added to a list of values which are later assembled into the Map. We keep this general structure in the translated python code.

4 Implementation

Given the size and complexity of using the maps to recycle a preconditioner, I started the process of translation by implementing just the creation of the maps in python, without the recycling part. This is the extent of my contribution. The code is ready for the next steps of the project which I will describe later.

```
# k = 0
Jac = mat_contents["saveSys"][0][0]       You, 2 days ago • k = 1
n = Jac.shape[0]
I, J, _ = find(Jac) #dont need values
findJ = coo_matrix((np.ones_like(I), (I, J)), shape=Jac.shape, dtype=bool)
PP = findJ.todense().astype(bool)
PP2 = np.matmul(PP, PP) #equivelent of double multiplication

nnzMM = np.count_nonzero(PP2)
rowM = np.zeros((2*nnzMM,1))
colM = np.zeros((2*nnzMM,1))
valM = np.zeros((2*nnzMM,1))
J0 = Jac

#other preprocessing
nz_M = []
nnz_M = []
nz_LS = []
nnz_LS = []
for j in range(n):
    nz_M.append(np.nonzero(PP[j])[1])
    nnz_M.append(len(nz_M[j]))
    nz_LS.append(np.nonzero(PP2[j])[1])
    nnz_LS.append(len(nz_LS[j]))
```

The code above shows the preprocessing section of the code. This sets up a few things we need for calculating the maps. This includes the sparsity pattern, number and position of non-zeros in each row, and the initialization of a couple lists.

```
def SAM(k, n, Jac, J0, nnz_LS, nnz_M, nz_LS, nz_M):
    max_col = max(nnz_M)
    max_row = max(nnz_LS)
    G = np.zeros((max_row, max_col))
    M = np.zeros((max_row))

    rowM = []
    colM = []
    valM = []
    for j in range(n):
        #get the submatrix from G
        G = [[Jac[nz_LS[j][row], nz_M[j][col]] \
              for col in range(nnz_M[j])] \
                for row in range(nnz_LS[j])]

        #np.linalg.solve does not work for not Square A. use least squares
        M = np.linalg.lstsq([g_col[:nnz_M[j]] \
                for g_col in G[:nnz_LS[j]]],\
                [col[j] for col in J0.todense()[nz_LS[j]].tolist()])
        M = M[0] #just solution (also gives residual)

        rowM = np.append(rowM, nz_M[j][:nnz_M[j]])
        jarray = np.full((nnz_M[j]), j, dtype=np.int64)
        colM = np.append(colM, jarray)
        valM = np.append(valM, np.array(M[:nnz_M[j]]))

    MM = coo_array((valM, (rowM.astype(np.int64), colM.astype(np.int64))))
```

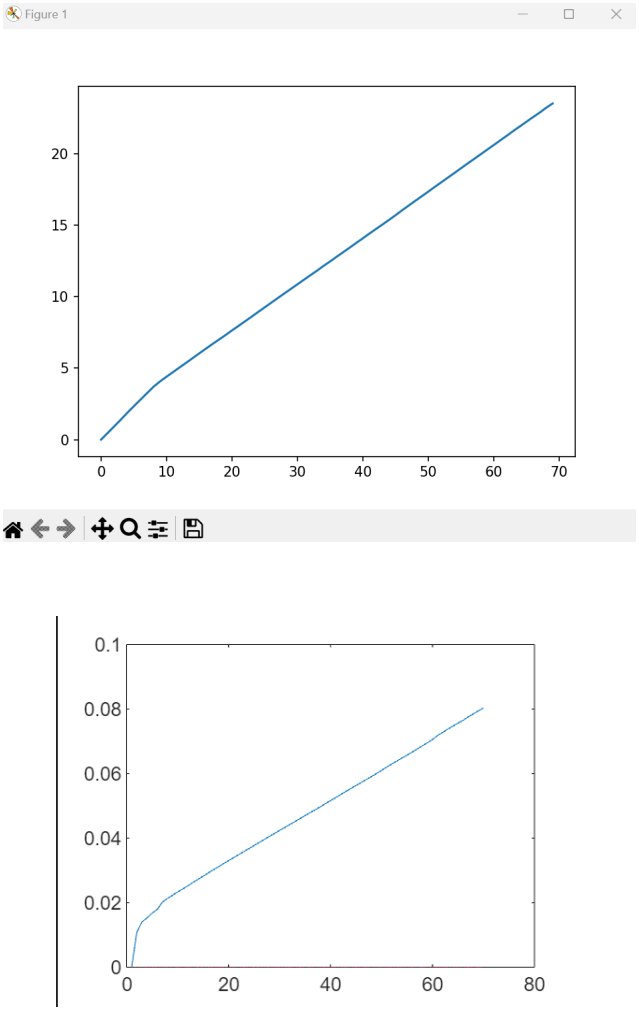
This code above shows the core functionality as represented in python. It closely follows the Matlab code, but uses NumPY and SciPY sparse formats, and operations instead of Matlab's. Based on documentation, both Pytorch and CuPy both support drop in alternatives to the Least Squares solution, and either could be used with their respective data structures. I was unable to test either due to hardware incompatibilities (CuPy does not work well with AMD) and time limitations. This will be a future step of the project. All project source code can be found at https://github.com/MWLatva/SAMs398

5 Challenges, Future

1. One of the future steps here will be to test the effects of adding either PyTorch or CuPy functions to the python code. I think both of these packages have the opportunity to improve the speed of map computation.

2. Add the rest of the code. Currently the code just computes the maps from a list of systems given. In order to be used as a preconditioner recycler, significant additions would need to be made to the code.

Ultimately the Python translation is a good start to this project, but it still has a lot of work left. Matlab is faster than Python here. With python taking some 20x longer.



Top: Python, Bottom: Matlab. With seconds on Y axis, Iterations on X.

Works Cited

1. K. Ahuja, B. K. Clark, E. de Sturler, D. M. Ceperley, and J. Kim, *Improved scaling for quantum Monte Carlo on insulators*, SIAM J. Sci. Comput., 33 (2011), pp. 1837–1859.

2. A. Carr, E. De Sturler, S. Gugercin, *Preconditioning Parameterized Linear Systems*, SIAM J. Sci. Comput., 43 (2021), pp. A2242–2267

3. H. Anzt, T. Huckle, J. Brackle, and J. Dongarra ,; Incomplete sparse approximate in- verses for parallel preconditioning, *Parallel Comput.*, 71 (2018), pp. 1–22.

4. K. Osawa, S. Ishikawa, R. Yokota, S. Li, T. Hoefler, ASDL: A Unified Interface for Gradient Preconditioning in PyTorch *HOOML2022: Order Up! The Benefits of Higher-Order Optimization in Machine Learning* Paper 15 (2022)

5. M. Cardiff and W. Barrash, 3-D Transient hydraulic tomography in unconfined aquifers with fast drainage response, *Water Resources Research*, 47 (2011).

6. J. Sanders, E. Kandrot, Jul 19, 2010. CUDA By Example: An Introduction to General Purpose GPU Programming. *Addison-Wesley Professional*

7. Preferred Networks (2015) CuPy - NumPy and SciPy for GPU, *CUPY.dev*

8. PyTorch Contributors (2023) PyTorch Documentation *pytorch.org*

Retrievals, and Annotations

1. Retrieved from https://arxiv.org/pdf/1008.5113.pdf
   Quantum Monte Carlo paper that introduces method of recycling preconditioners

2. Introduces the method I will be writing in python, and includes Matlab pseudo code to create these SAM's, provides reference to materials that will be useful in understanding these methods, and their uses.

3. Retrieved from https://www.sciencedirect.com/science/article/abs/pii/S016781911730176X
   Paper that talks about parallelizing the SAI process, which will be useful later on in my python implementation when I may have to write parallel code.

4. Retrieved from https://order-up-ml.github.io/papers/15.pdf
   Talks about preconditioning at pytorch. Even though they are focusing on a much different topic, they include code snippets of preconditioner type work in python. This has a chance of being useful later.

5. Retrieved from https://agupubs.onlinelibrary.wiley.com/doi/full/10.1029/2010WR010367
   Cited source on a real life use case for this type of method.

6. Book that explains GPU computing which CUDA uses to run fast calculations. Both CuPy and Pytorch primarily use this to accelerate their operations (assuming compatibility)

7. CuPy docs which were essential in learning more about the package, and ultimately for finding out that it would not work on my system. Found that I could set up a container to work on this instead.

8. Pytorch docs which were used to find how Pytorch could help this project. Will be useful in continued work on this project