Journal of Scientific Research & Reports
7(3): 207-217, 2015, Article no.JSRR.2015.202
ISSN: 2320-0227

SCIENCEDOMAIN *international*
www.sciencedomain.org

Using Python to Solve the Navier-Stokes Equations - Applications in the Preconditioned Iterative Methods

Jia Liu[1*], Lina Wu[2] and Xingang Fang[3]

[1] *Department of Mathematics and Statistics, University of West Flroida, Pensacola, Florida, 32514, USA.*

[2] *Department of Mathematics, Borough of Manhattan Community College, The City University of New York, New York, NY 10007, USA.*

[3] *Department of Computer Science, University of West Flroida, Pensacola, Florida, 32514, USA.*

Article Information
DOI: 10.9734/JSRR/2015/17346
Editor(s):
(1) Narcisa C. Apreutesei, Technical University of Iasi, Romania.
Reviewers:
(1) Anonymous, China.
(2)Ette Harrison Etuk, Department of Mathematics Computer Science, Rivers State University of Science and Technology, Nigeria.
Complete Peer review History:
http://www.sciencedomain.org/review-history.php?iid=1127&id=22&aid=9129

Original Research Article

Received: 9th March 2015
Accepted: 3rd April 2015
Published: 5th May 2015

Abstract

This article describes a new numerical solver for the Navier-Stokes equations. The proposed solver is written in Python which is a newly developed language. The Python packages are built to solve the Navier-Stokes equations with existing libraries. We have created discretized coefficient matrices from systems of the Navier-Stokes equations by the finite difference method. In addition we focus on the preconditioned Krylov subspace iterative methods in the linearized systems. Numerical results of performances for the Preconditioned iterative methods are demonstrated. The comparison between Python and Matlab is discussed at the end of the paper.

Keywords: Python; Navier-Stokes equations; Iterative methods; Preconditioner.
2010 Mathematics Subject Classification: 65Y15; 65F10; 68N15

**Corresponding author: E-mail: jliu@uwf.edu;*

1 Introduction

The numerical solvers for Partial Differential Equations (PDEs) are developed greatly in recent years. Most of them are based on the traditional languages such as C, C++, or Fortran etc. However, those languages are high performances but lower level languages. In general Languages like C, C++ require relative strong programming backgrounds. Therefore more scientists start to develop the numerical applications for PDEs in Python due to its huge potential advantages in the computational mathematics areas.

1.1 Why Python?

Python is an interpreted, high-level, object-oriented, dynamic general-purpose programming language. Since python 1.0 was published in 1994, it has been one of the most popular programming languages (3rd place on PYPL index, Feb 2014). The advantages of the Python programming language include rapid development, excellent readability, fast learning curves, huge community support and great cross-platform capability. It is extensively used in desktop application developments, web application developments, scriptings, system managements as well as scientific computations. In the scientific computation area, Python has the following important features compared with Matlab:

- Code quality: Compared with the Matlab scripting language, Python is a full stack language supporting multiple programming paradigms including object-oriented, imperative, functional programming, etc. It is capable of more computational tasks. With the unique indent block, Python code is more readable. The Python module system is a sophisticated way to organize source code files compared to Matlab which lacks the support on namespace. The zero-based bracket array indexing is much more compatible with most other programming languages. In addition Python is a signal processing system compared to the Matlab which is a one-based parenthesis system.
- Portability: Python is a free and open source on all platforms. The run time environment is light-weighted (239M EPD Python full scientific computing distribution) compared to the multi-gigabyte Matlab. It can be easily deployed anywhere while the installation of Matlab is limited by the license.
- Flexibility: Python is a perfect glue language. It is not hard to find or write Python wrapper to provide API (application programming interface) of existing high performance software packages written in C, C++, FORTRAN or even Matlab. Thus, it provides a way to quickly develop applications by reusing existing code. In the case of Matlab's closed ecosystem, scientists have to rely on the packages only from MathWorks, the company which developed Matlab. Unfortunately, the underlying algorithms are proprietary and expensive.
- Performance: Though performance is a major concern of Python given its interpreted, dynamic nature. This problem can be usually circumvent by delegating most computational tasks to high performance packages written in C, FORTRAN. Cython, the variation of standard Python (CPython), provided a way to take the advantages of both the rapid development capability of Python and the high performance of C language by translating a superset of Python syntax to C code. In many scenarios, it will provide comparable performance to scientific packages.
- Availability of packages: Python and the huge number of extension packages are freely available (41311 on PyPI, a Python package repository on Mar. 2014). There are also high quality packages for all aspects of scientific computing. Numpy/Scipy are popular Python numerical packages that provide similar functionality and performance to Matlab. Matplotlib is a very useful package for plotting. Numpy/Scipy and Matplotlib together can cover most of the tasks done on Matlab. Moreover, there are many other very useful Python packages for

scientific computing including the scientific notebook iPython, the statistic package panda, the cross-platform GUI toolkits pyQt, wxPython and GTK, the all-in-one scientific Python distribution Pythonxy and canopy express (former EPD Python), the test framework unittest and nose.

- Cost: Matlab is very expensive for the researchers while Python and corresponding packages are totally free. It is more favorable for students who would like to learn the numerical analysis with no cost.

With the advanced features of Python, we are able to develop some efficient and easy-to-use softwares to solve PDEs. In this paper, we focus on using Python to solve the PDEs arising from the incompressible flow problems, especially the Navier-Stokes equations. We are more interested in the applications of the preconditioned Krylov subspace iterative methods. We will compare the performances between Python and Matlab.

1.2 The Naviver-Stokes Equations

The Navier-Stokes equations, which are named after Claude-Louis Navier and George Gabriel Stokes, come from the motion of fluid substances. The equations are important with both academic and economic interests. The Navier-Stokes equations model the movement of the fluid from various types, such as the weather, ocean currents, water flows in a pipe and air flows around a wing. They are also used as the models to study the blood flow, the design of power stations, the analysis of pollution. The incompressible Navier-Stokes equations are given by the following PDEs:

$$\frac{\partial \mathbf{u}}{\partial t} - \nu \Delta \mathbf{u} + (\mathbf{u} \cdot \nabla)\,\mathbf{u} + \nabla p = \mathbf{f} \qquad \text{in} \quad \Omega \times (0, T] \tag{1.1}$$

$$\nabla \cdot \mathbf{u} = 0 \qquad \text{in} \quad \Omega \times [0, T] \tag{1.2}$$

$$\mathcal{B}\mathbf{u} = \mathbf{g} \qquad \text{on} \quad \Omega \times [0, T] \tag{1.3}$$

$$\mathbf{u}(\mathbf{x}, 0) = \mathbf{u}_0 \qquad \text{in} \quad \Omega \tag{1.4}$$

Here the variable $\mathbf{u} = \mathbf{u}(\mathbf{x}, t) \in R^d$, where $d = 2$, or $d = 3$ is a vector-valued function representing the velocity of the fluid, and the scalar function $p = p(\mathbf{x}, t) \in R$ represents the pressure. $\Omega \subset R^d$,where $d = 2$, or $d = 3$. For an example, Ω is defined as $[0, 1] \times [0, 1] \subset R^2$. $T > 0$ is a constant. Here $\mathcal{B}$ is the boundary condition operator such as the Dirichlet Boundary condition or the Neumann's boundary condition.

Assume $\Omega \subset R^2$. Denoting $\mathbf{u} = (u, v)$, and $\mathbf{f} = (f_1, f_2)$, we can write the equations (1.1)–(1.2) in scalar form as follows:

$$\begin{aligned}
\frac{\partial u}{\partial t} - \nu \Delta u + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + \frac{\partial p}{\partial x} &= f_1\,, \\
\frac{\partial v}{\partial t} - \nu \Delta v + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + \frac{\partial p}{\partial y} &= f_2\,, \\
\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} &= 0\,.
\end{aligned}$$

The pressure field, p, is determined up to an additive constant. To uniquely determine p we may impose some additional condition, such as

$$\int_{\Omega} p \, d\mathbf{x} = 0 \,. \tag{1.5}$$

The source function $\mathbf{f}$ is given on a d-dimensional domain denoted by $\Omega \subset R^d$, with $\partial\Omega$ the boundary of Ω. Here $\nu > 0$ is a given constant called the kinematic viscosity, which is $\nu = O(Re^{-1})$. Here Re is the Reynolds number: $Re = \frac{VL}{\nu}$, where V denotes the mean velocity and L is the diameter of Ω (see [1]).

Numerical algorithms for the computational solutions of the Navier-Stokes equations have been discussed for several decades. This field still remains active in the research area. In this paper, we will implement the efficient solvers for the Navier-Stokes equations using Python. Though we already have some numerical solvers or softwares for solving PDEs on line, very few of them consider the preconditioned iterative solvers for the Navier-Stokes equations. We are the first to develop a Python package which enables us to solve the Navier-Stokes equations using different preconditioning techniques. We will compare the performances of different precondtioners and the convergence rates of the iterative methods. During our numerical experiences, we have used many mature libraries from the Web in the package, including NumPy ($http://numpy.scipy.org$), Scipy ($www.scipy.org$), and Dolfin ($www.fenics.org/dolfin$).

2 Materials and Methods

2.1 Creating Matrices for the Navier-Stokes Equations

The unsteady (time-dependent) case leads to sequences of saddle point systems when fully implicit time-stepping schemes are used. For example, we can discretize the time derivative term $\frac{\partial \mathbf{u}}{\partial t}$ using backward Euler or Crank-Nicolson schemes; see [2]. For the space discretization, either finite element methods or finite difference methods will work. We first consider the Marker-and-Cell (MAC) discretization, which is due to Harlow and Welch (1965). See [3] for more details. The particularity of MAC scheme is the location of the velocity and pressure unknowns. Pressures are defined at the center of each cell and the velocity components are defined at the cell edges (or cell faces in 3D). Such an arrangement makes the grid suitable for a control volume discretization. We also can use Dolfin to generate the matrices for the Navier-Stokes equations using the finite element method.

First we can use backward Euler's method to discretize (1.1)–(1.2) with respect to the time and use the Picard's iteration to linearize equations (1.1)–(1.2). We obtain the a sequence of linear problems which are also called the Oseen equations:

$$\begin{aligned} \alpha\mathbf{u} - \nu\Delta\mathbf{u} + (\mathbf{u}_k \cdot \nabla)\,\mathbf{u} + \nabla p = \mathbf{f} \qquad &\text{in} \qquad \Omega \times (0,T] \\ \nabla \cdot \mathbf{u} = 0 \qquad &\text{in} \qquad \Omega \times [0,T] \end{aligned}$$

Here α is $O(\delta t)$, where δt is the time step. And $\mathbf{u}_k$ is the solution which is obtained from the previous step. The initial guess $\mathbf{u}_0$ can be the solution from the Stokes equations which are

$$\begin{aligned} \alpha\mathbf{u} - \nu\Delta\mathbf{u} + \nabla p = \mathbf{f} \qquad &\text{in} \qquad \Omega \times (0,T] \\ \nabla \cdot \mathbf{u} = 0 \qquad &\text{in} \qquad \Omega \times [0,T] \end{aligned}$$

The following finite difference expressions are utilized (for the two dimensional case) when we discretize the equations with respect to the space:

$$
\begin{aligned}
\frac{\partial^2 u}{\partial x^2}|_{j+1/2;k} &= \frac{u_{j+3/2,k} - 2u_{j+1/2,k} + u_{j-1/2,k}}{\Delta x^2}\,, \\
\frac{\partial^2 u}{\partial y^2}|_{j+1/2;k} &= \frac{u_{j+1/2,k+1} - 2u_{j+1/2,k} + u_{j+1/2,k-1}}{\Delta y^2}\,, \\
\frac{\partial^2 v}{\partial x^2}|_{j+1;k-1/2} &= \frac{v_{j+2,k-1/2} - 2v_{j+1,k-1/2} + v_{j,k-1/2}}{\Delta x^2}\,, \\
\frac{\partial^2 v}{\partial y^2}|_{j;k-1/2} &= \frac{v_{j,k+1/2} - 2v_{j,k-1/2} + v_{j,k-3/2}}{\Delta y^2}, \\
\frac{\partial u}{\partial x}|_{j+1/2;k} &= \frac{u_{j+3/2,k} - u_{j-1/2,k}}{\Delta x}\,, \\
\frac{\partial u}{\partial y}|_{j+1/2;k} &= \frac{u_{j+1/2,k+1} - u_{j+1/2,k-1}}{\Delta y}\,, \\
\frac{\partial v}{\partial x}|_{j+1;k-1/2} &= \frac{v_{j+2,k-1/2} - v_{j,k-1/2}}{\Delta x}\,, \\
\frac{\partial v}{\partial y}|_{j;k-1/2} &= \frac{v_{j,k+1/2} - u_{j,k-3/2}}{\Delta y}\,, \\
\frac{\partial p}{\partial x}|_{j+1/2;k} &= \frac{p_{j+1,k} - p_{j,k}}{\Delta x}\,, \\
\frac{\partial p}{\partial y}|_{j;k-1/2} &= \frac{p_{j,k} - p_{j,k-1}}{\Delta y}\,.
\end{aligned}
$$

With the above scheme, we obtain the following linear system $\mathcal{A}\mathbf{x} = \mathbf{b}$, where

$$
\mathcal{A} = \begin{bmatrix} \mathbf{A}_1 & \mathbf{0} & \mathbf{B}_1^T \\ \mathbf{0} & \mathbf{A}_2 & \mathbf{B}_2^T \\ \mathbf{B}_1 & \mathbf{B}_2 & \mathbf{0} \end{bmatrix}, \tag{2.1}
$$

where $\mathbf{A}_1$, $\mathbf{A}_2$are discrete reaction-convection-diffusion operators. $\mathbf{A}_1$ is the approximation of $\alpha u - \nu\Delta u + u\,\frac{\partial u}{\partial x} + v\,\frac{\partial u}{\partial y}$, and $\mathbf{A}_2$ is the approximation of $\alpha v - \nu\Delta v + v\,\frac{\partial v}{\partial x} + v\,\frac{\partial v}{\partial y}$. Here $\mathbf{B}_1 \approx \frac{\partial}{\partial x}$ and $\mathbf{B}_2 \approx \frac{\partial}{\partial y}$. We denote the matrix $\mathcal{A}$ in the block form:

$$
\mathcal{A} = \begin{bmatrix} \mathbf{A} & \mathbf{B}^T \\ \mathbf{B} & \mathbf{0} \end{bmatrix}, \tag{2.2}
$$

where

$$
\mathbf{A} = \begin{bmatrix} \mathbf{A}_1 & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_2 \end{bmatrix}, \tag{2.3}
$$

and $\mathbf{B} = [\mathbf{B}_1 \;\; \mathbf{B}_2]$.

The right hand side $\mathbf{b} = [f_1, f_2, 0]^T$. The discretized linear system $\mathcal{A}\mathbf{x} = \mathbf{b}$ is the major system we focus on. Notice here the matrix $\mathcal{A}$ is a large sparse matrix. Therefore the computation involving the matrix $\mathcal{A}$ should always consider the sparsity and use sparse operations.

Notice here the linear system $\mathcal{A}\mathbf{x} = \mathbf{b}$ is the linear system we need to solve at each Picard's iteration. Therefore, at each k-iteration in the Picard's iteration, we have the solution $\mathbf{x}$. Under certain conditions, the sequence $\mathbf{x}_{k=1}^{\inf}$ will converge to the solution of the nonlinear problem (1.1) to (1.4).

2.2 Numerical Solvers

Currently, our most important application is to play with the different numerical solvers for the Saddle point system $\mathcal{A}\mathbf{x} = \mathbf{b}$ with the coefficient matrix 2.1. We focus on Krylov subspace iterative methods: such as the General Minimum Residual Method (GMRES) [4] and the Biconjugate gradient stabilized method (BICGSTAB) [5]. For both methods, the preconditioning techniques are important for the Navier-Stokes equations. The preconditioner $\mathcal{P}$ is a matrix which approximates $\mathcal{A}$ in some yet-undefined sense. It is obvious that the linear system

$$
\mathcal{P}^{-1}\mathcal{A}\mathbf{x} = \mathcal{P}^{-1}\mathbf{b} \tag{2.4}
$$

has the same solution as

$$
\mathcal{A}\mathbf{x} = \mathbf{b}\,. \tag{2.5}
$$

However (2.4) may be easier to solve, which means GMRES or other Krylov subspace iterative methods may converge faster. System (2.4) is preconditioned from the left, and we can also preconditioned from the right:

$$\mathcal{A}\mathcal{P}^{-1}\mathbf{y} = \mathbf{b},\ \ \mathbf{x} = \mathcal{P}^{-1}\mathbf{y}\,. \tag{2.6}$$

Many research work has been published in the area of the preconditioning technique. Unfortunately, there is no universal "ideal" preconditioner for all linear systems. See a thorough discussion of the precondtioning techniques in [6,7], ect. The choice of the preconditioner is strongly case dependent which is the most challenging topic in the iterative solvers. We implemented the GMRES iterative solvers with no preconditioner, and with different choices of preconditions. The performance of the iterative solver and the preconditioners can be explored. At this stage, we have already tested the following different precondtioners with GMRES.

2.2.1 Block diagonal preconditioner

$$\mathcal{P} = \begin{bmatrix} \hat{A} & O \\ O & I \end{bmatrix}.$$

where $\hat{A}$ is the approximation of A (in most our cases, $\hat{A}$ is A or $A + \alpha I$ where α is time variable, and $\alpha = O(\frac{1}{\delta t})$).

2.2.2 Block triangular preconditioner

$$\mathcal{P} = \begin{bmatrix} \hat{A} & O \\ B & I \end{bmatrix}.$$

Then the block LU factorization form of block triangular reconditioner is

$$\mathcal{P} = \begin{bmatrix} I_n & O \\ O & I_m \end{bmatrix} \begin{bmatrix} \hat{A} & O \\ O & I_m \end{bmatrix} \begin{bmatrix} I_n & -\hat{A}^{-1}B \\ O & I_m \end{bmatrix} \tag{2.7}$$

2.2.3 Hermitian and Skew Hermitian (HSS) Preconditioner

$$\mathcal{P} = \frac{1}{2r}\hat{\mathcal{H}}\hat{\mathcal{S}}$$

where

$$\mathcal{H} = \begin{bmatrix} \frac{A+A^T}{2} & O \\ O & O \end{bmatrix}, \mathcal{S} = \begin{bmatrix} \frac{A-A^T}{2} & B^T \\ B & O \end{bmatrix}.$$

$$\hat{\mathcal{H}} = \mathcal{H} + rI_{n+m}, \hat{\mathcal{S}} = \mathcal{S} + rI_{n+m}.$$

As we can see, $\mathcal{H}$ is the symmetric part of the coefficient matrix $\mathcal{A}$, i.e.

$$\mathcal{H} = \frac{\mathcal{A} + \mathcal{A}^T}{2},$$

and S is the skew-symmetric part of the coefficient matrix $\mathcal{A}$, i.e.

$$\mathcal{S} = \frac{\mathcal{A} - \mathcal{A}^T}{2}.$$

In order to make H and S invertible, we shift both matrices by adding rI_{n+m}, where $r \neq 0$ is an arbitrary parameter, and I_{n+m} is a identity matrix of size $n + m$.

The performances of the above three preconditioners are discussed in both [6,8]. Here we focus on the application of Python in those numerical solvers and the outcomes from the software package.

3 Numerical Experiments

In this section we report on several numerical experiments meant to illustrate the behavior of the Python applications on the preconditioned GMRES in a wide range of model problems. We consider Navier-Stokes problems in the two dimensional spaces.

3.1 Numerical Solutions

In this session, we will show the example using Python to solve the Navier-Stokes equations and plot the solution. We use a few libraries: numpy and matplotlib. Here numpy is a library that provides matrix operations and matplotlib provides the 2D plotting library that we could use to plot the results. We consider the two-dimensional lid-driven cavity problem. This is a classical problem from the Navier-Stokes equations. We take a square cavity with sides of unit length and kinematic viscosity $\nu = 0.05$. The initial condition is u,v,$p = 0$ everywhere, and the boundary conditions are:

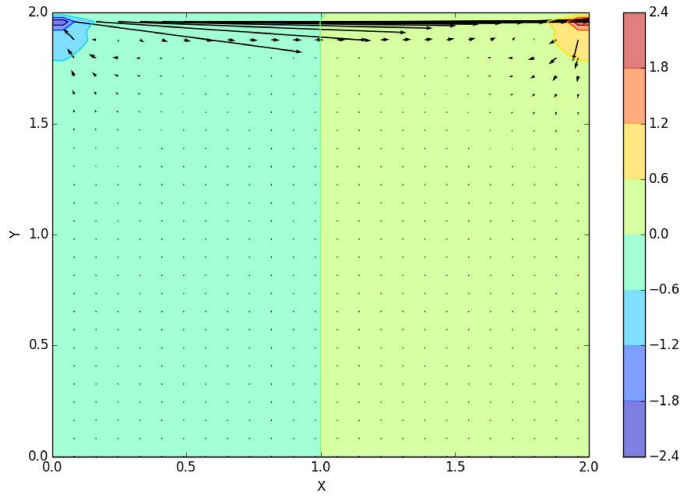

Figure 1: Simulation Results for the Driven Cavity Problem t = 1

Table 1: Sample Output: Iteration counts for the 2D Cavity problem

	64×64	128×128	256×256
Block Diagonal Preconditioner	31	33	34
Block triangular Preconditioner	16	17	18
HSS Preconditioner	49	52	55

- $u = 1$ at $y = 2$;
- u,$v = 0$ on the other boundaries;
- To satisfy the continuity, we need to correct the pressure term and here are the boundary conditions for the pressure: $\frac{\partial p}{\partial y} = 0$ at $y = 0$;$p = 0$ at $y = 2$; and $\frac{\partial p}{\partial x} = 0$ at $x = 0,\ 2$

Figure 1 - 4 show the solutions of the above driven cavity problems with viscosity $\nu = 0.1$. Figure 1 is the solution as $t = 1$, Figure 2 is the solution as $t = 200$, Figure 3 is the solution as $t = 500$, and Figure 4 is the solution as $t = 2000$. We can see as t goes to around 500, the system gradually to stabilize.

3.2 2D Equations with Different Preconditioners

In this session, we explore the iteration properties of the 2D Navier-Stokes equations with different preconditioners. We consider the Oseen equation in this case. We apply the Krylov subspace iterative methods: GMRES with three different preconditioner we introduced in the previous section. We use the library scipy, numpy and matplotlib. In Table 1 we present results for the iteration counts for 2D Oseen problem with different precondtitioners. Here the exact solver is used in each preconditioning action. The mesh size increases from 64×64 to 256×256.

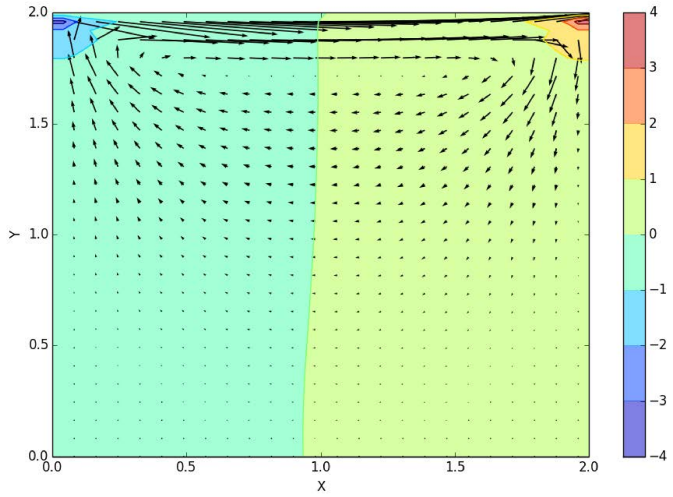

Figure 2: Simulation Results for the Driven Cavity Problem t = 200

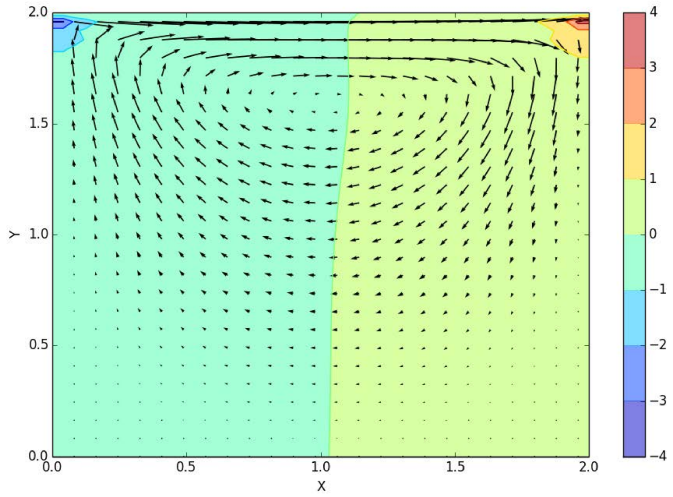

Figure 3: Simulation Results for the Driven Cavity Problem t = 500

Figure 5 shows the convergence curves for those three preconditioned GMRES methods.

We did a few numerical experiments using Matlab and Python. The performance of both languages are quite close.

1. The length of the codes are similar. Both Matlab and Python are script languages. They are easy to use and easy to implement.
2. Both Matlab and Python can obtain the images of the solutions, plot the

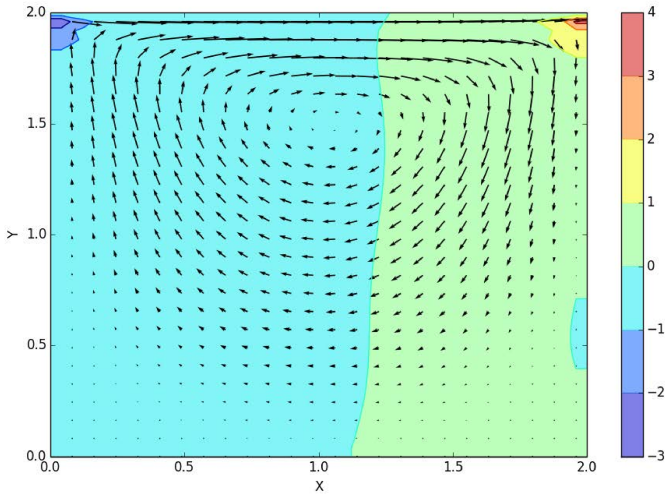

Figure 4: Simulation Results for the Driven Cavity Problem t = 2000

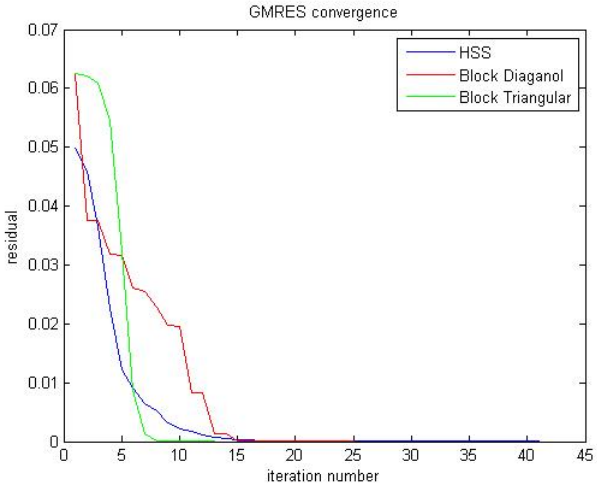

Figure 5: Sample Output: GMRES iteration convergence for the Driven Cavity Problem

eigenvalues, velocity streamlines etc.

3. The speed are also similar. Here is the comparison. We run the 2D Oseen problem with constant wind using Matlab and Python on the same computer. We used HSS preconditioning technique with GMRES iterations. Table 2 shows the cpu time for both languages. The CPU time for Python is higher than the CPU time in Matlab. The reason is due to the optimization of the sparsity operations in Matlab. We currently are working on improving the sparsity operations in Python too.

Table 2: CPU time for the 2D Oseen problem with constant wind

Matrix Size	Matlab CPU time	Python CPU time
40×40	0.029	0.05
176×176	0.13	0.45
736×736	2.69	36.8
3008×3008	83	119

A good way to explore the capabilities of this package in Python is to download the codes at $http : www.uwf.edu/jliu/research/Python$. The Python codes are clean and simple but they are easy to learn. With the development of our Python package, scientists and students can explore the properties of the preconditioned Krylov subspace methods. The other software in the preconditioning technique area is the IFISS developed by David Silvester (School of Mathematics, University of Manchester), Howard Elman (Computer Science Department, University of Maryland), and Alison Ramage (Department of Mathematics and Statistics, University of Strathclyde). See [1]. This software is written in Matlab and it is a graphical package of the numerical study of the incompressible flows. This is a good way to study and explore the numerical methods of the Navier-Stokes equations. But you need to have Matlab installed which costs around one thousand dollars. It is expensive for the students who would like to explore the numerical solutions for the Navier-Stokes equations. On the other hand, Python is free. In addition, Python is very similar with Matlab in many features. Both of them are script languages and easy to learn. In our numerical experiments, we would like to show that we can use this free high-level language to develop a package. Our package will help to implement the Navier-Stokes equations of finite difference methods, explore a range of preconditioned Krylov subspace solvers, provide a visualization tool and etc. Notice that with Python alone, the calculation speed is not competitive with the softwares based on C/C++/Fortran, or even Matlab. But with the combination of C/C++/Fortran, the Python script is able to run the meshes that have millions of nodes on desktop computers, see [9].

4 Conclusions

In this article, we have developed a numerical solver for the Navier-Stokes equations using Python. We use the preconditioned iterative methods to solve the problem and we explore the effectiveness and performances of the different preconditioners for the Krylov subspace methods. The numerical solutions are analyzed for their basic properties. We also compare the performances between Python and Matlab. The major contribution of this project is to develop a free, efficient software and numerical algorithms for the Navier-Stokes equations. It is the first time to study the preconditioning techniques in Python codes. The development of the Python package may benefit both scientists and students to analyze the numerical solvers of the Navier-Stokes equations, especially in observing and developing the preconditioning techniques.

Acknowledgment

The research from the first author was supported by Faculty Scholarly and Creative Activity Award 2013-2014 at the University of West Florida. Support for this project was provided by the Research and Sponsored Program at the University of West Florida. The research from the second author was partially supported by PSC-CUNY Grant. Support for her project was provided by PSC-CUNY Award, jointly funded by The Professional Staff Congress and The City University of New York.

Competing Interests

Authors have declared that no competing interests exist.

References

[1] Elman HC, Silvester DJ, Wathen AJ, Finite Elements and fast iterative solvers: With applications in incompressible fluid dynamics.Oxford University Press; 2005.

[2] Turek S. Efficient solvers for incompressible flow problems: An algorithmic and computational approach. Lecture Notes in Computational Science and Engineering. 1999;6. Springer, Berlin.

[3] Harlow FH, Welch JE. Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface. Phys. Fluids. 1965;8:2182-2189.

[4] Saad Y, Schultz MH, GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems preconditioners for the discrete steady-state Navier-Stokes equations. SIAM J. Sci. Stat. Comput. 2002;7:856-869.

[5] Van der Vorst HA. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. SIAM J. Sci. and Stat. Comput. 1992;13(2):631-644.

[6] Benzi M, Golub GH, Liesen J, Numerical solution of saddle point problems. Acta Numerica. 2005;14:1-137.

[7] Elman HC, Silvester DJ, Wathen AJ. Performance and analysis of saddle point preconditioners for the discrete steady-state Navier–Stokes equations. Numer. Math. 2002;90:665-688.

[8] Benzi M, Liu J. An efficient solver for the incompressible Navier-Stokes equations in rotation form. SIAM J. Scientific Computing. 2007;29:1959-1981.

[9] Logg A, Mardal A, Wells G, Automated solution of differential equations by the finite element method: The fenics book. Springer; 2012.

Peer-review history:
The peer review history for this paper can be accessed here (Please copy paste the total link in your browser address bar)
www.sciencedomain.org/review-history.php?iid=1127&id=22&aid=9129