Solving Sparse Linear Systems

MATH692

April 13th, 2023

Austin Smith

But First, a Diversion

- FFTs on hardware?
  - Yes! But not on your computer
  - First outlined by D. Cohen, 1976 [1]
  - Seems like it was first implemented in 1988 [2]
- Current applications seem to be very ML focused
  - Specialized chips
  - Convolutional neural nets [3]

Analog is the Future

- Veritasium video here
- Analog FFT chip?
  - No fully analog chip yet?
  - Designed a year ago [4]

THE COMPARISON OF AVERAGE RUNNING TIME TO CALCULATE FFT WITH DIFFERENT SIZE BETWEEN MATLAB AND CIRCUIT

FFT size	MATLAB(ms)	Circuit(μs)
8	7.638	0.23
16	10.122	0.25
32	11.281	0.26
64	12.346	0.27
128	13.419	0.28
256	13.841	0.28
512	15.231	0.29
1024	17.957	0.30

Gaussian Elimination on Hardware?

- Gaussian Elimination over GF(2)
  - GF(2) or $\mathbb{Z}/2\mathbb{Z}$, Galois field with two elements
  - Field with the smallest number of elements
  - Typically used as 0 and 1, or true and false
- Regular Gaussian Elimination is $O(n^3)$, $c = 1/3$
  - Software GE over GF(2) still is $O(n^3)$, $c = 1/4$
  - Hardware GE over GF(2) is at worst $O(n^2)$ in time and space [5]
  - On average $O(n)$ in time, $c = 2$
    - Exceptions; very **sparse** or very dense
    - $0.05 < \alpha < 0.95$

How Do They Do It?

- Shiftup computed until $a_{11} = 1$
- Pivoting?
  - Add a_1 to all other rows where $a_{i1}=1$
  - Shift-up all rows
  - Shift-left of all columns
  - All rows are “collected” at bottom

Algorithm 2 Parallelized Binary Gaussian Elimination

Require: Regular matrix $A \in \{0,1\}^{n \times n}$

```
for each column k = 1 : n do
    while a_11 = 0 do
        A := shiftup(n − k + 1, A)
    A := eliminate(A)
```

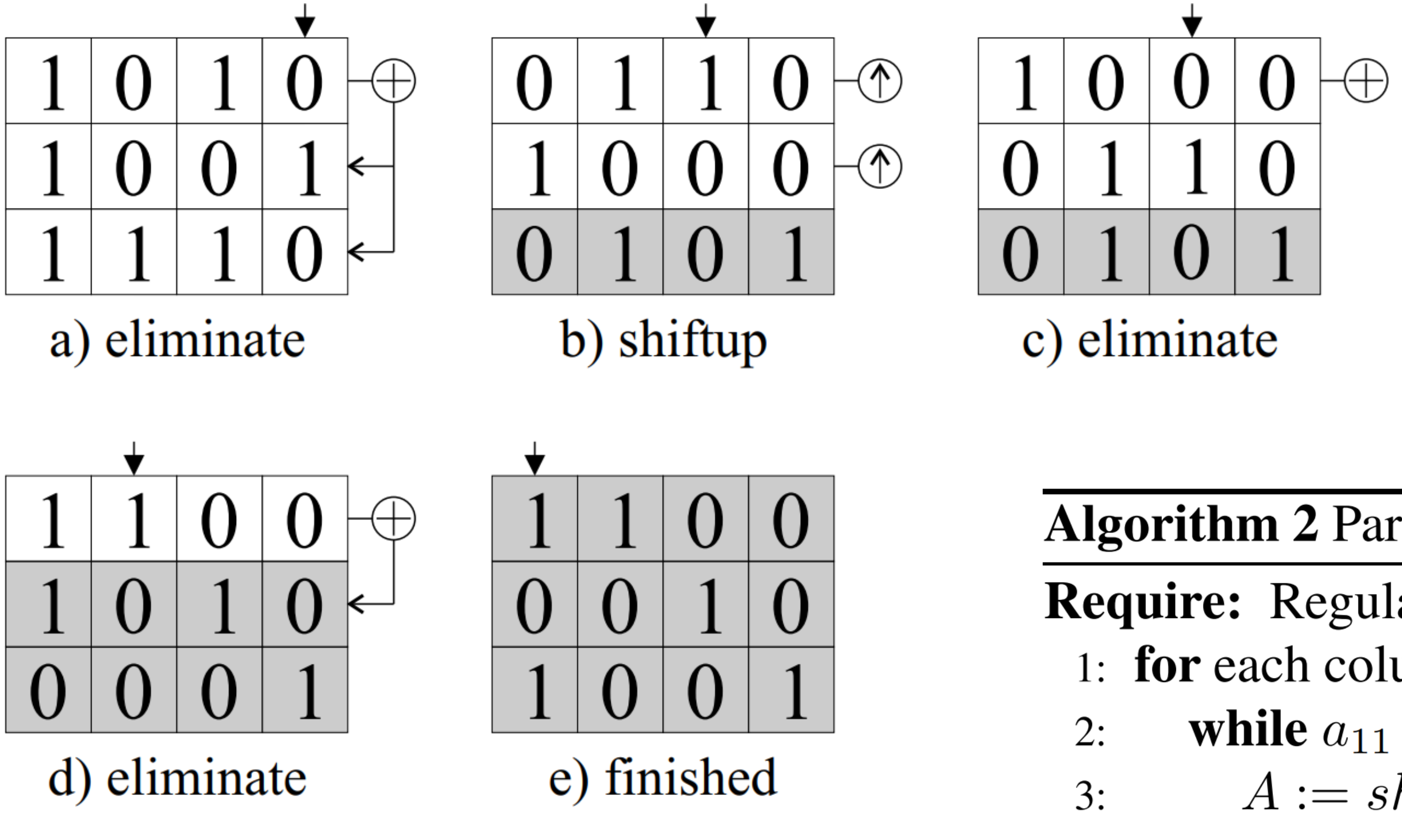
$$shiftup : \{1, \ldots, n\} \times \{0,1\}^{n \times n} \to \{0,1\}^{n \times n}$$

$$(i, (\vec{a}_1, \ldots, \vec{a}_n)^T) \mapsto (\vec{a}_2, \ldots, \vec{a}_i, \vec{a}_1, \vec{a}_{i+1}, \ldots \vec{a}_n)^T$$

$$eliminate : \{0,1\}^{n \times n} \to \{0,1\}^{n \times n}$$

$$\begin{pmatrix} 1 & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \ldots & a_{nn} \end{pmatrix} \mapsto \begin{pmatrix} a_{22} \oplus (a_{12} \wedge a_{21}) & \ldots & a_{2n} \oplus (a_{1n} \wedge a_{21}) & 0 \\ \vdots & & \vdots & \vdots \\ a_{n2} \oplus (a_{12} \wedge a_{n1}) & \ldots & a_{nn} \oplus (a_{1n} \wedge a_{n1}) & 0 \\ a_{12} & \ldots & a_{1n} & 1 \end{pmatrix}$$

How Do They Do It?



Algorithm 2 Parallelized Binary Gaussian Elimination

Require: Regular matrix $A \in \{0, 1\}^{n \times n}$

1: **for** each column $k = 1 : n$ **do**

2:     **while** $a_{11} = 0$ **do**

3:         $A := shiftup(n - k + 1, A)$

4:     $A := eliminate(A)$

Sparce Matrices

- What is a sparce matrix?
  - Wikipedia:

    Number of non-zero entries in $\mathbf{A}^{nxm} \approx$ n or m
  - Learn Data Science:

    2/3 of **A** is 0
- Sparsity
  - Number of 0s divided by nxm

Implementation

- Python, Scipy
    - coo_matrix
        - COOrdinate format
    - csc_matrix
        - Compressed Sparse Column
    - csr_matrix
        - Compressed Sparse Row
    - dia_matrix
        - DIAgonal storage
    - bsr_matrix
        - Block Sparse Row
    - lil_matrix
        - row-based LIst of Lists

COO

- Advantages of the COO format
  - Facilitates fast conversion among sparse formats
  - Very fast conversion to and from CSR/CSC formats
- Disadvantages of the COO format
  - No arithmetic operations
  - No slicing
- Intended Usage
  - COO is a fast format for constructing sparse matrices
  - Once a matrix has been constructed, convert to CSR or CSC

(taken from Scipy documentation)

CSC(R)

- Advantages of the CSC(R) format
  - Efficient arithmetic operations CSC(R)+CSC(R), CSC(R)*CSC(R)
  - Efficient column (row) slicing
  - Fast matrix vector products; CSR may be faster than CSC
- Disadvantages of the CSC(R) format
  - Slow row (column) slicing operations
  - Changes to the sparsity structure are expensive (consider LIL)

(taken from Scipy documentation)

DIA/BSR

- DIA
  - Easy to build for constant diagonal or circulant matrices
- BSR
  - Similar to CSR
  - Useful for vector-valued FE discretization
  - May provide blocksize

```
In [1]: import numpy as np
   ...: from scipy.sparse import dia_matrix
   ...: n = 10
   ...: ex = np.ones(n)
   ...: data = np.array([ex, 2 * ex, ex])
   ...: offsets = np.array([-1, 0, 1])
   ...: dia_matrix((data, offsets), shape=(n, n)).toarray()
Out[1]:
array([[2., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
       [1., 2., 1., 0., 0., 0., 0., 0., 0., 0.],
       [0., 1., 2., 1., 0., 0., 0., 0., 0., 0.],
       [0., 0., 1., 2., 1., 0., 0., 0., 0., 0.],
       [0., 0., 0., 1., 2., 1., 0., 0., 0., 0.],
       [0., 0., 0., 0., 1., 2., 1., 0., 0., 0.],
       [0., 0., 0., 0., 0., 1., 2., 1., 0., 0.],
       [0., 0., 0., 0., 0., 0., 1., 2., 1., 0.],
       [0., 0., 0., 0., 0., 0., 0., 1., 2., 1.],
       [0., 0., 0., 0., 0., 0., 0., 0., 1., 2.]])
```

LIL

- Advantages of the LIL format
  - Supports flexible slicing
  - Changes to the matrix sparsity structure are efficient
- Disadvantages of the LIL format
  - Arithmetic operations are slow (consider CSR or CSC)
  - Slow column slicing (consider CSC)
  - Slow matrix vector products (consider CSR or CSC)

(taken from Scipy documentation)

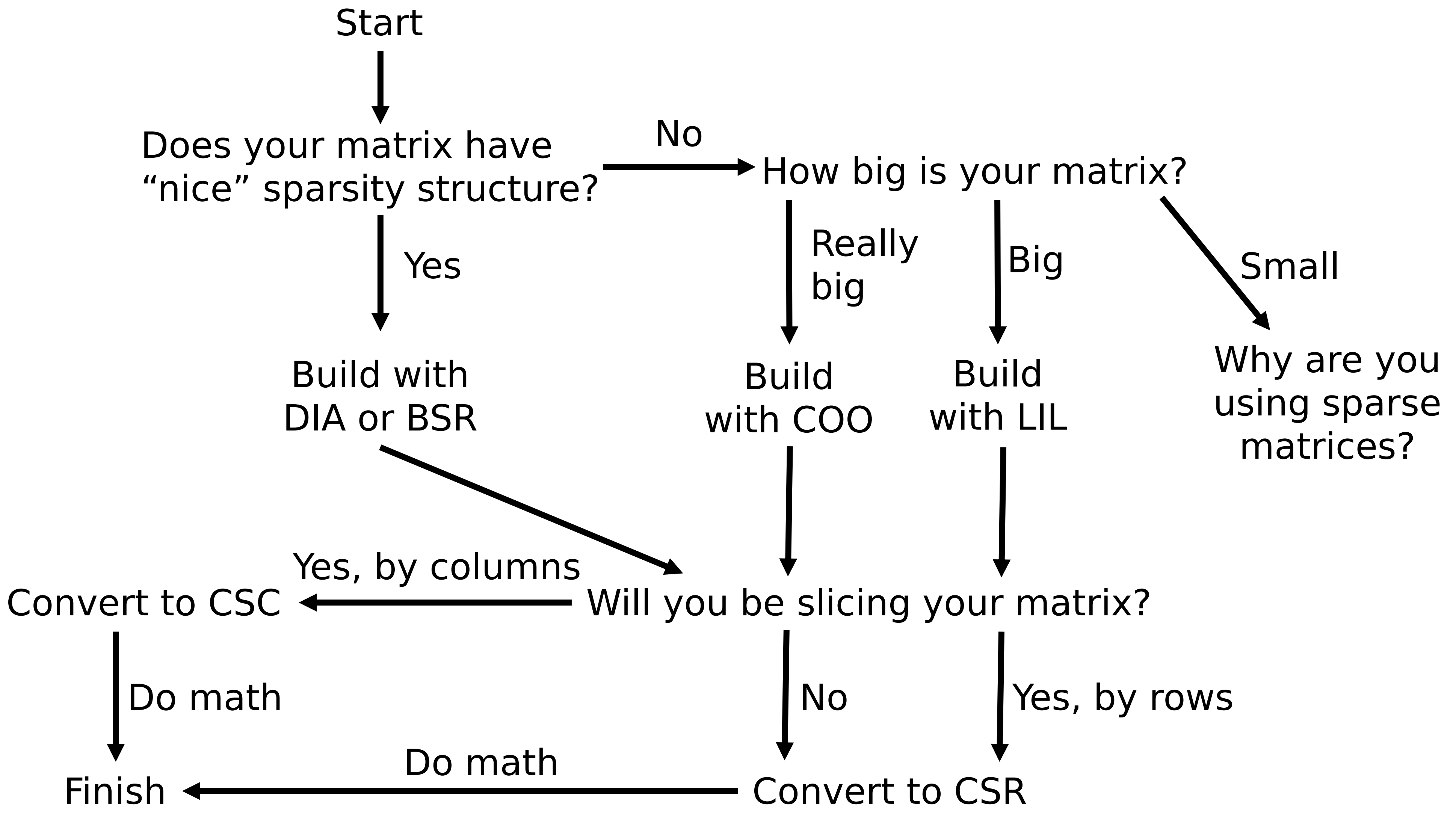
LIL

- Intended Usage
  - LIL is a **convenient** format for constructing sparse matrices
  - Once a matrix has been constructed, convert to CSR or CSC format
  - Consider using the COO format when constructing large matrices

```
C:\Users\15303\Anaconda3\lib\site-packages\scipy\sparse\_index.py:103: SparseEfficiencyWarning: Changing the sparsity
structure of a csr_matrix is expensive. lil_matrix is more efficient.
```

(taken from Scipy documentation)

Flowchart


Start
Does your matrix have
“nice” sparsity structure?
No
How big is your matrix?
Yes
Really
big
Big
Small
Build with
DIA or BSR
Build
with COO
Build
with LIL
Why are you
using sparse
matrices?
Yes, by columns
Convert to CSC
Will you be slicing your matrix?
Do math
No
Yes, by rows
Do math
Finish
Convert to CSR

Doing Math (Optimally)

- Tridiagonal Solver
  - Thomas Algorithm
  - O(n)
- $a_i \rightarrow 0$, $b_i \rightarrow 1$
- Replace c_i with:

$$c'_i = \begin{cases} \dfrac{c_i}{b_i}, & i = 1, \\ \dfrac{c_i}{b_i - a_i c'_{i-1}}, & i = 2, 3, \dots, n-1 \end{cases}$$

- Replace d_i with:

$$d'_i = \begin{cases} \dfrac{d_i}{b_i}, & i = 1, \\ \dfrac{d_i - a_i d'_{i-1}}{b_i - a_i c'_{i-1}}, & i = 2, 3, \dots, n \end{cases}$$

- Backsubtitute to solve:

$$x_n = d'_n,$$
$$x_i = d'_i - c'_i x_{i+1}, \quad i = n-1, n-2, \dots, 1$$

$$\begin{bmatrix} b_1 & c_1 & & & 0 \\ a_2 & b_2 & c_2 & & \\ & a_3 & b_3 & \ddots & \\ & & \ddots & \ddots & c_{n-1} \\ 0 & & & a_n & b_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_n \end{bmatrix}$$

History

- Llewellyn H. Thomas (1903-1992)
  - Physicist and applied mathematician
  - Big in quantum mechanics
  - Thomas algorithm seems to come out of [6] (1942)
  - “LHT had a huge influence on the physics, mathematics, and machine design principles and hardware of the Watson Lab.”[7]

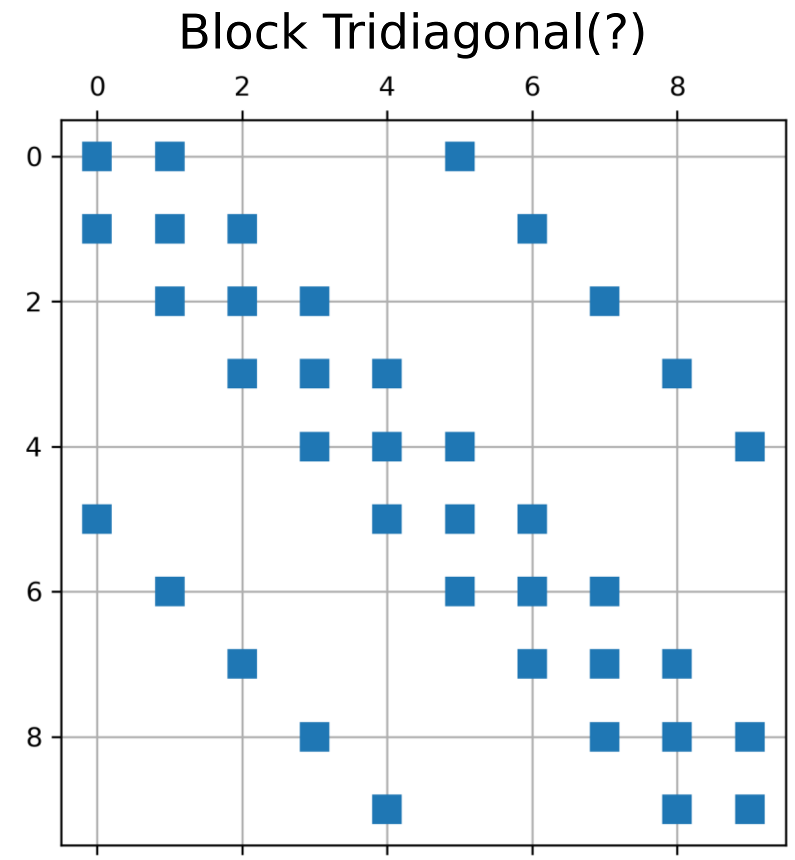
Thomas and **Wallace J. Eckart**, Director of Watson Lab, at work

Numerical Experiments

- Device specifications
  - 2016 Dell Inspiron 15
  - Intel Core i5-7200U
  - CPU at 2.50GHz
  - 8.00 GB Ram (memory exceeded with n= 5×10^6 square matrix)*
  - 64-bit operating system
- Software
  - Python 3.9.13
  - Scipy 1.9.1

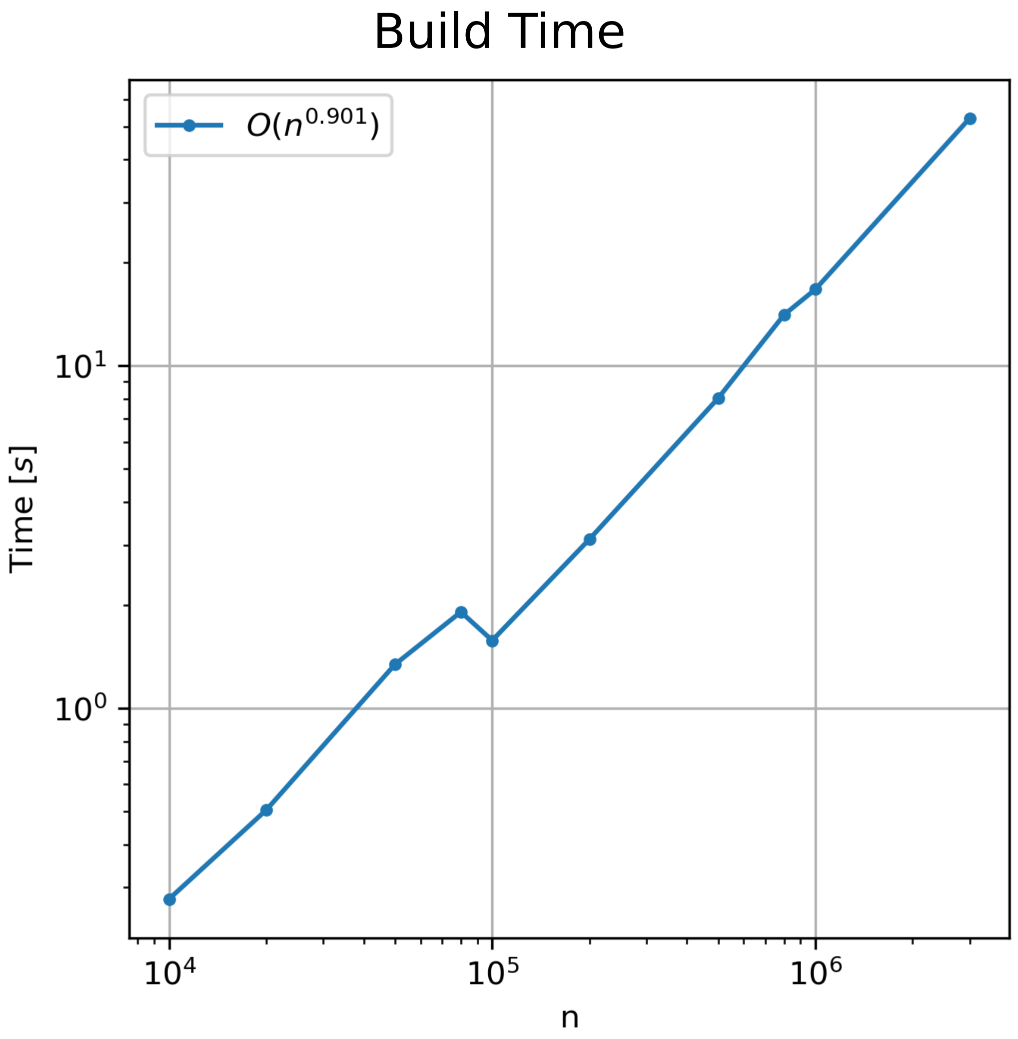
Numerical Experiments

- Recall CSR may be faster than CSC
- Some questions to ask:
  - What are the build times?
  - What are the solve times?
  - How expensive is converting sparce matrix types?
  - How does the sparsity structure change run efficiency?

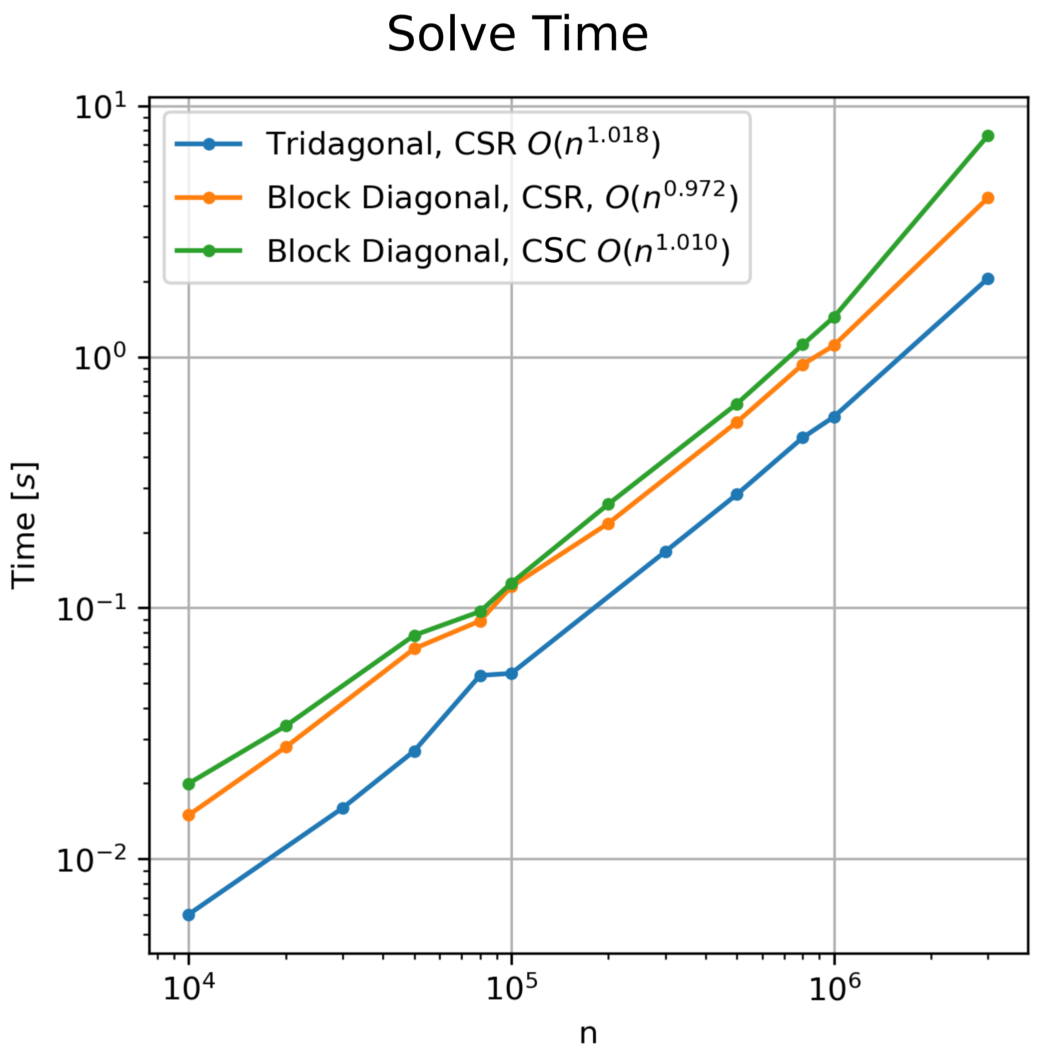


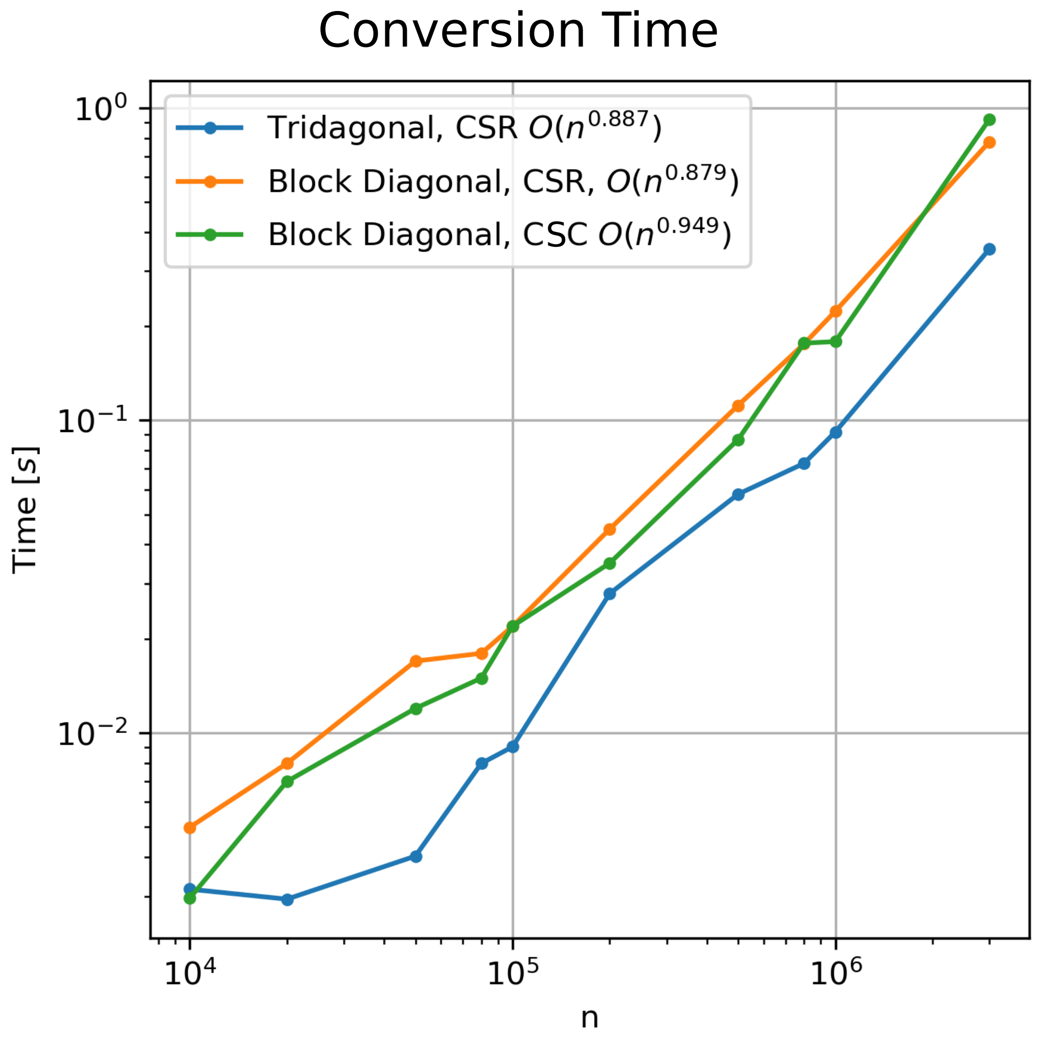
Numerical Experiments

- Solving **Ax=b**
  - **A** tridiagonal
  - **A** tridiagonal with additional entries at j=±5
- Built with LIL
  - Building **A** is the most expensive process

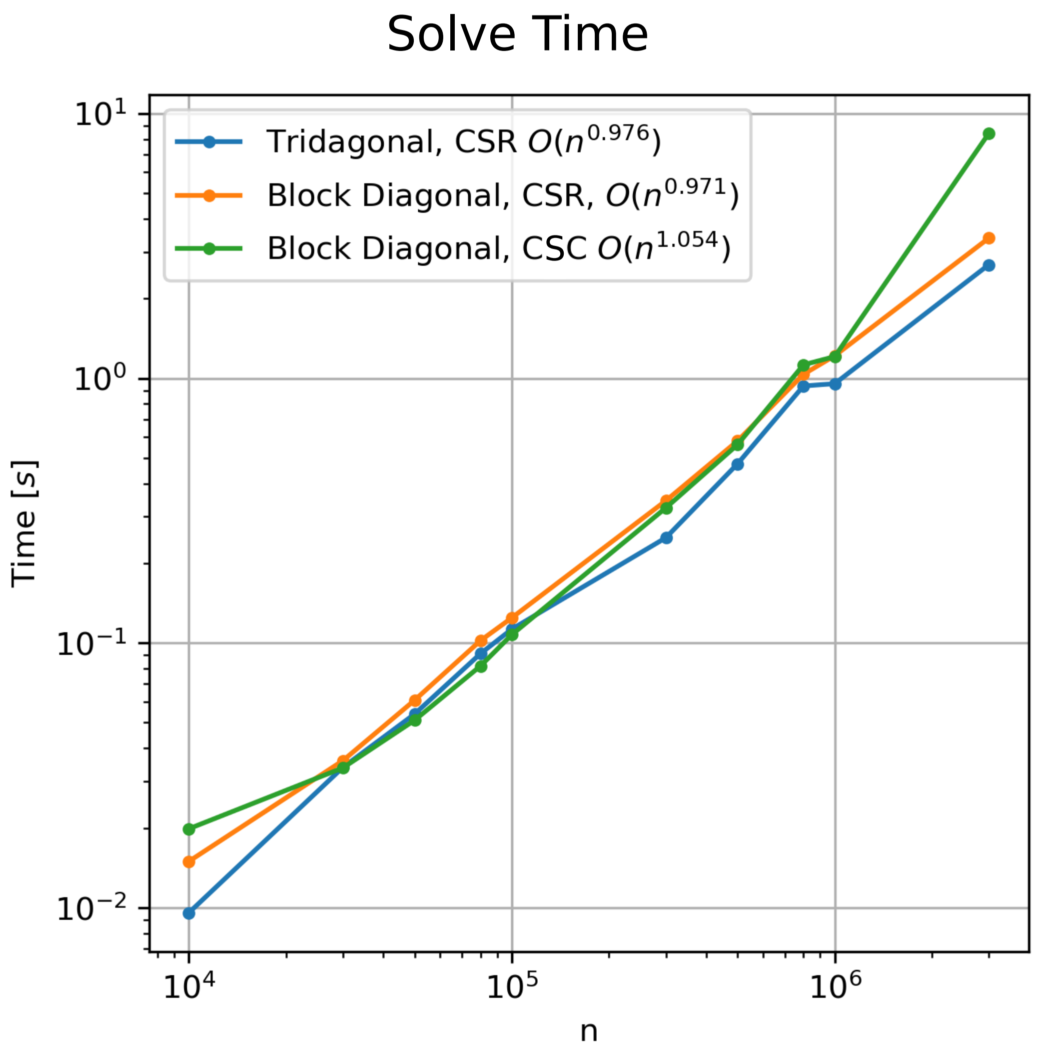


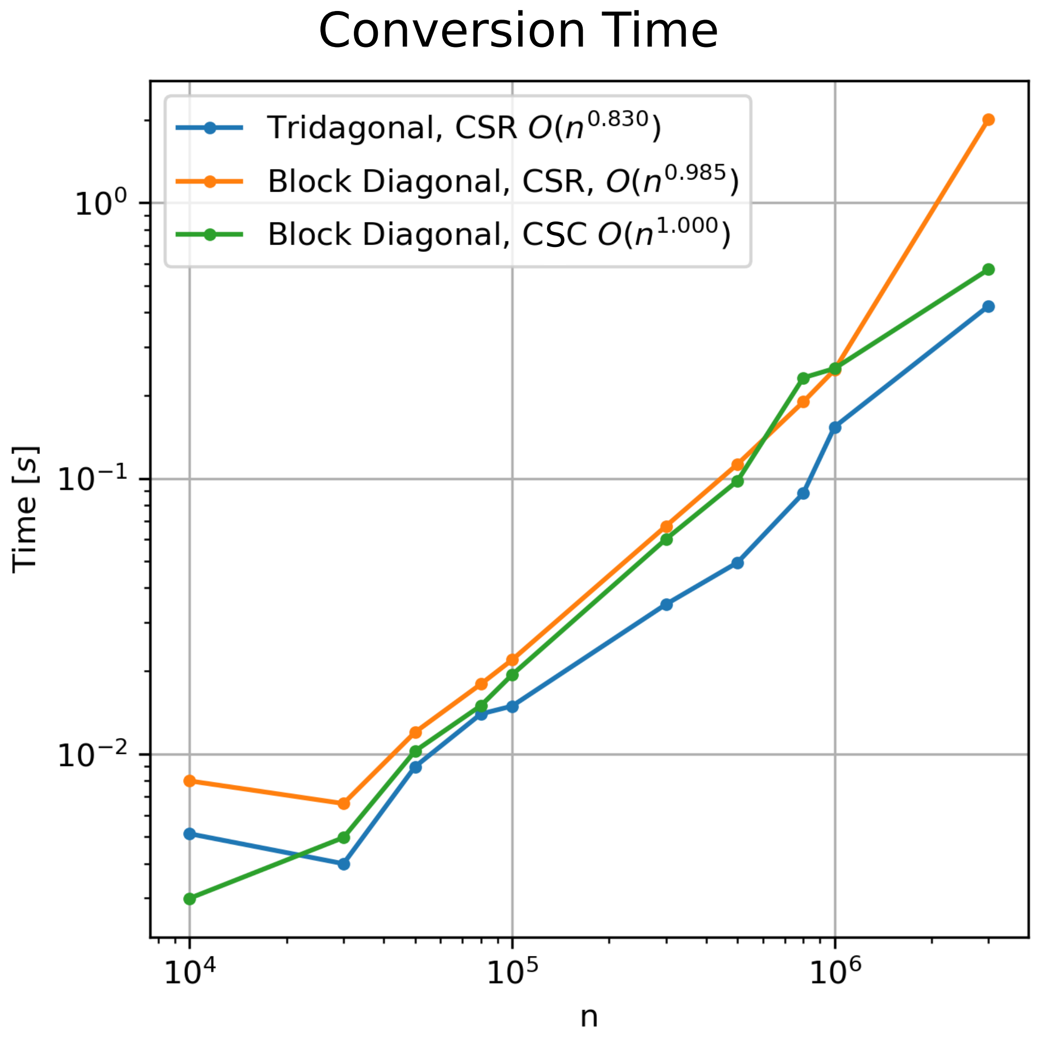
Numerical Experiments





Numerical Experiments

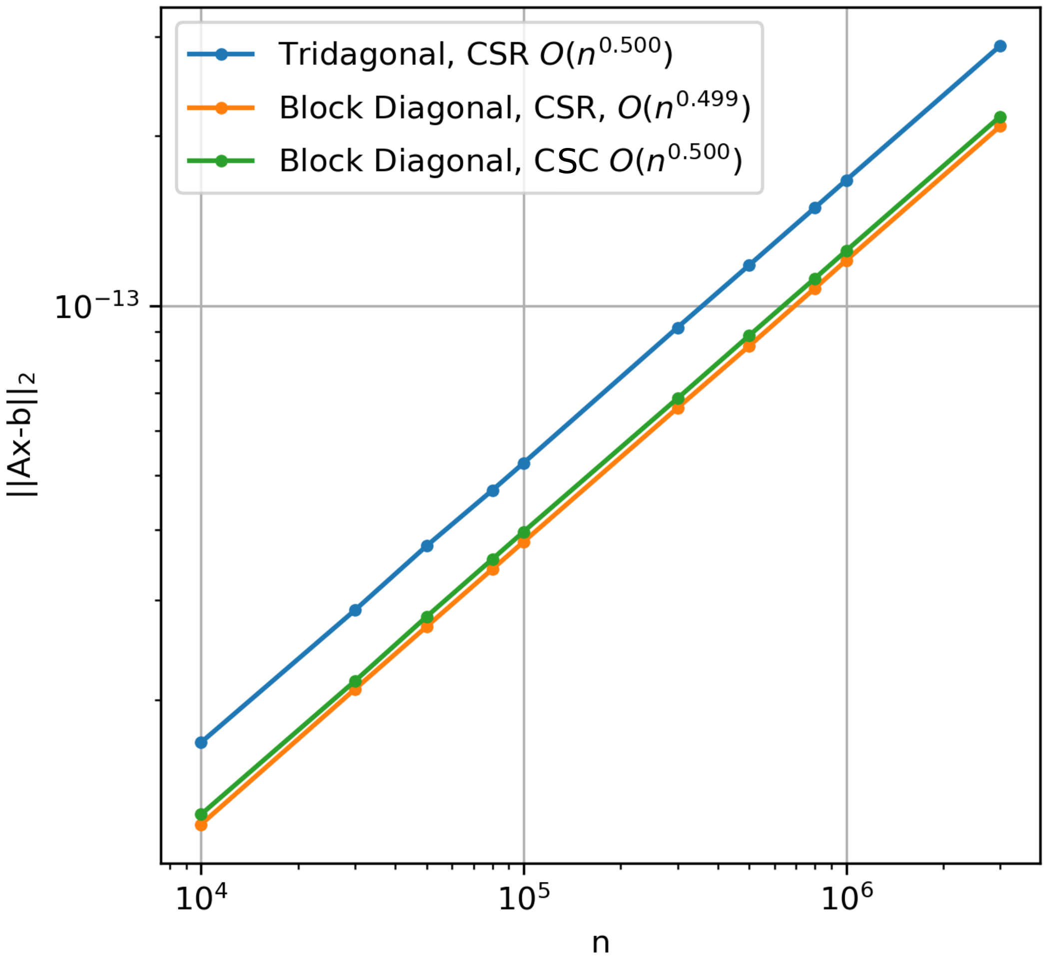




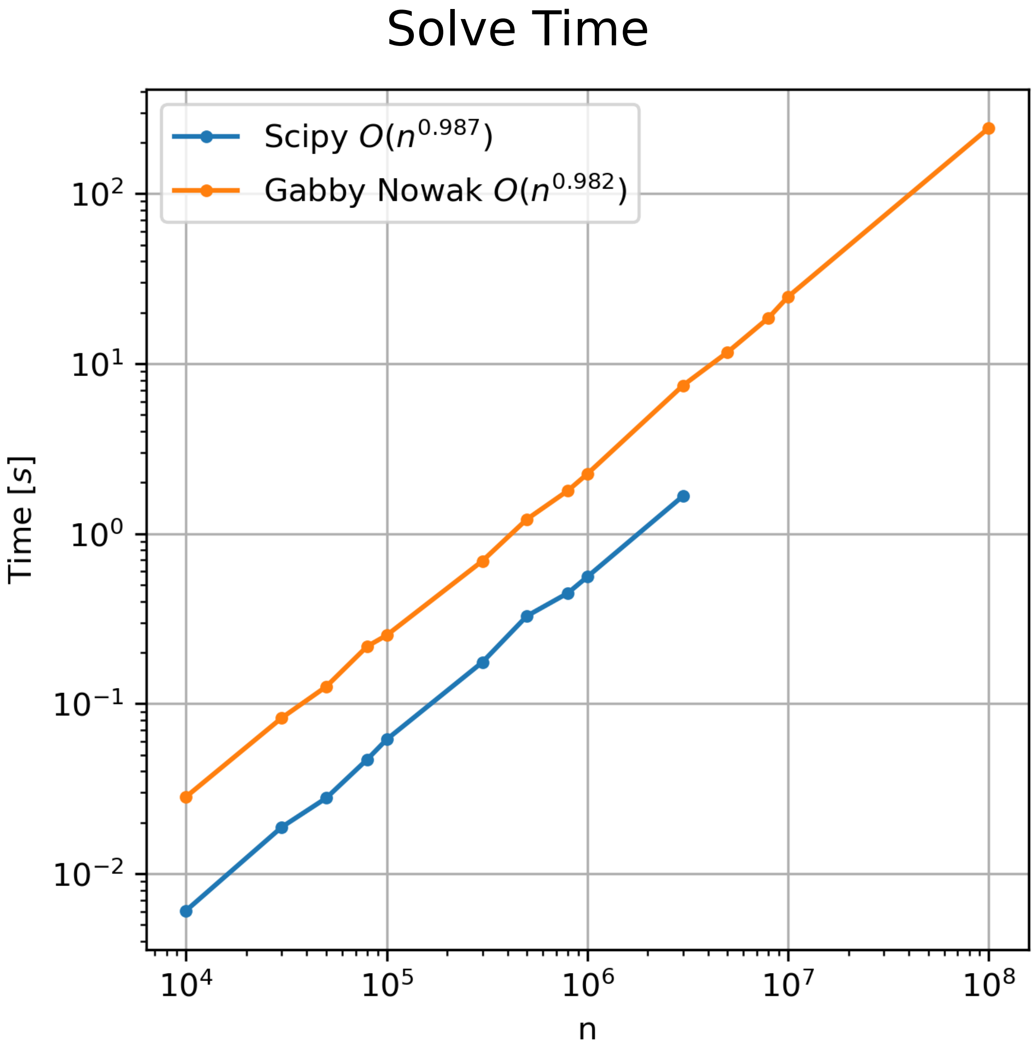
Numerical Experiments

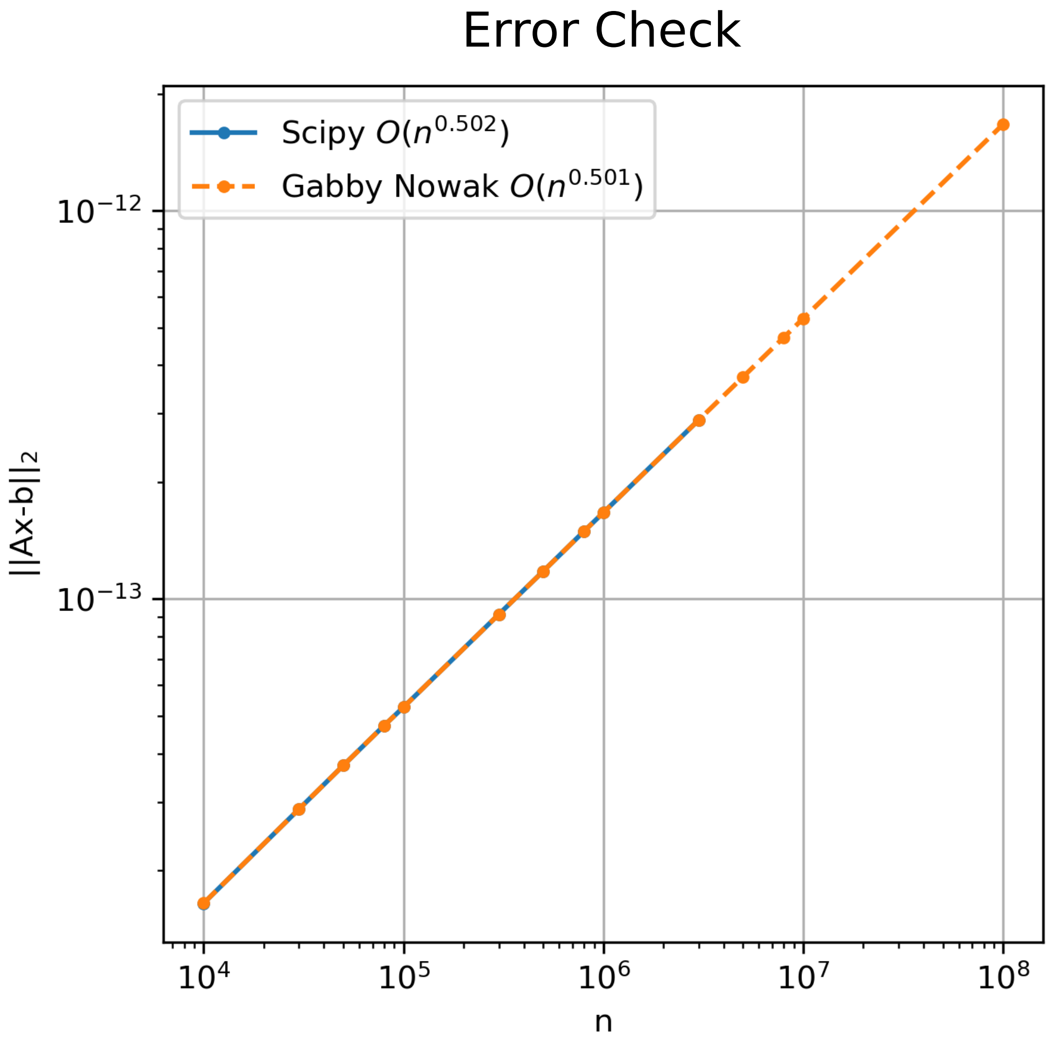
- CSR vs CSC doesn't make that much of a difference
- Scipy might be inverting the matrix in order to solve **Ax=b**
  - Memory runs outs at solve step
  - Hand made sparse solver?

Check that I am doing the right thing



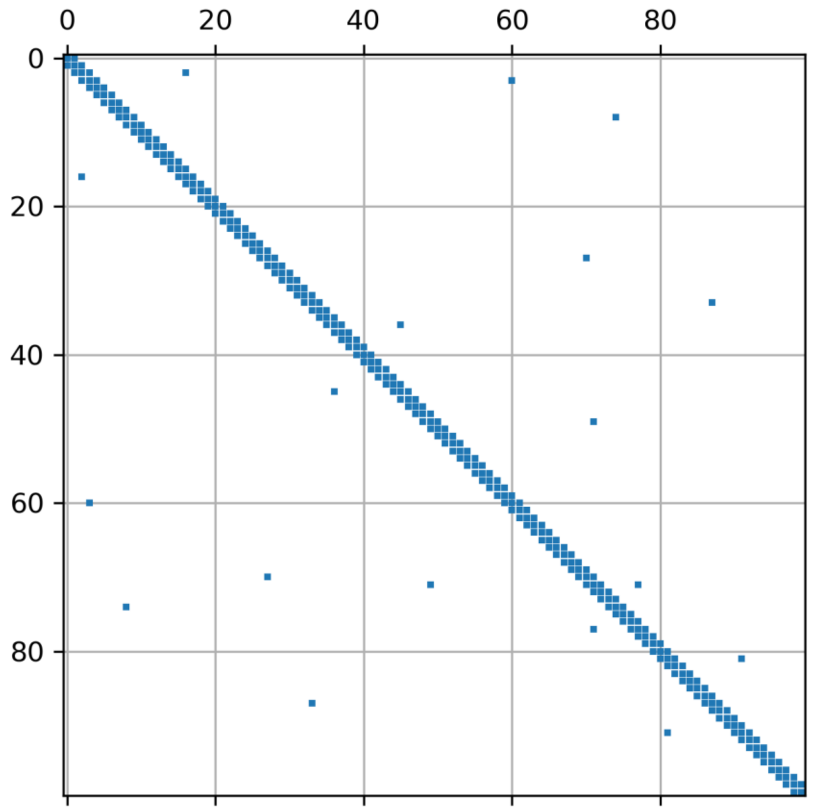
Numerical Experiments



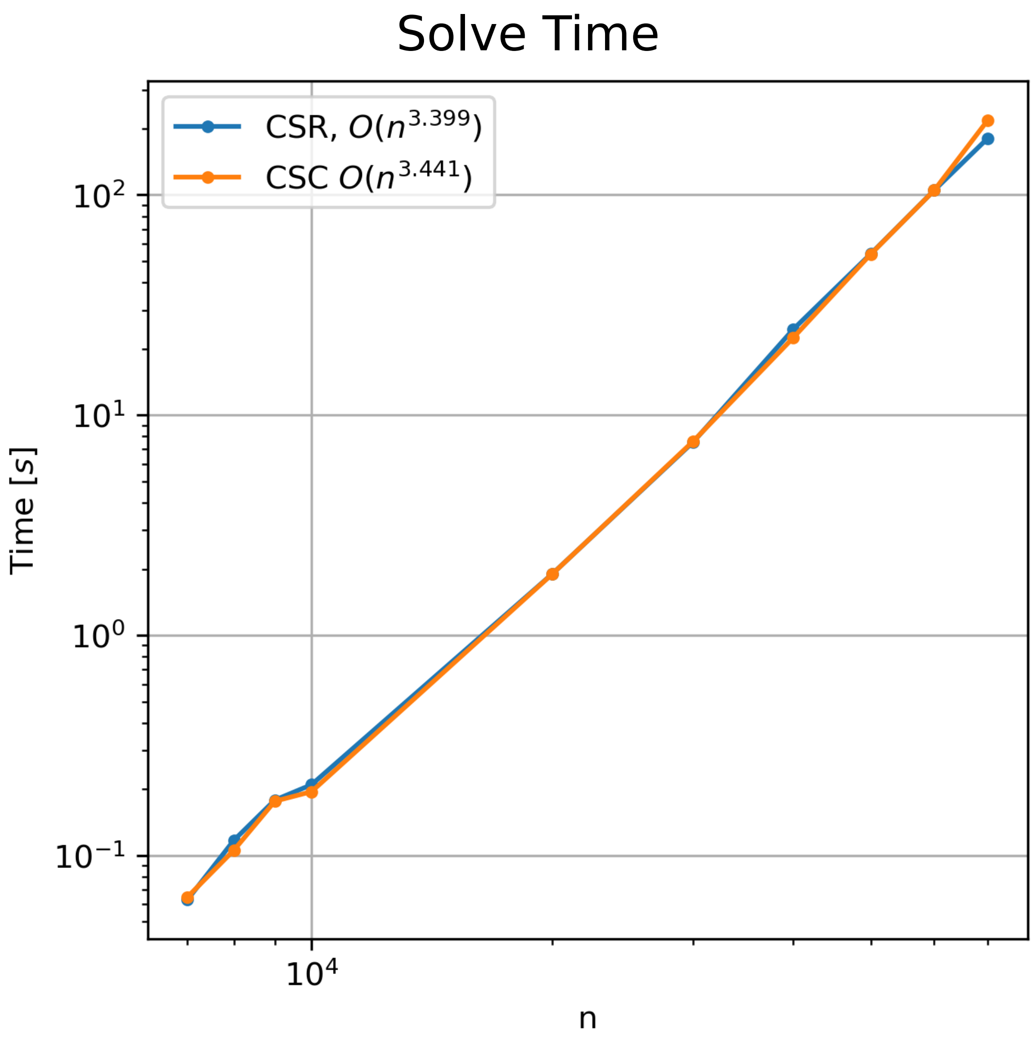


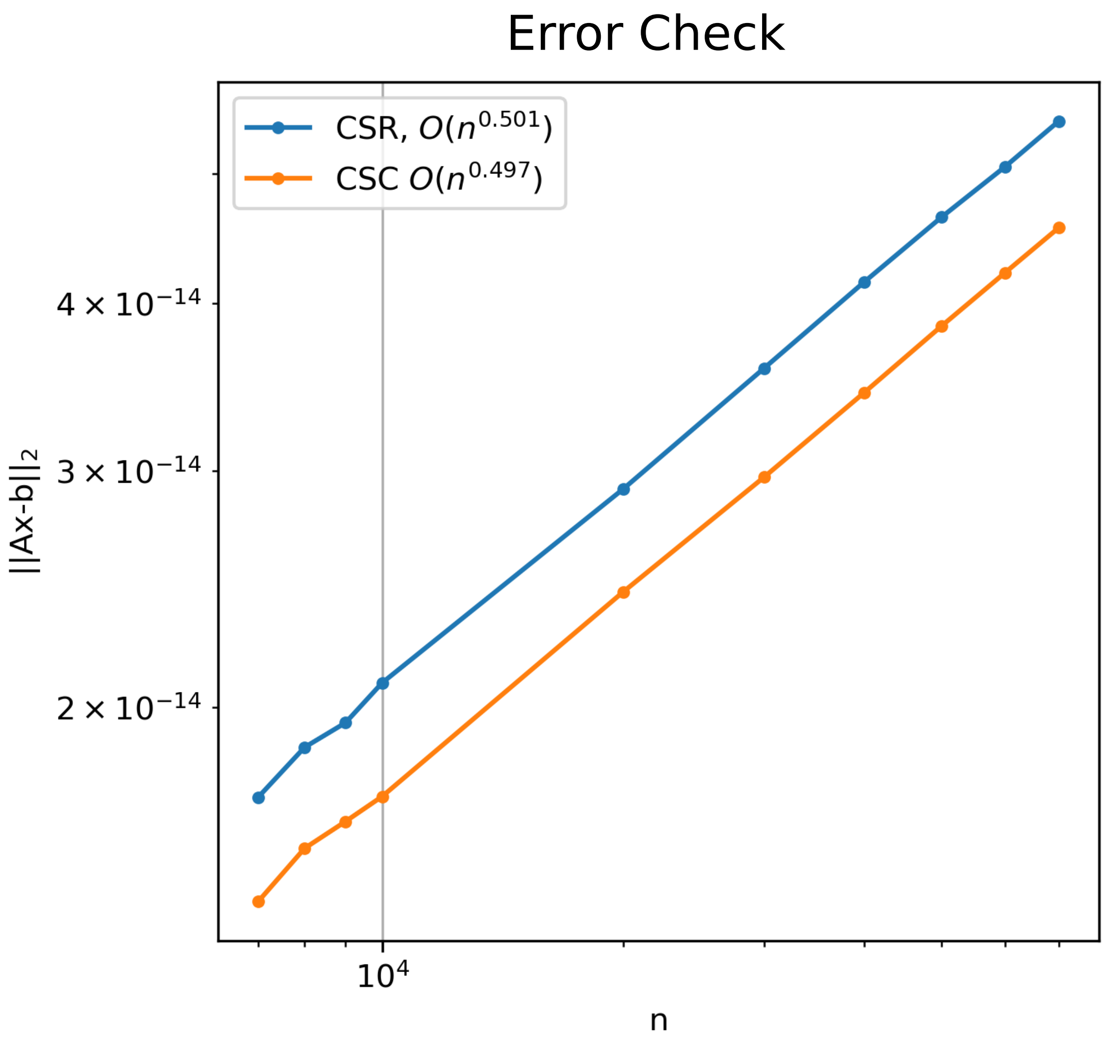
Numerical Experiments

- Thomas algorithm does not take as much memory
- Non-banded matrices?
  - Finite element motivated
  - Tri-diagonal + 1% filled
  - Still sparse
  - Symmetric positive (hopefully definite)
  - How well does Scipy handle this?



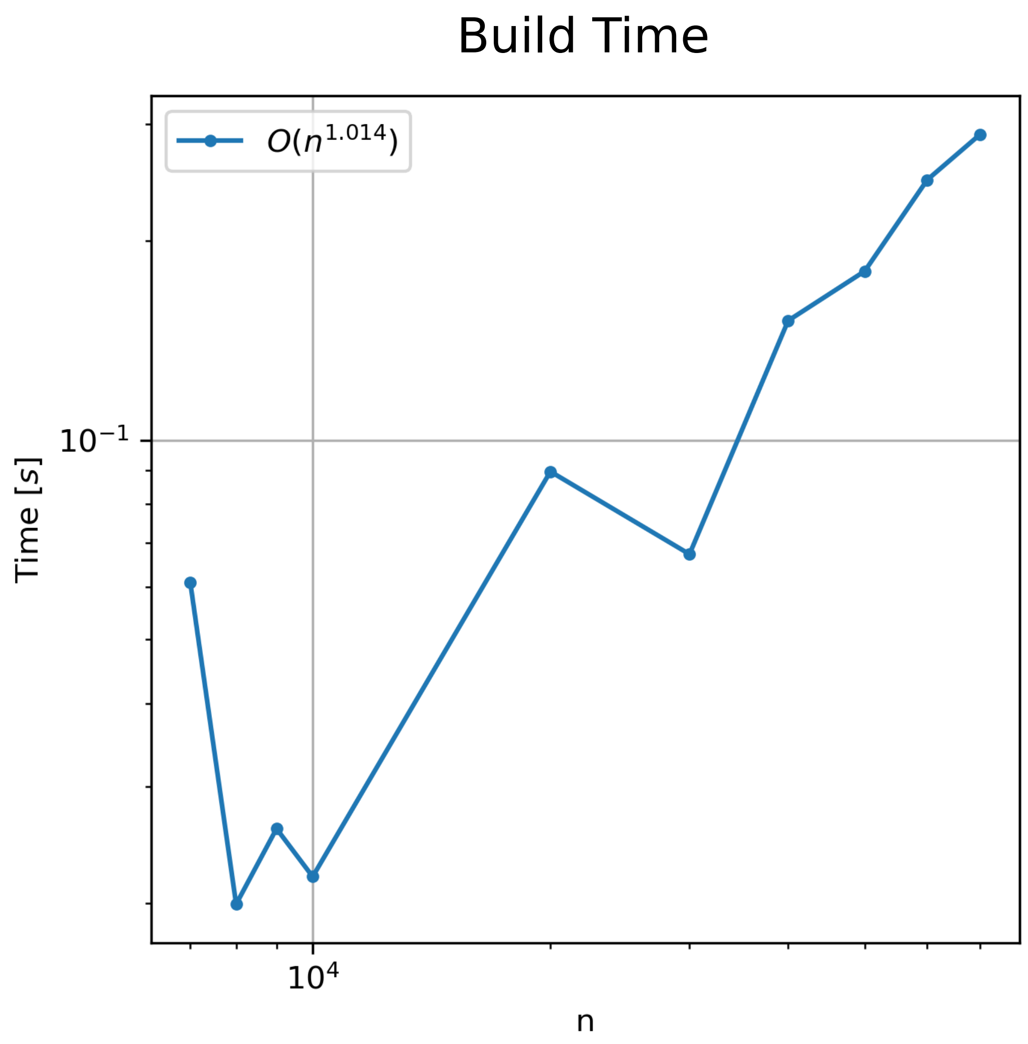
Numerical Experiments



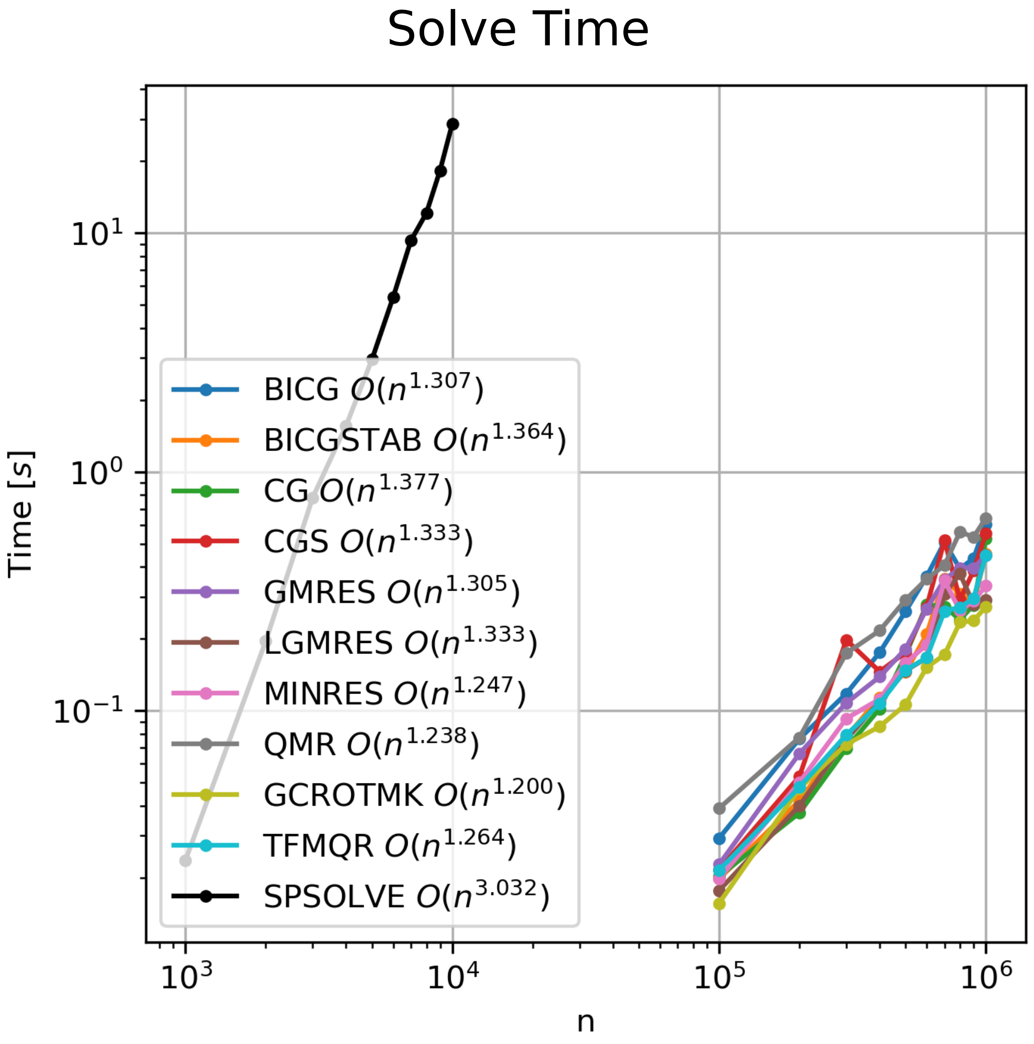


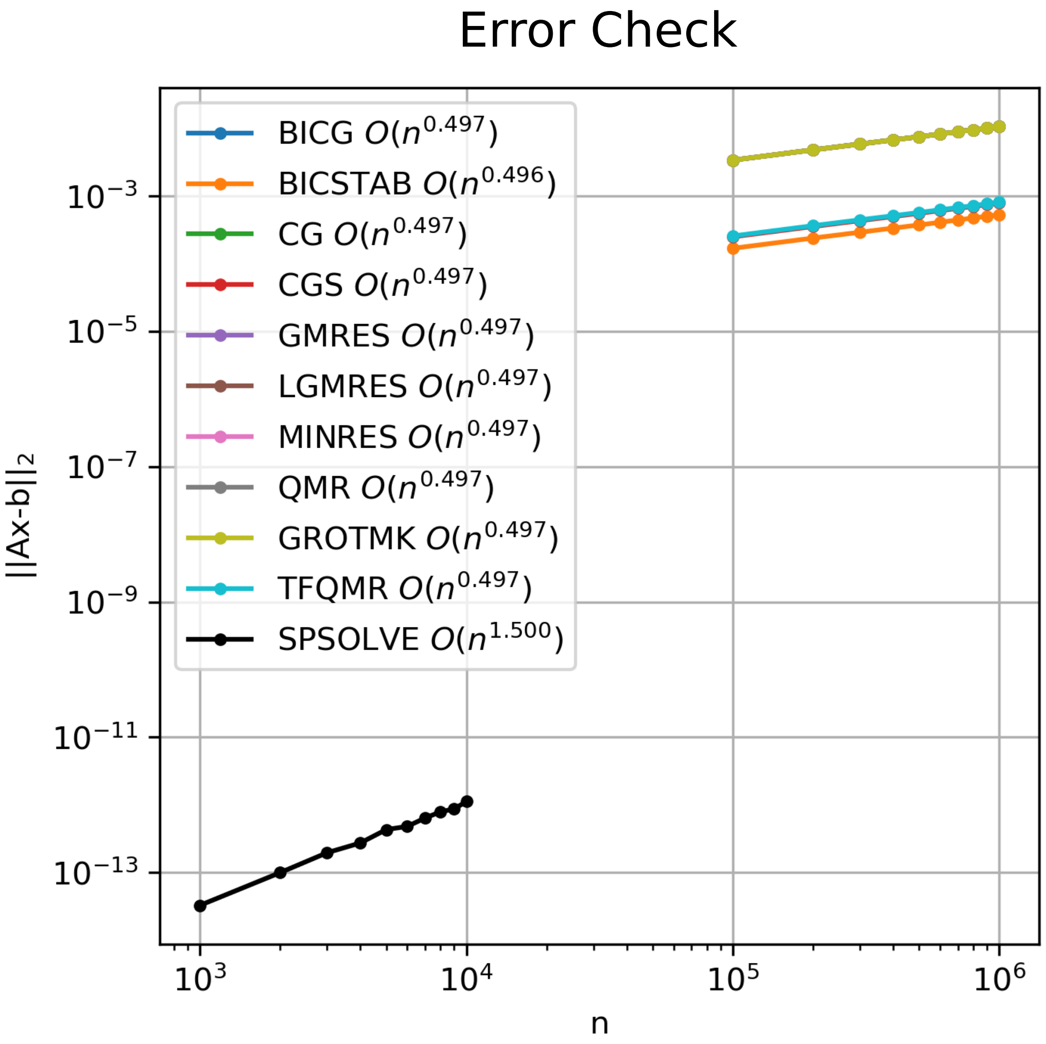
Numerical Experiments

- Awful!
  - SPSolve does not have any tricks up its sleeves
  - Perhaps it needed to be at least 5% sparse?
  - Next part is at most 10% sparse + tridiagonal
- There is another way!
  - Iterative methods
  - Scipy has 10 different iterative $\mathbf{Ax} = \mathbf{b}$ solvers
  - How do they compare?

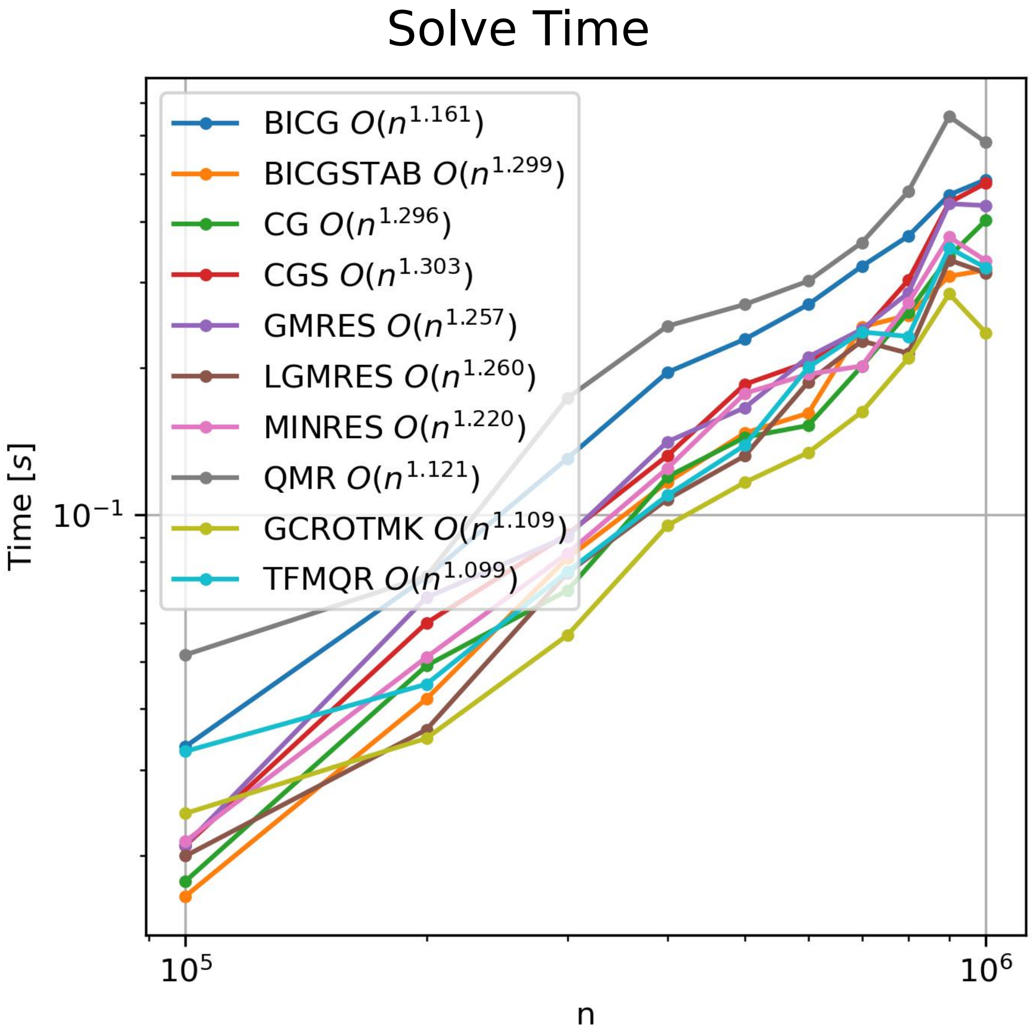


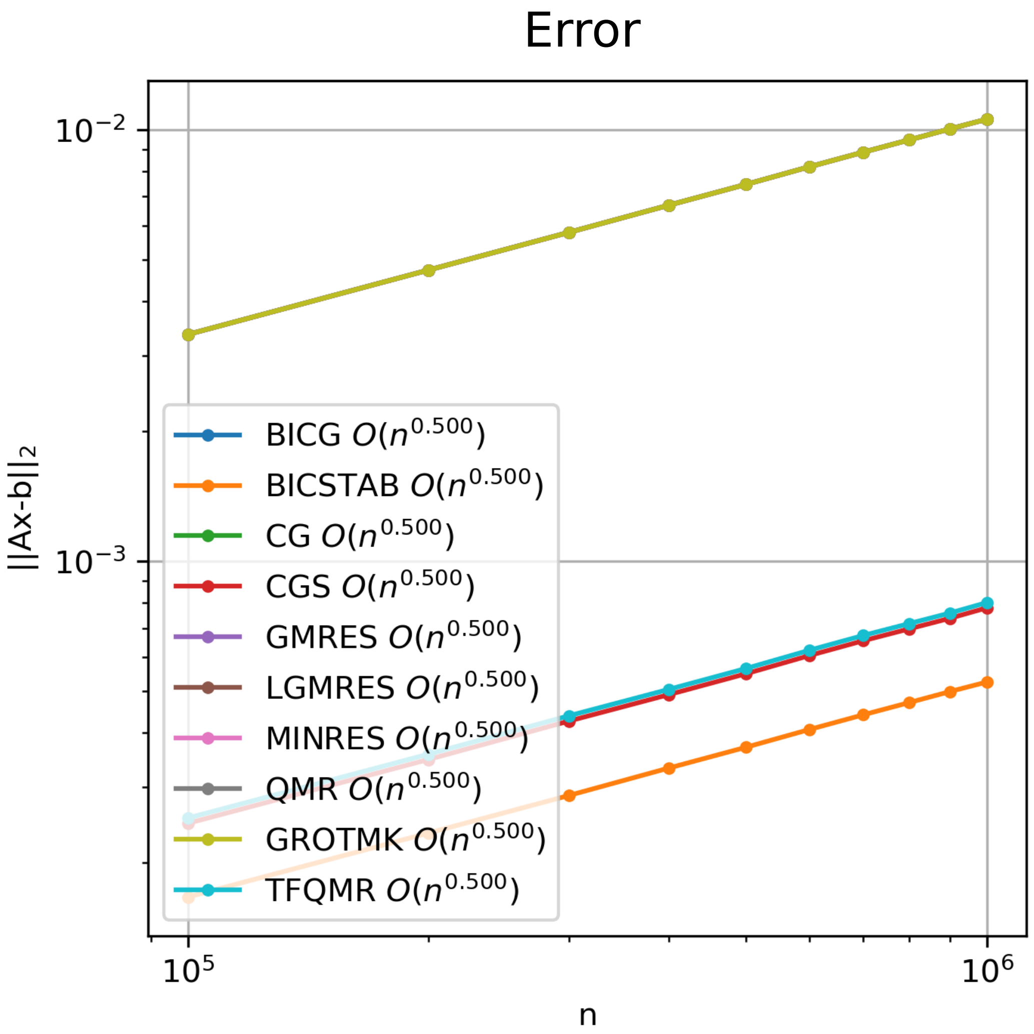
Numerical Experiments





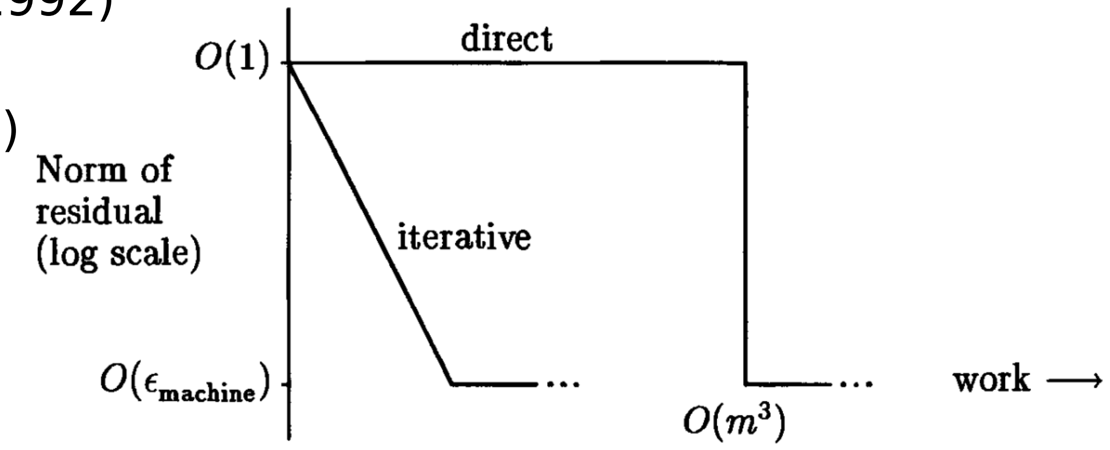
Numerical Experiments





Discovering Where I Should Have Started

- Numerical Linear Algebra (L. N. Trefethen and D. Bau, 1997)
  - Part VI is iterative methods
  - Mentions most of scipy's implemented methods
    - BIConjugate Gradient
    - BIConjugate Gradient STABilized (1992)
    - Conjugate Gradient (1952)
    - Conjugate Gradient Squared (1989)
    - MINimal RESidues
    - Generalized Minimal RESiduals
    - Quasi-Minimal Residuals (1991)
    - Transpose-Free QMR (1993)
  - New methods
    - Loose GMRES [8] (2004)
    - Generalized Conjugate Residuals [9] (1983) with inner Orthogonalization [10] (1996) Truncated [11] (1999) for an (M,K) matrix



Arnoldi Iteration

- Simply the modified Gram-Schmidt iteration

Algorithm 33.1. Arnoldi Iteration

$b =$ arbitrary, $q_1 = b/\|b\|$
for $n = 1, 2, 3, \ldots$
    $v = Aq_n$
    for $j = 1$ **to** n
        $h_{jn} = q_j^* v$
        $v = v - h_{jn} q_j$
    $h_{n+1,n} = \|v\|$ [see Exercise 33.2 concerning $h_{n+1,n} = 0$]
    $q_{n+1} = v/h_{n+1,n}$

- One interpretation is the Krylov subspaces/matrices

$$K_n = \left[\begin{array}{c|c|c|c} \\ b & Ab & \cdots & A^{n-1}b \\ \\ \end{array}\right]$$

- Which has a reduced QR factorization

GRMES

Algorithm 35.1. GMRES

$q_1 = b/\|b\|$

for $n = 1, 2, 3, \ldots$

  ⟨ *step n of Arnoldi iteration, Algorithm* 33.1 ⟩

  Find y to minimize $\|\tilde{H}_n y - \|b\| e_1\|$ $(= \|r_n\|)$

  $x_n = Q_n y$.

- Minimizes r_n = b – $A^{(mxm)} x_n$ over all vectors
  - Least squares problem with Hessenburg structure
  - QR factorization for $O(m^2)$
  - With a Givens rotation it is apparently O(m)…?

CG

Algorithm 38.1. Conjugate Gradient (CG) Iteration

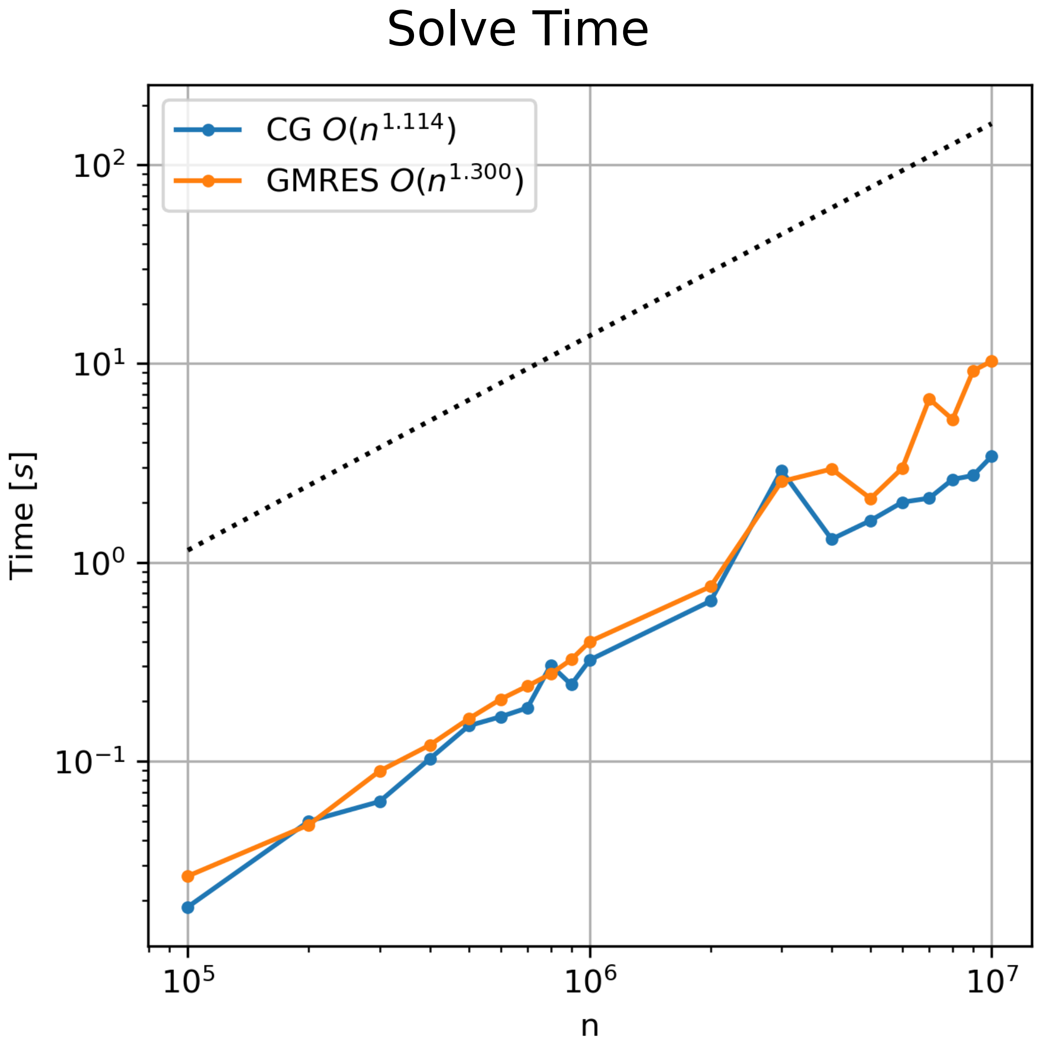
$x_0 = 0, \;\; r_0 = b, \;\; p_0 = r_0$

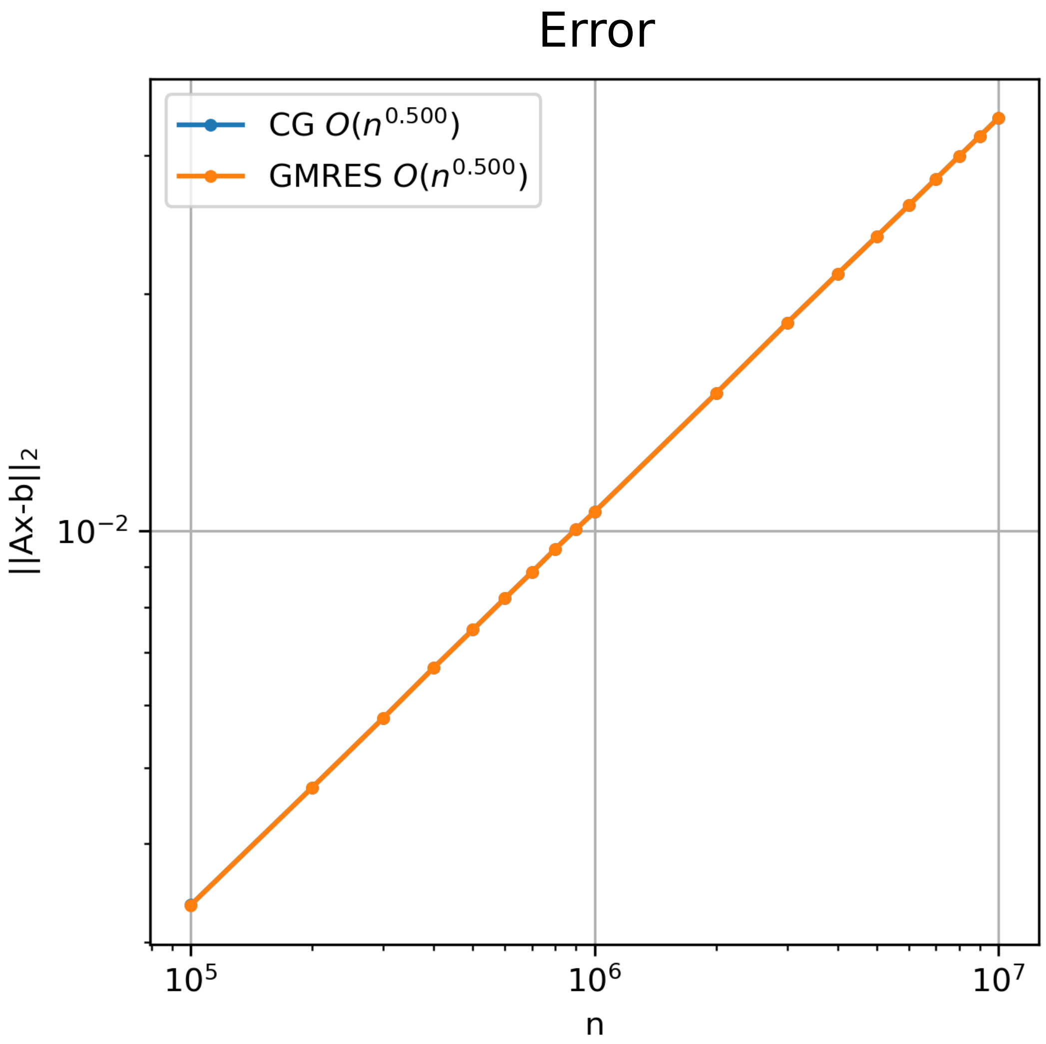
for $n = 1, 2, 3, \ldots$

$\alpha_n = (r_{n-1}^T r_{n-1})/(p_{n-1}^T A p_{n-1})$	step length
$x_n = x_{n-1} + \alpha_n p_{n-1}$	approximate solution
$r_n = r_{n-1} - \alpha_n A p_{n-1}$	residual
$\beta_n = (r_n^T r_n)/(r_{n-1}^T r_{n-1})$	improvement this step
$p_n = r_n + \beta_n p_{n-1}$	search direction

- If **A** is dense, $O(m^2)$/step
- If **A** is sparse, potentially $O(m)$/step

Numerical Experiments

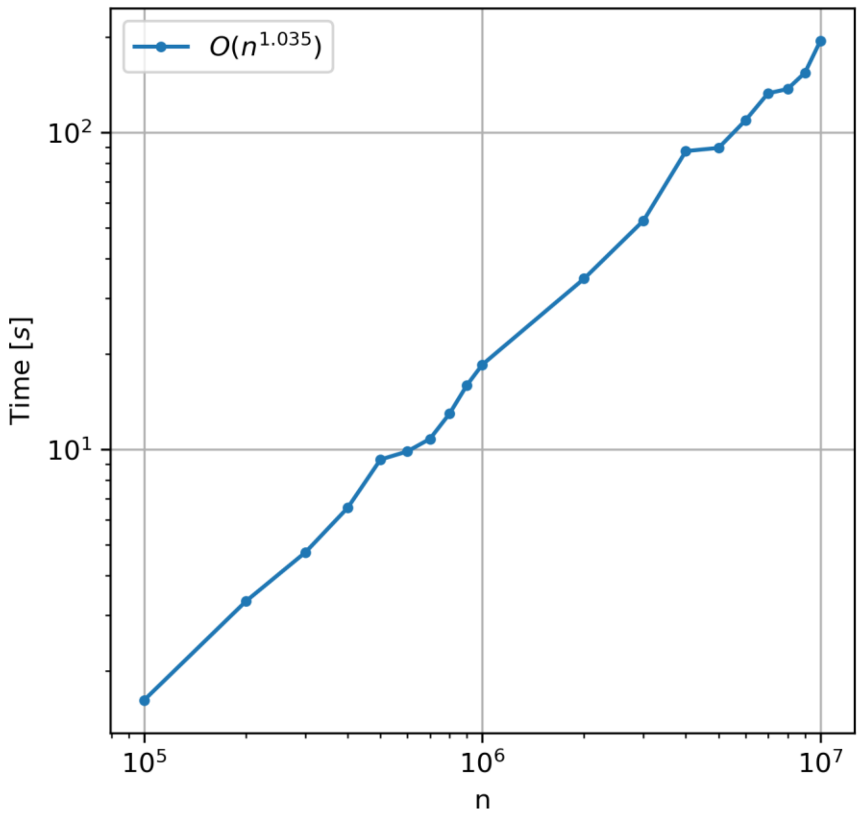




Numerical Experiments

- Sacrifice accuracy for time
- Seems to be O(m log(m))
- Things I didn't look at
  - Preconditioning
  - Tolerance conditions

Build Time



Summary

- Sparse storage is useful
  - Building
  - Storing
  - Directly solve “simple” sparse systems (via Thomas or inversing)
  - Iteratively solve more complex systems
- If your problem is big, you should either parallelize, move to hardware, or both

Other Implementations

Software [edit]

Many software libraries support sparse matrices, and provide solvers for sparse matrix equations. The following are open-source:

- SuiteSparse, a suite of sparse matrix algorithms, geared toward the direct solution of sparse linear systems.
- PETSc, a large C library, containing many different matrix solvers for a variety of matrix storage formats.
- Trilinos, a large C++ library, with sub-libraries dedicated to the storage of dense and sparse matrices and solution of corresponding linear systems.
- Eigen3 is a C++ library that contains several sparse matrix solvers. However, none of them are parallelized.
- MUMPS (**MU**ltifrontal **M**assively **P**arallel sparse direct **S**olver), written in Fortran90, is a frontal solver.
- deal.II, a finite element library that also has a sub-library for sparse linear systems and their solution.
- DUNE, another finite element library that also has a sub-library for sparse linear systems and their solution.
- PaStix.
- SuperLU.
- Armadillo provides a user-friendly C++ wrapper for BLAS and LAPACK.
- SciPy provides support for several sparse matrix formats, linear algebra, and solvers.
- SPArse Matrix (spam) R and Python package for sparse matrices.
- Wolfram Language Tools for handling sparse arrays
- ALGLIB is a C++ and C# library with sparse linear algebra support
- ARPACK Fortran 77 library for sparse matrix diagonalization and manipulation, using the Arnoldi algorithm
- SPARSE Reference (old) NIST package for (real or complex) sparse matrix diagonalization
- SLEPc Library for solution of large scale linear systems and sparse matrices
- Sympiler, a domain-specific code generator and library for solving linear systems and quadratic programming problems.
- scikit-learn, a Python library for machine learning, provides support for sparse matrices and solvers.
- sprs implements sparse matrix data structures and linear algebra algorithms in pure Rust.
- Basic Matrix Library (bml) supports several sparse matrix formats and linear algebra algorithms with bindings for C, C++, and Fortran.

My Unanswered Questions

- Why does error go as $O(n^{1/2})$?
    - CG has a convergence to a tolerance in $O(\kappa^{1/2})$
    - κ is the 2-norm condition number
    - Is this a related property for all methods?
- Stability concerns?
    - All my matrices are (with high likelihood) positive symmetric definite
    - Many of the methods do not work (it seems) if not
- Looking for the right tool for the job, not just using all the tools
- Doing Other Math (Optimally)
    - Inverses [12]
    - LU decomposition
    - QR factorization
    - Matrix-vector multiplication

Questions?

Codes, Unoptimally

```python
# -*- coding: utf-8 -*-
"""
Created on Sun Apr  9 14:51:18 2023

@author: Austin Smith
"""
import numpy as np
import time
from scipy.sparse  import dia_matrix
import scipy
import matplotlib.pyplot as plt

ns = [10**4,3*10**4,5*10**4,8*10**4,10**5,3*10**5,5*10**5,8*10**5,10**6,3*10**6]
norm0=[]
build_time=[]
conversion_time=[]
solve_time_tri=[]
for n in ns:
    start_time=time.time()

    ex = np.ones(n)
    data = np.array([ex, 100.0 * ex, ex])
    offsets = np.array([-1, 0, 1])
    A=dia_matrix((data, offsets), shape=(n, n))

    end_time=time.time()-start_time
    build_time.append(end_time)

    b=np.random.rand(n)+1.0

    start_time=time.time()
    A=A.tocsr()
    end_time=time.time()-start_time
    conversion_time.append(end_time)

    start_time=time.time()
    x=scipy.sparse.linalg.spsolve(A,b)
    end_time=time.time()-start_time
    solve_time_tri.append(end_time)

    norm0.append(np.linalg.norm(A.dot(x)-b))


#-----------------------------------------------------------------------------
aa=1*10

bb=2*10
cc=5*20
dd=2*10

ee=1*10
#-----------------------------------------------------------------------------

#-----------------------------------------------------------------------------

ns = [10**4,3*10**4,5*10**4,8*10**4,10**5,3*10**5,5*10**5,8*10**5,10**6,3*10**6]

build_time=[]
conversion_time1=[]
conversion_time2=[]
solve_time_off_pent1=[]
solve_time_off_pent2=[]
norm1=[]
norm2=[]
for n in ns:
    print(n)
    start_time=time.time()

    ex = np.ones(n)
    A=scipy.sparse.lil_matrix((n,n))
    for j in range(0,n):

        A[j,j]=cc
        if j==0:
            A[j,j+1]=dd
            A[j,j+5]=ee
        elif j==n-1:
            A[j,j-1]=bb
            A[j,j-5]=aa
        elif j<5:
            A[j,j-1]=bb
            A[j,j+1]=dd
            A[j,j+5]=ee
        elif j>n-6:
            A[j,j+1]=dd
            A[j,j-1]=bb
            A[j,j-5]=aa
        else:
            A[j,j-1]=bb
            A[j,j+1]=dd
            A[j,j+5]=ee
            A[j,j-5]=aa
    end_time=time.time()-start_time
    build_time.append(end_time)

    """
    if n==ns[0]:
        plt.figure(figsize=(5,5),dpi=300)
        plt.grid()
        plt.spy(A)
        plt.show()
        plt.close()
    """
    b=np.random.rand(n)

    start_time=time.time()
    A1=A.tocsc()
    end_time=time.time()-start_time
    conversion_time1.append(end_time)

    start_time=time.time()
    x=scipy.sparse.linalg.spsolve(A1,b)
    end_time=time.time()-start_time
    solve_time_off_pent2.append(end_time)

    norm1.append(np.linalg.norm(A.dot(x)-b))


    start_time=time.time()
    A2=A.tocsr()
    end_time=time.time()-start_time
    conversion_time2.append(end_time)

    start_time=time.time()
    x=scipy.sparse.linalg.spsolve(A2,b)
    end_time=time.time()-start_time
    solve_time_off_pent1.append(end_time)

    norm2.append(np.linalg.norm(A.dot(x)-b))

plt.figure(figsize=(5,5),dpi=300)
plt.grid()

plt.loglog()
plt.ylabel(r"Time [$s$]")
plt.xlabel(r"n")
P,c=  np.polyfit(np.log(np.asarray(ns)),np.log(np.asarray(solve_time_tri)),1) #fitting the line
plt.plot(ns,solve_time_tri,marker=".",label=r"Tridagonal, CSR $O(n^{" +"{:.3f}".format(P)+"})$")

P,c=  np.polyfit(np.log(np.asarray(ns)),np.log(np.asarray(solve_time_off_pent1)),1) #fitting the line
plt.plot(ns,solve_time_off_pent1,marker=".",label="Block Diagonal, CSR, $O(n^{" +"{:.3f}".format(P)+"})$")

P,c=  np.polyfit(np.log(np.asarray(ns)),np.log(np.asarray(solve_time_off_pent2)),1) #fitting the line
plt.plot(ns,solve_time_off_pent2,marker=".",label="Block Diagonal, CRC $O(n^{" +"{:.3f}".format(P)+"})$")
plt.legend()
plt.show()
plt.close()



plt.figure(figsize=(5,5),dpi=300)
plt.grid()

plt.loglog()
plt.ylabel(r"Time [$s$]")
plt.xlabel(r"n")
P,c=  np.polyfit(np.log(np.asarray(ns)),np.log(np.asarray(conversion_time)),1) #fitting the line
plt.plot(ns,conversion_time,marker=".",label=r"Tridagonal, CSR $O(n^{" +"{:.3f}".format(P)+"})$")


P,c=  np.polyfit(np.log(np.asarray(ns)),np.log(np.asarray(conversion_time1)),1) #fitting the line
plt.plot(ns,conversion_time1,marker=".",label="Block Diagonal, CSR, $O(n^{" +"{:.3f}".format(P)+"})$")

P,c=  np.polyfit(np.log(np.asarray(ns)),np.log(np.asarray(conversion_time2)),1) #fitting the line
plt.plot(ns,conversion_time2,marker=".",label="Block Diagonal, CRC $O(n^{" +"{:.3f}".format(P)+"})$")
plt.legend()
plt.show()
plt.close()

plt.figure(figsize=(5,5),dpi=300)
plt.grid()

plt.loglog()
plt.ylabel(r"||Ax-b||$_2$")
plt.xlabel(r"n")
P,c=  np.polyfit(np.log(np.asarray(ns)),np.log(np.asarray(norm0)),1) #fitting the line
plt.plot(ns,norm0,marker=".",label=r"Tridagonal, CSR $O(n^{" +"{:.3f}".format(P)+"})$")


P,c=  np.polyfit(np.log(np.asarray(ns)),np.log(np.asarray(norm1)),1) #fitting the line
plt.plot(ns,norm1,marker=".",label="Block Diagonal, CSR, $O(n^{" +"{:.3f}".format(P)+"})$")

P,c=  np.polyfit(np.log(np.asarray(ns)),np.log(np.asarray(norm2)),1) #fitting the line
plt.plot(ns,norm2,marker=".",label="Block Diagonal, CRC $O(n^{" +"{:.3f}".format(P)+"})$")
plt.legend()
plt.show()
plt.close()
```

```python
# -*- coding: utf-8 -*-
"""
Created on Mon Apr 10 13:56:26 2023

@author: Austin Smith
"""
import numpy as np
import time
from scipy.sparse  import dia_matrix
import scipy
import matplotlib.pyplot as plt

def thomas(a,b,c,d):
    #if linalg.norm(a)+linalg.norm(c) > linalg.norm(b):
    #    print('WARNING: ILL-CONDITIONED MATRIX, RESULTS MAY BE INNACURATE')

    #d must must be a numpy array
    cp_arr = np.zeros(len(d))
    dp_arr = np.zeros(len(d))
    v_arr =  np.zeros(len(d))
    cp_arr[0] = c/b
    dp_arr[0] = d[0]/b

    for j in range(1,len(d)):
        cp_arr[j] = c/(b-a*cp_arr[j-1])
        dp_arr[j] = (d[j]-a*dp_arr[j-1])/(b-a*cp_arr[j-1])
    v_arr[-1] = dp_arr[-1]
    for j in range(len(d)-2,-1,-1):
        v_arr[j] = dp_arr[j]-v_arr[j+1]*cp_arr[j]
    return v_arr


ns = [10**4,3*10**4,5*10**4,8*10**4,10**5,3*10**5,5*10**5,8*10**5,10**6,3*10**6,5*10**6,8*10**6,10**7,10**8]
norm_sp=[]
norm_AS=[]
build_time_sp=[]
conversion_time_sp=[]
solve_time_tri_sp=[]
solve_time_tri_AS=[]
SPCount=0
for n in ns:
    start_time=time.time()

    ex = np.ones(n)
    data = np.array([ex, 100.0 * ex, ex])
    offsets = np.array([-1, 0, 1])
    A=dia_matrix((data, offsets), shape=(n, n))

    end_time=time.time()-start_time
    build_time_sp.append(end_time)

    b=np.random.rand(n)+1.0

    if n < 3*10**6 + 1:
        start_time=time.time()
        A=A.tocsr()
        end_time=time.time()-start_time
        conversion_time_sp.append(end_time)

        start_time=time.time()
        x=scipy.sparse.linalg.spsolve(A,b)
        end_time=time.time()-start_time
        solve_time_tri_sp.append(end_time)

        norm_sp.append(np.linalg.norm(A.dot(x)-b))
        SPCount+=1
    start_time=time.time()
    x=thomas(1.0,100.0,1.0,b)
    end_time=time.time()-start_time
    solve_time_tri_AS.append(end_time)

    norm_AS.append(np.linalg.norm(A.dot(x)-b))

plt.figure(figsize=(5,5),dpi=300)
plt.grid()

plt.loglog()
plt.ylabel(r"Time [$s$]")
plt.xlabel(r"n")
P,c=  np.polyfit(np.log(np.asarray(ns[:SPCount])),np.log(np.asarray(solve_time_tri_sp[:SPCount])),1) #fitting the line
plt.plot(ns[:SPCount],solve_time_tri_sp[:SPCount],marker=".",label=r"Scipy $O(n^{" +"{:.3f}".format(P)+"})$")

P,c=  np.polyfit(np.log(np.asarray(ns)),np.log(np.asarray(solve_time_tri_AS)),1) #fitting the line
plt.plot(ns,solve_time_tri_AS,marker=".",label=r"Austin Smith $O(n^{" +"{:.3f}".format(P)+"})$")

plt.legend()
plt.show()
plt.close()

plt.figure(figsize=(5,5),dpi=300)
plt.grid()

plt.loglog()
plt.ylabel(r"||Ax-b||$_2$")
plt.xlabel(r"n")
P,c=  np.polyfit(np.log(np.asarray(ns[:SPCount])),np.log(np.asarray(norm_sp[:SPCount])),1) #fitting the line
plt.plot(ns[:SPCount],norm_sp[:SPCount],marker=".",label=r"Scipy $O(n^{" +"{:.3f}".format(P)+"})$")

P,c=  np.polyfit(np.log(np.asarray(ns)),np.log(np.asarray(norm_AS)),1) #fitting the line
plt.plot(ns,norm_AS,marker=".",ls="--",label="Austin Smith $O(n^{" +"{:.3f}".format(P)+"})$")

plt.legend()
plt.show()
plt.close()
```

```python
# -*- coding: utf-8 -*-
"""
Created on Mon Apr 10 15:42:58 2023

@author: 15303
"""

import numpy as np
import time
from scipy.sparse import dia_matrix
import scipy
import matplotlib.pyplot as plt


#ns1 = [10**3,2*10**3,3*10**3,4*10**3,5*10**3,6*10**3,7*10**3,8*10**3,9*10**3,10**4]#,3*10**5,5*10**5,8*10**5,10**6,3*10**6]
ns2 = [10**5,2*10**5,3*10**5,4*10**5,5*10**5,6*10**5,7*10**5,8*10**5,9*10**5,10**6,2*10**6,3*10**6,4*10**6,5*10**6,6*10**6,7*10**6,8*10**6,9*10**6,10**7]#,3*10**5,5*10**5,8*10**5,10**6,3*10**6]

norm1=[]
norm2=[]
norm3=[]
norm4=[]
norm5=[]
norm6=[]
norm7=[]
norm8=[]
norm9=[]
norm10=[]
norm11=[]

build_time=[]
conversion_time=[]

solve_time_1=[]
solve_time_2=[]
solve_time_3=[]
solve_time_4=[]
solve_time_5=[]
solve_time_6=[]
solve_time_7=[]
solve_time_8=[]
solve_time_9=[]
solve_time_10=[]
solve_time_11=[]

"""
for n in ns1:
    print(n)
    start_time=time.time()

    ex = np.ones(n)
    data = np.array([ex, 100.0 * ex, ex])
    offsets = np.array([-1, 0, 1])
    D=dia_matrix((data, offsets), shape=(n, n))

    O=scipy.sparse.lil_matrix((n,n))
    for i in range(0,n):
        r_col=np.random.randint(0,n-1)
        r_row=np.random.randint(0,n-1)
        O[r_row,r_col]=1.0
        O[r_col-n,r_row-n]=1.0
    A1=D.tocsr()+O.tocsr()

    end_time=time.time()-start_time
    build_time.append(end_time)



    b=np.random.rand(n)+1.0

    start_time=time.time()
    x=scipy.sparse.linalg.spsolve(A1,b)
    end_time=time.time()-start_time
    solve_time_1.append(end_time)

    norm1.append(np.linalg.norm(A1.dot(x)-b))
"""
build_time=[]
for n in ns2:
    print(n)
    start_time=time.time()

    ex = np.ones(n)
    data = np.array([ex, 100.0 * ex, ex])
    offsets = np.array([-1, 0, 1])
    D=dia_matrix((data, offsets), shape=(n, n))

    O=scipy.sparse.lil_matrix((n,n))
    for i in range(0,n):
        r_col=np.random.randint(0,n-1)
        r_row=np.random.randint(0,n-1)
        O[r_row,r_col]=1.0
        O[r_col-n,r_row-n]=1.0
    A=D.tocsr()+O.tocsr()

    end_time=time.time()-start_time
    build_time.append(end_time)



    b=np.random.rand(n)+1.0


    """
    start_time=time.time()
    x=scipy.sparse.linalg.bicg(A,b)
    end_time=time.time()-start_time
    solve_time_2.append(end_time)

    norm2.append(np.linalg.norm(A.dot(x[0])-b))

    start_time=time.time()
    x=scipy.sparse.linalg.bicgstab(A,b)
    end_time=time.time()-start_time
    solve_time_3.append(end_time)

    norm3.append(np.linalg.norm(A.dot(x[0])-b))
    """
    start_time=time.time()
    x=scipy.sparse.linalg.cg(A,b)
    end_time=time.time()-start_time
    solve_time_4.append(end_time)

    norm4.append(np.linalg.norm(A.dot(x[0])-b))
    """
    start_time=time.time()
    x=scipy.sparse.linalg.cgs(A,b)
    end_time=time.time()-start_time
    solve_time_5.append(end_time)

    norm5.append(np.linalg.norm(A.dot(x[0])-b))
    """
    start_time=time.time()
    x=scipy.sparse.linalg.gmres(A,b)
    end_time=time.time()-start_time
    solve_time_6.append(end_time)

    norm6.append(np.linalg.norm(A.dot(x[0])-b))
    """
    start_time=time.time()
    x=scipy.sparse.linalg.lgmres(A,b)
    end_time=time.time()-start_time
    solve_time_7.append(end_time)

    norm7.append(np.linalg.norm(A.dot(x[0])-b))

    start_time=time.time()
    x=scipy.sparse.linalg.minres(A,b)
    end_time=time.time()-start_time
    solve_time_8.append(end_time)

    norm8.append(np.linalg.norm(A.dot(x[0])-b))

    start_time=time.time()
    x=scipy.sparse.linalg.qmr(A,b)
    end_time=time.time()-start_time
    solve_time_9.append(end_time)

    norm9.append(np.linalg.norm(A.dot(x[0])-b))

    start_time=time.time()
    x=scipy.sparse.linalg.gcrotmk(A,b)
    end_time=time.time()-start_time
    solve_time_10.append(end_time)

    norm10.append(np.linalg.norm(A.dot(x[0])-b))

    start_time=time.time()
    x=scipy.sparse.linalg.tfqmr(A,b)
    end_time=time.time()-start_time
    solve_time_11.append(end_time)

    norm11.append(np.linalg.norm(A.dot(x[0])-b))
    """
plt.figure(figsize=(5,5),dpi=300)
plt.grid()

plt.loglog()
plt.ylabel(r"Time [$s$]")
plt.xlabel(r"n")
"""
P,c= np.polyfit(np.log(np.asarray(ns2)),np.log(np.asarray(solve_time_2)),1) #fitting the line
plt.plot(ns2,solve_time_2,marker=".",label="BICG $O(n^{" +"{:.3f}".format(P)+"})$")

P,c= np.polyfit(np.log(np.asarray(ns2)),np.log(np.asarray(solve_time_3)),1) #fitting the line
plt.plot(ns2,solve_time_3,marker=".",label="BICGSTAB $O(n^{" +"{:.3f}".format(P)+"})$")
"""
P,c= np.polyfit(np.log(np.asarray(ns2)),np.log(np.asarray(solve_time_4)),1) #fitting the line
plt.plot(ns2,solve_time_4,marker=".",label="CG $O(n^{" +"{:.3f}".format(P)+"})$")

"""
P,c= np.polyfit(np.log(np.asarray(ns2)),np.log(np.asarray(solve_time_5)),1) #fitting the line
plt.plot(ns2,solve_time_5,marker=".",label="CGS $O(n^{" +"{:.3f}".format(P)+"})$")
"""

P,c= np.polyfit(np.log(np.asarray(ns2)),np.log(np.asarray(solve_time_6)),1) #fitting the line
plt.plot(ns2,solve_time_6,marker=".",label="GMRES $O(n^{" +"{:.3f}".format(P)+"})$")
"""
P,c= np.polyfit(np.log(np.asarray(ns2)),np.log(np.asarray(solve_time_7)),1) #fitting the line
plt.plot(ns2,solve_time_7,marker=".",label="LGMRES $O(n^{" +"{:.3f}".format(P)+"})$")

P,c= np.polyfit(np.log(np.asarray(ns2)),np.log(np.asarray(solve_time_8)),1) #fitting the line
plt.plot(ns2,solve_time_8,marker=".",label="MINRES $O(n^{" +"{:.3f}".format(P)+"})$")

P,c= np.polyfit(np.log(np.asarray(ns2)),np.log(np.asarray(solve_time_9)),1) #fitting the line
plt.plot(ns2,solve_time_9,marker=".",label="QMR $O(n^{" +"{:.3f}".format(P)+"})$")

P,c= np.polyfit(np.log(np.asarray(ns2)),np.log(np.asarray(solve_time_10)),1) #fitting the line
plt.plot(ns2,solve_time_10,marker=".",label="GCROTMK $O(n^{" +"{:.3f}".format(P)+"})$")

P,c= np.polyfit(np.log(np.asarray(ns2)),np.log(np.asarray(solve_time_11)),1) #fitting the line
plt.plot(ns2,solve_time_11,marker=".",label="TFMQR $O(n^{" +"{:.3f}".format(P)+"})$")

P,c= np.polyfit(np.log(np.asarray(ns1)),np.log(np.asarray(solve_time_1)),1) #fitting the line
plt.plot(ns1,solve_time_1,marker=".",c="k",label="SPSOLVE $O(n^{" +"{:.3f}".format(P)+"})$")
"""

plt.plot(ns2,np.asarray(ns2)*np.log(np.asarray(ns2))/(1.0*10**6),ls=":",c="k")
plt.legend()
plt.show()
plt.close()




plt.figure(figsize=(5,5),dpi=300)
plt.grid()

plt.loglog()
plt.ylabel(r"Time [$s$]")
plt.xlabel(r"n")

P,c= np.polyfit(np.log(np.asarray(ns2)),np.log(np.asarray(build_time)),1) #fitting the line
plt.plot(ns2,build_time,marker=".",label="$O(n^{" +"{:.3f}".format(P)+"})$")
plt.legend()
plt.show()
plt.close()



plt.figure(figsize=(5,5),dpi=300)
plt.grid()

plt.loglog()
plt.ylabel(r"||Ax-b||$_2$")
plt.xlabel(r"n")


"""
P,c= np.polyfit(np.log(np.asarray(ns2)),np.log(np.asarray(norm2)),1) #fitting the line
plt.plot(ns2,norm2,marker=".",label="BICG $O(n^{" +"{:.3f}".format(P)+"})$")
P,c= np.polyfit(np.log(np.asarray(ns2)),np.log(np.asarray(norm3)),1) #fitting the line
plt.plot(ns2,norm3,marker=".",label="BICSTAB $O(n^{" +"{:.3f}".format(P)+"})$")
"""
P,c= np.polyfit(np.log(np.asarray(ns2)),np.log(np.asarray(norm4)),1) #fitting the line
plt.plot(ns2,norm4,marker=".",label="CG $O(n^{" +"{:.3f}".format(P)+"})$")
"""
P,c= np.polyfit(np.log(np.asarray(ns2)),np.log(np.asarray(norm5)),1) #fitting the line
plt.plot(ns2,norm5,marker=".",label="CGS $O(n^{" +"{:.3f}".format(P)+"})$")
"""
P,c= np.polyfit(np.log(np.asarray(ns2)),np.log(np.asarray(norm6)),1) #fitting the line
plt.plot(ns2,norm6,marker=".",label="GMRES $O(n^{" +"{:.3f}".format(P)+"})$")
"""
P,c= np.polyfit(np.log(np.asarray(ns2)),np.log(np.asarray(norm7)),1) #fitting the line
plt.plot(ns2,norm7,marker=".",label="LGMRES $O(n^{" +"{:.3f}".format(P)+"})$")

P,c= np.polyfit(np.log(np.asarray(ns2)),np.log(np.asarray(norm8)),1) #fitting the line
plt.plot(ns2,norm8,marker=".",label="MINRES $O(n^{" +"{:.3f}".format(P)+"})$")
P,c= np.polyfit(np.log(np.asarray(ns2)),np.log(np.asarray(norm9)),1) #fitting the line
plt.plot(ns2,norm9,marker=".",label="QMR $O(n^{" +"{:.3f}".format(P)+"})$")

P,c= np.polyfit(np.log(np.asarray(ns2)),np.log(np.asarray(norm10)),1) #fitting the line
plt.plot(ns2,norm10,marker=".",label="GROTMK $O(n^{" +"{:.3f}".format(P)+"})$")

P,c= np.polyfit(np.log(np.asarray(ns2)),np.log(np.asarray(norm11)),1) #fitting the line
plt.plot(ns2,norm11,marker=".",label="TFQMR $O(n^{" +"{:.3f}".format(P)+"})$")

P,c= np.polyfit(np.log(np.asarray(ns1)),np.log(np.asarray(norm1)),1) #fitting the line
plt.plot(ns1,norm1,marker=".",c="k",label="SPSOLVE $O(n^{" +"{:.3f}".format(P)+"})$")
"""


plt.legend()
plt.show()
plt.close()
```

References

[1] D. Cohen, "Simplified control of FFT hardware," in IEEE Transactions on Acoustics, Speech, and Signal Processing, vol. 24, no. 6, pp. 577-579, December 1976, doi: 10.1109/TASSP.1976.1162854.

[2] S. Magar, S. Shen, G. Luikuo, M. Fleming and R. Aguilar, "An application specific DSP chip set for 100 MHz data rates," in ICASSP-88., International Conference on Acoustics, Speech, and Signal Processing, New York, NY, USA, 1988 pp. 1989,1990,1991,1992.
doi: 10.1109/ICASSP.1988.197015

[3] T. Abtahi, C. Shea, A. Kulkarni and T. Mohsenin, "Accelerating Convolutional Neural Network With FFT on Embedded Hardware," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 26, no. 9, pp. 1737-1749, Sept. 2018, doi: 10.1109/TVLSI.2018.2825145.

[4] Y. Liu, C. Wang, J. Sun, S. Du and Q. Hong, "One-Step Calculation Circuit of FFT and Its Application," in IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 69, no. 7, pp. 2781-2793, July 2022, doi: 10.1109/TCSI.2022.3159803.

[5] A. Rupp, J. Pelzl, C. Paar, M. C. Mertens and A. Bogdanov, "A Parallel Hardware Architecture for fast Gaussian Elimination over GF(2)," 2006 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, Napa, CA, USA, 2006, pp. 237-248, doi: 10.1109/FCCM.2006.12.

[6] A.H. Baker and E.R. Jessup and T. Manteuffel, “A Technique for Accelerating the Convergence of Restarted GMRES”, SIAM J. Matrix Anal. Appl. 26, 962 (2005).

[7] L. H. Thomas , "A Practical Method for the Solution of Certain Problems in Quantum Mechanics by Successive Removal of Terms from the Hamiltonian by Contact Transformations of the Dynamical Variables Part I. General Theory", J. Chem. Phys. 10, 532-537 (1942) https://doi.org/10.1063/1.1723760

[8] FHP_Thomas_talk_12min.ppt (aps.org)

[9] E. de Sturler, ‘’Truncation strategies for optimal Krylov subspace methods’’, SIAM J. Numer. Anal. 36, 864 (1999).

[10] S. C. Eisenstat, H. C. Elman, and M. H. Schultz, Variational iterative methods for nonsymmetric systems of linear equations, SIAM Journal on Numerical Analysis, 20 (1983), pp. 345–357

[11] E. de Sturler, Nested Krylov methods based on GCR, Journal of Computational and Applied Mathematics, 67 (1996), pp. 15–41.

[12] S. Li, W. Wu, E. Darve, A fast algorithm for sparse matrix computations related to inversion, Journal of Computational Physics, Volume 242, 2013, Pages 915-945, ISSN 0021-9991, https://doi.org/10.1016/j.jcp.2013.01.036.