1. Introduction

This chapter concerns the numerical solution of linear systems,

$$Au = b,$$

where the linear system comes from discretization of PDEs. Such linear systems are characterized by being large, that is A is typically a $n \times n$ matrix where n is between 10^6 to 10^9. Furthermore, the matrix is typically extremely sparse, typically containing only $\mathcal{O}(n)$ non-zeros. However, A^{-1} will typically full. Finally, because the matrix is a discretized differential operator, the condition number is typically large, and the condition number increases as we increase the mesh resolution. Because of these special characteristics, special considerations needed in the choice of linear solver. The difference between some different algorithms are illustrated in the following example.

Example 1.1 (CPU times of different algorithms)**.** *In this example we will solve the problem*

$$\begin{aligned} u - \Delta u &= f, \\ \frac{\partial u}{\partial n} &= 0, \end{aligned}$$

on the unit square with first order Lagrange elements. The problem is solved with four different methods:

- *a LU solver,*
- *Conjugate Gradient method,*
- *Conjugate Gradient method with an ILU preconditioner,*
- *Conjugate Gradient method with an AMG preconditioner.*

Figure 1.1 shows that there is a dramatic difference between the different algorithms. In fact, the Conjugate gradient (CG) with an AMG preconditioner is 22 times faster than the slowest method, which is the CG solver without preconditioner. However, the log-log plot reveals that all algorithms appears to have polynomial complexity, i.e., $\mathcal{O}(N^\beta)$, with respect to number of degrees of freedom, N. The order of the polynomial β vary, though.

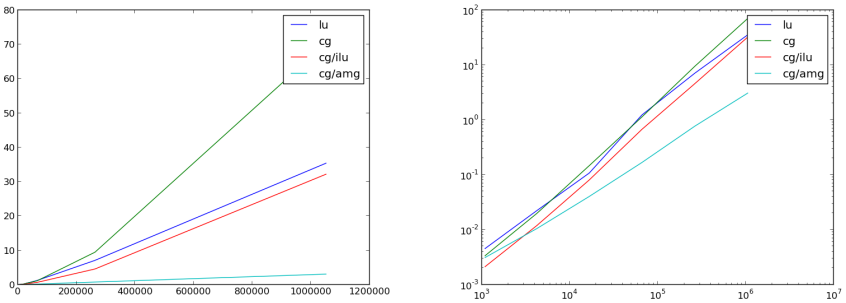

Figure 1. The left figure shows the CPU time (in seconds) used to solve the linear system with different mesh resolutions (N, N). The right figure shows a log-log plot of the same data.

The code used in this experiment is as follows:

```
import time
from dolfin import *
lu_time = []
cg_time = []
cgilu_time = []
cgamg_time = []
Ns = []

parameters["krylov_solver"]["relative_tolerance"] =
    1.0e-8
parameters["krylov_solver"]["absolute_tolerance"] =
    1.0e-8
parameters["krylov_solver"]["monitor_convergence"] =
    True
parameters["krylov_solver"]["report"] = True
parameters["krylov_solver"]["maximum_iterations"] =
    50000

for N in [32, 64, 128, 256, 512, 1024]:

  Ns.append(N)

  mesh = UnitSquare(N, N)
  print " N ", N, " dofs ", mesh.num_vertices()
  V = FunctionSpace(mesh, "Lagrange", 1)
  u = TrialFunction(V)
  v = TestFunction(V)

  f = Expression("sin(x[0]*12) - x[1]")
  a = u*v*dx  + inner(grad(u), grad(v))*dx
  L = f*v*dx

  U = Function(V)

  A = assemble(a)
  b = assemble(L)

  t0 = time.time()
  solve(A, U.vector(), b, "lu")
  t1 = time.time()
  print "Time for lu ", t1-t0
  lu_time.append(t1-t0)

  t0 = time.time()
  U.vector()[:] = 0
  solve(A, U.vector(), b, "cg")
  t1 = time.time()
  print "Time for cg ", t1-t0
  cg_time.append(t1-t0)

  t0 = time.time()
  U.vector()[:] = 0
  solve(A, U.vector(), b, "cg", "ilu")
  t1 = time.time()
```

```
  print "Time for cg/ilu ", t1-t0
  cgilu_time.append(t1-t0)

  t0 = time.time()
  U.vector()[:] = 0
  solve(A, U.vector(), b, "cg", "amg")
  t1 = time.time()
  print "Time for cg/amg ", t1-t0
  cgamg_time.append(t1-t0)


import pylab

pylab.plot(Ns, lu_time)
pylab.plot(Ns, cg_time)
pylab.plot(Ns, cgilu_time)
pylab.plot(Ns, cgamg_time)
pylab.legend(["lu", "cg", "cg/ilu", "cg/amg"])
pylab.show()

pylab.loglog(Ns, lu_time)
pylab.loglog(Ns, cg_time)
pylab.loglog(Ns, cgilu_time)
pylab.loglog(Ns, cgamg_time)
pylab.legend(["lu", "cg", "cg/ilu", "cg/amg"])
pylab.show()
```

2. The Richardson iteration

In Example 1.1 we considered one direct method, namely the LU solver, and three iterative methods involving CG with and without different preconditioners. However, before we start with these advanced methods we will consider the simplest iterative method: the so-called *Richardson iteration*:

$$u^n = u^{n-1} - \tau(f - Au^{n-1})$$

The iteration is a linear since it is realized as a matrix $(I - \tau A)$ times the solution on the previous iteration plus a vector (τf). Furthermore, the iteration requires $\mathcal{O}(n)$ floating points of memory. Whether it is a convergent iteration and the number of iterations required to obtain to achieve a reasonable solution is however unclear.

To analyse the speed of convergence we consider the error in the n'th iteration,

$$e^n = u^n - u$$

By subtracting u from both sides and using the fact that $Au = b$, we obtain the recursion formula for the error,

$$e^n = e^{n-1} - \tau A e^{n-1}.$$

The iteration will be convergent if

$$\|e^n\| \leq \|e^{n-1}\|$$

Or

$$\|(I - \tau A)e^{n-1}\| \leq \|e^{n-1}\|$$

Hence, the Richardson iteration will be convergent if

$$\|I - \tau A\| \leq \rho < 1$$

Or

$$0 < (1 - \rho) \leq \|\tau A\| \leq (1 + \rho) < 2$$

Therefore, convergence may always be obtained if we choose $\tau = \frac{c}{\|A\|}$, and $c < 2$. In fact, the optimal τ is $\tau = \frac{2}{\|A\|+1/\|A^{-1}\|}$.

Example 2.1 (The Richardson iteration on a Poisson problem)**.** *Let us consider the Richardson iteration on a two point boundary problem discretized with a finite difference method,*

$$\begin{aligned} (Au)_i = -\frac{u_{i-1} - 2u_i + u_{i+1}}{h^2} &= f_i, \quad , i = 1, \ldots, N, \\ u_0 &= 0, \\ u_{N+1} &= 0, \end{aligned}$$

In Figure 2.1 the eigenvalues of A is shown for $N = 150$. The smallest and largest eigenvalues are approximately 9.6 and 89 000, respectively, and we see that there is an even distribution between these extreme values. Furthermore, in the log-log plot, the nearly linear line suggests that the eigenvalue distribution is resonably well approximated as Ch^{-2}.

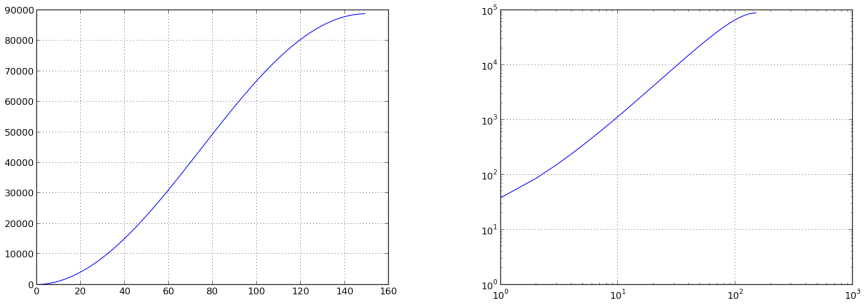

FIGURE 2. The eigenvalues of the stiffness matrix of the 1D Poisson problem for $N = 150$ is shown in the left picture, while a log-log plot of the same eigenvalues are shown to the right.

Figure 2.1 show the numerical solution at different iterations when using the Richardson method for two different τ values. In both cases it is clear that the solution improves quite a lot in the first 10 iterations, but slow down since a large part of the error remain essentially unchanged during the iterations. The left figure clearly shows that the solution after 1000 iterations is dominated by low and high frequency components. The figure on the right, where $\tau = \frac{0.9}{\max_i \lambda_i}$ shows that the high frequency components of the error can be removed by choosing a larger than optimal τ. However, in both cases the low frequency components or the error remains essentially unchanged throughout the iterations.

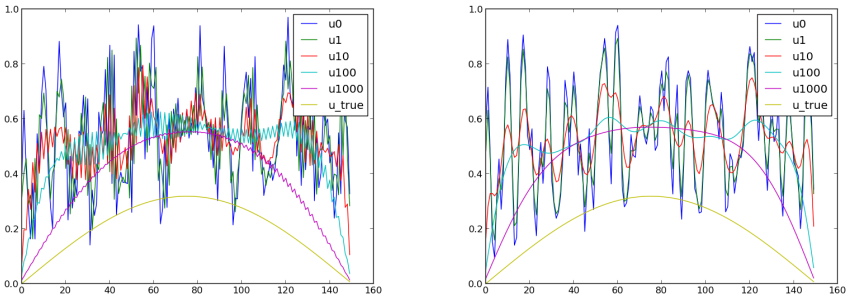

FIGURE 3. The numerical solution at different steps of the Richardson method is shown. The left figure uses the optimal τ, that is, $\tau = \frac{2}{\max_i \lambda_i + \| \min_i \lambda_i}$ while $\tau = \frac{0.9}{\max_i \lambda_i}$ is used in the right figure.

The code used in this example is as follows:

```
import numpy

def create_stiffness_matrix(N):
  h = 1.0/(N-1)
  A = numpy.zeros([N,N])
  for i in range(N):
    A[i,i] = 2.0/(h**2)
    if i > 0:
      A[i,i-1] = -1.0/(h**2)
    if i < N-1:
      A[i,i+1] = -1.0/(h**2)
  A = numpy.matrix(A)
  return A

import pylab
from numpy import linalg

N = 150
x = numpy.arange(0, 1, 1.0/(N))
f = numpy.matrix(numpy.sin(3.14*x)).transpose()
u_true =
    (1.0/3.14)*numpy.matrix(numpy.sin(3.14*x)).transpose()
u0 = numpy.matrix(numpy.random.random(N)).transpose()
u_prev = u0
A = create_stiffness_matrix(N)

eigenvalues = linalg.eigvals(A)
eigenvalues = numpy.sort(eigenvalues)

lambda_max = eigenvalues[-1]

tau = 0.9/lambda_max
for i in range(1001):
  u = u_prev - tau*(A*u_prev - f)
  u_prev = u
```

```
    if i == 0 or i == 1 or i == 10 or i == 100 or i ==
        1000:
        pylab.plot(u)
pylab.plot(u_true)
pylab.legend(["u0", "u1", "u10", "u100", "u1000",
    "u_true"])
pylab.show()
```

2.1. **Order optimal algorithms and Spectral Equivalence.** Any linear iteration for solving the linear system

$$Au = f,$$

can be written as

$$u^n = u^{n-1} - \tau(f - Au^{n-1}).$$

The convergence factor is

$$\rho = \|I - \tau A\|,$$

We have also seen that we may introduce a preconditioned system

$$BAu = Bf,$$

Here, B is a matrix or linear operator designed to make the condition number of the preconditioned system smaller, while B is cheap (ideally $\mathcal{O}(n)$) in storage and evaluation.

$$u^n = u^{n-1} - \tau B(f - Au^{n-1}).$$

The convergence factor of the preconditioned iteration is

$$\rho = \|I - \tau BA\|,$$

Definition 2.1. *Two matrices or linear operators, B and C, are spectrally equivalent if*

$$c_0(Bu, u) \leq (Cu, u) \leq c_1(Bu, u), \quad \forall u$$

The condition number of $B^{-1}C$, $\text{cond}(B^{-1}C) \leq \frac{c_1}{c_0}$.

Hence a spectral equivalent preconditioner guaranties a convergence rate independent of the mesh resolution and therefore an order optimal algorithm, since ....

3. Krylov methods and preconditioning

Any linear iteration may be written as a Richardson iteration with a preconditioner. However, iterations like Conjugate Gradient method, GMRES, Mininal Residual method, and BiCGStab, are different. These are non-linear iterations methods. We will not go in detail on these methods, but they should be used together with a preconditioner, just as Richardson. Furthmore, some of then have special requirements.

Classification of methods. We classify the methods according to the matrices they solve The matrix may be:

- Symmetric Positive Definite: use Conjugate Gradient with an SPD preconditioner, see also Exercise 3.3.
- Symmetric and indefinite: use Minimal Residual method with and SPD preconditioner, see also Exercise 3.5.
- Positive: GMRES and ILU (or AMG) is often good, but you might need to experiment, see also Exercise 3.4.
- Nonsymmetric and indefinite: all bets are off.

Exercise 3.1. *Implement the Richardson iteration for the 1D Poisson problem with homogenous Dirichlet conditions, using finite differences and a multigrid preconditioner.*

Exercise 3.2. *Estimate the convergence factor for the Jacobi and Gauss-Seidel iteration for the 1D Poisson problem from Example 1.1.*

Exercise 3.3. *Test CG method with no, ILU and AMG preconditioner for the Poisson problem in 1D and 2D with homogenous Dirichlet conditions, with respect to different mesh resolutions. Do some of the iterations suggest spectral equivalence?*

Exercise 3.4. *Test CG, BiCGStab, GMRES with ILU, AMG, and Jacobi preconditioning for*

$$\begin{aligned} -\mu\Delta u + v\nabla u &= f \quad \text{in } \Omega \\ u &= 0 \quad \text{on } \partial\Omega \end{aligned}$$

Here Ω is the unit square, $v = csin(7x)$, and c varies as $1, 10, 100, 1000, 10000$ and the mesh resolution h varies as $1/8, 1/16, 1/32, 1/64$. You may assume homogenous Dirichlet conditions.

Exercise 3.5. *The following code snippet shows the assembly of the matrix and preconditioner for a Stokes problem:*

```
a = inner(grad(u), grad(v))*dx + div(v)*p*dx +
    q*div(u)*dx
L = inner(f, v)*dx

# Form for use in constructing preconditioner matrix
b = inner(grad(u), grad(v))*dx + p*q*dx

# Assemble system
A, bb = assemble_system(a, L, bcs)

# Assemble preconditioner system
P, btmp = assemble_system(b, L, bcs)

# Create Krylov solver and AMG preconditioner
solver = KrylovSolver("tfqmr", "amg")

# Associate operator (A) and preconditioner matrix (P)
solver.set_operators(A, P)
```

```
# Solve
U = Function(W)
solver.solve(U.vector(), bb)
```

Here, "tfqmr" is a variant of the Minimal residual method and "amg" is an algebraic multigrid implementation in HYPRE. Test, by varying the mesh resolution, whether the code produces an order–optimal preconditioner. HINT: You might want to change the "parameters" as done in Example 1.1:

```
# Create Krylov solver and AMG preconditioner
solver = KrylovSolver("tfqmr", "amg")
solver.parameters["relative_tolerance"] = 1.0e-8
solver.parameters["absolute_tolerance"] = 1.0e-8
solver.parameters["monitor_convergence"] = True
solver.parameters["report"] = True
solver.parameters["maximum_iterations"] = 50000
```