1 Iterative methods

By Anders Logg, Kent–Andre Mardal

This chapter concerns the numerical solution of large linear systems,

$$Au = b,$$

where the linear system comes from discretization of PDEs. In such linear systems A is a $N \times N$ matrix, and N might be between 10^6 and 10^9. Furthermore, the matrix is normally extremely sparse and contains only $\mathcal{O}(N)$ nonzeros. However, A^{-1} will typically be full. A Naive Gauss-elimination requires $\mathcal{O}(N^2)$ — $\mathcal{O}(N^3)$ floating point operations and $\mathcal{O}(N^{2/3})$ — $\mathcal{O}(N^2)$ floating point numbers of storage. Iterative methods need $\mathcal{O}(N)$ floating point operations and $\mathcal{O}(N)$ floating point numbers of storage.

1.1 A simple iterative method: Richardson iteration

Let us consider the problem: Find u such that

$$Au = b.$$

The Richardson iteration is

$$u^n = u^{n-1} - \tau(Au^{n-1} - b), \tag{1.1}$$

where τ is a parameter that must be determined. We see that if $u^{n-1} = u$, then $u^n = u$. Hence the iteration will not change the solution.

The standard approach to analyze iterative methods is to look at what happens with the error. Let $e^n = u^n - u$. We subtract u from both sides of (1.1) and obtain

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

We may therefore quantify the error in the L_2 norm,

$$\|e^n\| = \|e^{n-1} - \tau A e^{n-1}\| \leqslant \|I - \tau A\| \|e^{n-1}\|$$

We see if $\|I - \tau A\| < 1$, then the iteration will be convergent.
Lets say that we need to reduce the error by a factor of ϵ, that is, we need $\frac{\|e^n\|}{\|e^0\|} < \epsilon$. Let $\rho = \|I - \tau A\|$, then

$$\|e^n\| \leqslant \rho \|e^{n-1}\| \leqslant \rho^n \|e^0\|. \tag{1.2}$$

Assuming equality in the equation (1.2) and $\frac{\|e^n\|}{\|e^0\|} = \epsilon$. Then the number of iterations needed to achieve the desired error is:

$$n = \frac{\log \epsilon}{\log \rho}.$$

If n is independent of the resolution of the discretization, the algorithm will be **order-optimal**, meaning, the algorithm will cost $\mathcal{O}(N)$ floating point operations and storage.
Let us look at an example to see what happens with the error when we use the Richardson iteration (also the same happens for Jacobi, Gauss Seidel, etc.).

Example 1.1 (1D Poisson equation)**.** *The Richardson iteration on the Poisson equation in 1D, discretized with finite difference method (FDM).*

$$Lu = \begin{cases} -u'' = f \quad for \quad x \in (0,1) \\ u(0) = u(1) = 0 \end{cases} \tag{1.3}$$

Eigenvalues and eigenfunctions of Lu are $\lambda_k = (k\pi)^2$ and $v_k = \sin(k\pi x)$ for $k \in \mathbb{N}$. When discretizing with FDM we get a $Au = b$ system, where A is a tridiagonal matrix ($A = tridiagonal(-1, 2, -1)$). We have the same eigenvectors, but the eigenvalues are a little bit different: $\mu_k = \frac{4}{h^2} sin^2(\frac{k\pi h}{2})$. Here h is the step lenght Δx. We find the smallest and largest discrete eigenvalues

$$\mu_{min}(A) = \pi^2, \quad \mu_{max}(A) = \frac{4}{h^2}.$$

Let $\tau = \frac{c}{\mu_{max}}$ and assume that $e^0 = \sum_{k=1}^{N} c_k v_k$, then, by the Richardson iteration:

$$\begin{aligned} e^n &= (I - \tau A)e^{n-1} \\ &= (I - \tau A)^n e^0 \\ &= (I - \tau A)^n \sum_{k=1}^{N} c_k v_k \\ &= \sum_{k=1}^{N} (1 - \frac{c\mu_k}{\mu_{max}})^n c_k v_k. \end{aligned}$$

When using the equality above and the fact that $\{v_k\}_{k=1}^N$ are orthonormal, we can show that (show this!)

$$\|e^n\| \leqslant (1 - c)^{2n} \|e^0\|.$$

Thus, we have convergence for $0 < c < 2$.

However, different parts of the error will decrease at different speed! To illustrate this, first assume that $e^0 = v_N$, where v_N is the eigenfunction corresponding to the largest eigenvalue, then

$$e^n = (1 - c\frac{\mu_{max}}{\mu_{min}})^n v_N.$$

Setting $c = 0.9$ will reduce the error by a factor of 0.1 per iteration. Now assume that $e^0 = v_1$, where

v_1 is the eigenvector corresponding to the smallest eigenvalue, then[1]

$$e^n = (1 - c\frac{\mu_{min}}{\mu_{max}})^n v_1 = (1 - c\frac{1}{\kappa(A)})^n v_1 \approx (1 - ch^2)^n v_1 \approx e^0$$

for small h. The convergence is very slow in this case.

The idea of writing $e^n = \sum_{k=1}^{N} c_k v_k$, where $v_k = \sin(k\pi x)$ are the eigenvectors of A, is motivated from Fourier series. If e^n is mostly composed of v_k for small k, we will refer to this as the low frequent error. Similarly if e^n is mostly composed of v_k for large k, we refer to this as the high frequent error.

In summary: From the example 1.1 we saw that high frequent error is removed quickly, while low frequent error stays essentially unchanged throughout the iterations. This is common for many methods like Jacobi, Gauss–Seidel, SOR and SSOR.

1.2 Generalizing by a preconditioning

The basic idea of preconditioning is to replace $Au = b$ with $BAu = Bb$. Both systems have the same solution (if B is nonsingular). B should be chosen in a way that BA will have a "nicer" spectrum[2]. Furthermore Bu should cost $\mathcal{O}(N)$ operations to evaluate. The generalized Richardson iteration becomes

$$u^n = u^{n-1} - \tau B(Au^{n-1} - b). \tag{1.4}$$

The error in the n-th iteration is

$$e^n = e^{n-1} - \tau BAe^{n-1}$$

and the iteration is convergent if $\|I - \tau BA\| < 1$.

1.2.1 Preconditioner: Jacobi

Look at the i-th equation of the system $Au = b$, $\sum_{j=1}^{N} a_{ij}u_j = b_i$, this can be rewritten in an iterative way

$$u_i^n = \frac{1}{a_{ii}}(b_i - \sum_{j \neq i} a_{ij}u_j^{n-1}) \quad \text{which holds for} \quad i = 1, 2, ..., N. \tag{1.5}$$

On the Richardson iteration form this becomes,

$$u^n = u^{n-1} - D^{-1}(Au^{n-1} - b). \tag{1.6}$$

Here $D^{-1} = (diag(A))^{-1}$ is the preconditioner. The eigenvalues $\mu_k = \cos(\pi kh)$ of the Jacobi matrix $J = (I - D^{-1}A)$, are displayed in figure 1.1. Here $|\mu_k|$ close to 0 corresponds to high frequent errors and $|\mu_k|$ close to 1 corresponds to low frequent error. We recall from the previous section that high frequent error is easily handled, while low frequent errors is problematic. Therefor we would like $|\mu_k|$ close to zero.

[1] $\kappa(A)$ is the condition number of A, which is $\kappa(A) = |\lambda_{max}/\lambda_{min}|$.

[2] By "nicer" spectrum, we meen smaller condition number, $\kappa BA < \kappa A$

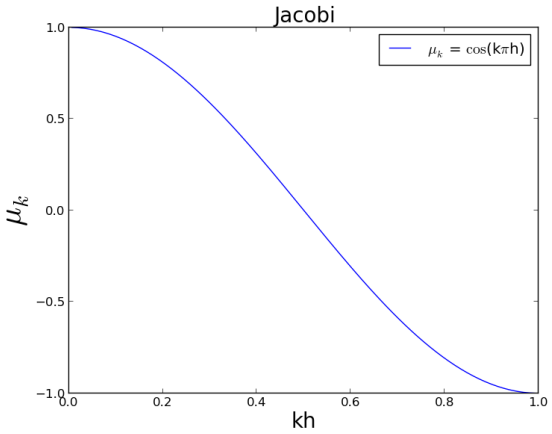

Figure 1.1: Plot of the eigenvalues of the Jacobi method on FDM discretized 1D Poisson with $N = 150$ and $\mu_k = \cos(\pi kh)$.

1.2.2 *Preconditioner: Relaxed Jacobi*

By looking at the Jacobi iteration (1.5), we see that u_i^n is computed based on u_j^{n-1} for $j \neq i$. We can extend (1.5) by including information from the previous iteration u_i^{n-1}

$$u_i^n = (1-\omega)u_i^{n-1} + \frac{\omega}{a_{ii}}(b_i - \sum_{j\neq i} a_{ij}u_j^{n-1}) \quad \text{for} \quad i = 1, 2, ..., N. \tag{1.7}$$

On the Richardson iteration form this becomes,

$$u^n = u^{n-1} - \omega D^{-1}(Au^{n-1} - b). \tag{1.8}$$

The relaxation parameter ω needs to be chosen. By choosing $\omega = 2/3$, we can see from figure 1.2 that more of the eigenvalues correspond to high frequent errors ($|\mu_k|$ close to 0). While the low frequent error (μ_k close to 1) is still a problem.

1.2.3 *Preconditioner: Gauss–Seidel*

A natural extension to the Jacobi iteration (1.5) is to use u_j^n instead of u_j^{n-1} for $j < i$ in the sum, since these values is already computed. This is the Gauss–Seidel iteration,

$$u_i^n = \frac{1}{a_{ii}}(b_i - \sum_{j<i} a_{ij}u_j^n - \sum_{j>i} a_{ij}u_j^{n-1}) \quad \text{for} \quad i = 1, 2, ..., N. \tag{1.9}$$

Let $A = D + U + L$, where D is the diagonal, U is the upper diagonal part and L is the lower diagonal part of A. On the Richardson iteration form this becomes,

$$u^n = u^{n-1} - (D + L)^{-1}(Au^{n-1} - b). \tag{1.10}$$

We can apply the same ideas for the relaxed Jacobi and get a relaxed Gauss–Seidel,

$$u_i^n = (1-\omega)u_i^{n-1} + \frac{\omega}{a_{ii}}(b_i - \sum_{j<i} a_{ij}u_j^n - \sum_{j>i} a_{ij}u_j^{n-1}) \quad \text{for} \quad i = 1, 2, ..., N. \tag{1.11}$$

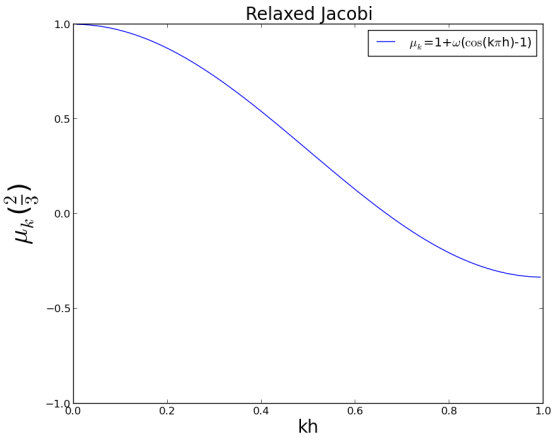

Figure 1.2: Plot of the eigenvalues of the relaxed Jacobi method on FDM discrefized 1D Poisson with $N = 150$ and $\mu_k = 1 + \omega(\cos(\pi kh) - 1)$.

On the Richardson iteration form,

$$u^n = u^{n-1} - \omega(D + L)^{-1}(Au^{n-1} - b). \tag{1.12}$$

Note that Gauss–Seidel is not symmetric (unless A is a diagonal matrix). As mention above, these methods handles high frequent error well, while the low frequent errors remains almost unchanged. The low frequent errors can be handled by multigrid methods.

1.3 Multigrid

The idea is to use smoothers, S, like Jacobi or Gauss–Seidel, to remove the high frequent errors, while the low frequent error is handled with a coarser mesh. These coarser meshes are more efficient to compute on, and are well suited to represent smooth components. The smoother, S, is then used on the coarse meshes.
So, let's concretize, we have $Au = b$. Assume we have performed m relaxed Jacobi iterations. The error is $e^m = u - u^m$ and assume that the error is smooth (containing only low frequent errors). Then,

$$Ae^m = r^m \quad \text{where} \quad r^m = Au^m - b,$$

r^m is the residual. Let us further introduce two different grids,

$$\Omega_h = \{x_j = jh, j = 1, 2, ..., N + 1\}$$
$$\Omega_H = \{x_j = jH, j = 1, 2, ..., M + 1\}.$$

An obvious choice is H = 2h. Instead of solving

$$A_h e_h = r_h, \tag{1.13}$$

we solve

$$A_H e_H = r_H, \tag{1.14}$$

Algorithm 1: 2–level multigrid algorithm

1. Remove the high frequent errors by a Pre–smoothing S (Richardson, Jacobi, GS, e.i.),
$$u^{n+1/3} = u^n - S(Au^n - b). \tag{1.16}$$
2. Solve on a coarser grid,
$$u^{n+2/3} = u^{n+1/3} - I_H^h A_H^{-1}(I_h^H(Au^{n+1/3} - b)). \tag{1.17}$$
3. Post–smoothing,
$$u^{n+1} = u^{n+2/3} - S(Au^{n+2/3} - b). \tag{1.18}$$

To be able to solve (1.14) instead of (1.13), we need an operator that transfer $e_h \to e_H$ and $r_h \to r_H$. A suitable choice is a standard restriction operator I_h^H defined by

$$u_j^H = (I_h^H u_j^h) = \frac{1}{4}(u_{2j-1}^h + 2u_{2j}^h + u_{2j+1}^h) \quad \text{for} \quad j = 1, ..., M. \tag{1.15}$$

Now we may compute e_H as

$$e_H = A_H^{-1} r_H = A_H^{-1} I_h^H r_h.$$

To go back from the coarser mesh we use an interpolation operator defined as $I_H^h = 2(I_h^H)^T$, such that

$$\begin{aligned} u_{2j}^h &= u_j^H \quad \text{for} \quad j = 1, ..., M, \\ u_{2j+1}^h &= \frac{1}{2}(u_j^H + u_{j+1}^H) \quad \text{for} \quad j = 0, .., M. \end{aligned}$$

The 2–level algorithm is summarized in Algorithm 1. Here we have included a smoothing sweep before and after we solve the coarser problem. The post–smoothing gives symmetry, and it also removes any high frequent components introduced by the interpolation operator I_H^h. It can be shown that if S is symmetric and linear, then the multigrid is symmetric and linear. Prove it!

In the 2–level algorithm we only made the grid coarser ones, then solved the system with a direct solver. In realty we may need to make the grid coarser several times before solving directly. We do the same as for the 2–level algorithm, but use the I_H^h operator (makes coarser grid) several times and smoothing each time. When the grid is small enough, we solve it with a direct solver. Then we have to use the interpolant operator I_h^H several times until the grid is at it's original size. Again we smoothed at each interpolation. This procedure is illustrated in figure 1.3 and is know as the V–cycle algorithm. Note that the 2–level algorithm is a special case of the V–cycle algorithm, where we only do one restriction and one interpolation.

1.4 Spectral equivalence and order optimal algorithms

Definition 1.1. *Two linear operators or matrices A^{-1} and B, that are symmetric and positive definite are spectral equivalent if:*

$$c_1(A^{-1}v, v) \leqslant (Bv, v) \leqslant c_2(A^{-1}v, v) \quad \forall \quad v \tag{1.19}$$

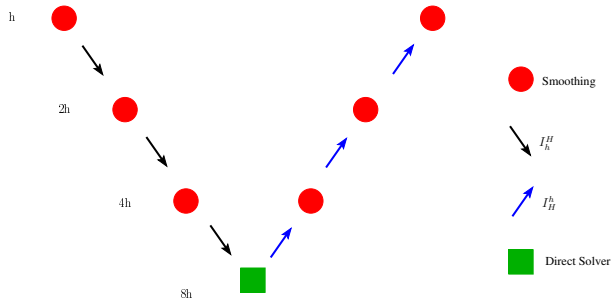

Figure 1.3: V–cycle algorithm

If A^{-1} and B are spectral equivalent, then the condition number of the matrix BA is $\kappa(BA) \leqslant \frac{c_2}{c_1}$. Proper conditions for an order optimal preconditioner:

- B should be spectrally equivalent with A^{-1}.
- The evaluation of B on a vector, Bv, should be $\mathcal{O}(N)$.
- The storage of B should be $\mathcal{O}(N)$.

To see this, we note that $e^n = (I - \tau BA)e^{n-1}$. We can estimate the behavior of e^n by using the A-norm, $\rho_A = \|I - \tau BA\|_A$. Then we get

$$\|e^n\|_A \leqslant \rho_A \|e^{n-1}\|_A.$$

Because BA is symmetric with respect to the A–inner product, ρ_A can be stated in terms of the eigenvalues μ_i of BA, such that

$$\rho_A = \|I - \tau BA\|_A = \sup_{\mu_i} |1 - \tau\mu_i| = \max(|1 - \tau\mu_0|, |1 - \mu_N|),$$

where μ_0 and μ_N is the smallest and largest eigenvalue. We may choose $\tau = \frac{2}{\mu_0 + \mu_N}$ which makes

$$\rho_A = 1 - \tau\mu_0 = 1 - \frac{2\mu_0}{\mu_0 + \mu_N} = \frac{\mu_N - \mu_0}{\mu_N + \mu_0} = \frac{\kappa - 1}{\kappa + 1}.$$

This leads to

$$\|e^n\|_A \leqslant (\frac{\kappa - 1}{\kappa + 1})^n \|e^0\|_A.$$

Hence, if the condition number is independent of the discretization parameter, then the convergent rate is independent as well! This gives an order optimal algorithm (Convergence in $\mathcal{O}(N)$ iterations independent of the discretization).
Multigrid produces a linear operator that is spectrally equivalent with A^{-1} for elliptic operators.

1.5 *Krylov methods and preconditioning*

Any linear iteration method may be written as a Richardson iteration with a preconditioner. However, iterations methods like Conjugate Gradient method, GMRES, Minimal Residual method, and BiCGStab, are different. These are nonlinear iteration methods. We will not go in detail on these methods, but they should be used together with a preconditioner, such as the Richardson methods. Furthermore, some of them have special requirements:

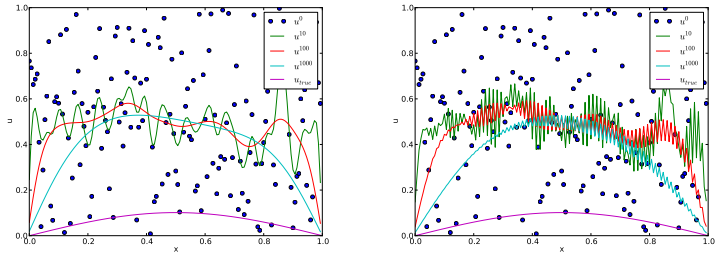

Figure 1.4: The numerical solution u^n from example 1.2 at different step of the Richardson method. The right figure uses the optimal $\tau = \frac{2}{\mu_{max}+\mu_{min}}$, while in the left figure $\tau = \frac{0.9}{\mu_{max}}$.

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

- Symmetric Positive Definite(SPD): Use Conjugate Gradient with an SPD preconditioner, see also Exercise 1.3.
- Symmetric and indefinite: Use Minimal Residual method with and SPD preconditioner, see also Exercise 1.5.
- Positive: GMRES and ILU (or AMG) are often good, but you might need to experiment, see also Exercise 1.4.
- Nonsymmetric and indefinite: All bets are off.

1.6 Examples

Example 1.2 (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 this example we will choose $f_i = \sin(\pi x_i), \quad i = 1, \ldots, N$ *and* $N = 150$*. For the initial vector* u^0 *we use random numbers between* 0 *and* 1*.*
Figure 1.4 shows the numerical solution at different number of iterations for two different τ *values. In both cases it is clear that the solution improves quite a lot in the first 10 iterations, but the improvement slows 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 both low and high frequency components. The figure on the left, where* $\tau = \frac{0.9}{\max_i \mu_i}$*, shows that the high frequency components of the error can be removed by choosing a larger* τ *than the optimal* τ*. However, in both cases the low frequency components of the error remains essentially unchanged throughout the iterations. The code used in this example is as follows:*

Python code

```
import numpy, pylab
from math import pi
from numpy import linalg

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

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

eigenvalues = numpy.sort(linalg.eigvals(A))
mu_max = eigenvalues[-1]
mu_min = eigenvalues[0]

print "Highes eigenvalue", mu_max
print "Lowest eigenvalue", mu_min

def iterate(tau):
  u_prev = u0; u = u0
  pylab.plot(x,u, 'o')
  for i in range(1001):
    if i == 10 or i == 100 or i == 1000 or i == 1000:
      pylab.plot(x,u)
    u = u_prev - tau*(A*u_prev - f)
    u_prev = u
  pylab.plot(x, u_true)
  pylab.xlabel('x')
  pylab.ylabel('u')
  pylab.legend(["$u^0$", "$u^{10}$", "$u^{100}$", "$u^{1000}$", "$u_{true}$"])
  pylab.show()

tau = 0.9/mu_max
iterate(tau)
tau = 2/(mu_max + mu_min)
iterate(tau)
```

Example 1.3 (CPU times of different algorithms)**.** *In this example we will solve the problem*

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

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

- *a LU solver,*

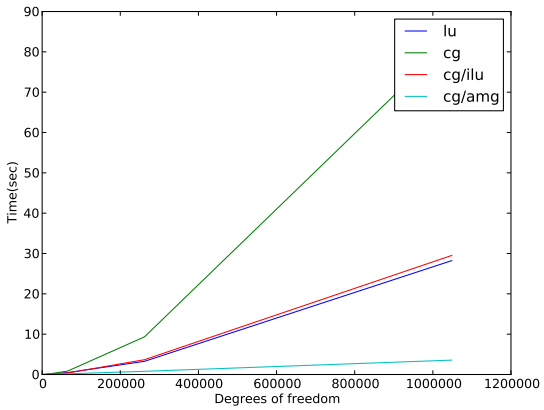

Figure 1.5: CPU time (in seconds) for solving a linear system of equation with N degrees of freedom (x-axis) for different solvers

- *Conjugate Gradient method,*
- *Conjugate Gradient method with an ILU preconditioner, and*
- *Conjugate Gradient method with an AMG preconditioner,*

for $N = 32^2, 64^2, 128^2, 256^2, 512^2, 1024^2$*, where* N *is the number of degrees of freedom.*
Figure 1.5 shows that there is a dramatic difference between the algorithms. In fact the Conjugate gradient (CG) with an AMG preconditioner is over 20 times faster then the slowest method, which is the CG solver without preconditioner. One might wonder why the LU solver is doing so well in this example when it costs $\mathcal{O}(N^2) - \mathcal{O}(N^3)$ *. However, if we increase the number of degrees of freedom, then the method would slow down compared to the other methods. The problem is then that it would require too much memory and the program would probably crash.*

Python code

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

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

def solving_time(A,b, solver):
  U = Function(V)
  t0 = time.time()
  if len(solver) == 2:
    solve(A, U.vector(), b, solver[0], solver[1]);
  else:
    solve(A, U.vector(), b, solver[0]);
  t1 = time.time()
  return t1-t0
```

```
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

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

  t2 = solving_time(A,b, ["lu"])
  print "Time for lu ", t2
  lu_time.append(t2)

  t2 = solving_time(A, b, ["cg"])
  print "Time for cg ", t2
  cg_time.append(t2)

  t2 = solving_time(A, b, ["cg", "ilu"])
  print "Time for cg/ilu ", t2
  cgilu_time.append(t2)

  t2 = solving_time(A, b, ["cg", "amg"])
  print "Time for cg/amg ", t2
  cgamg_time.append(t2)


import pylab

pylab.plot(Ns, lu_time)
pylab.plot(Ns, cg_time)
pylab.plot(Ns, cgilu_time)
pylab.plot(Ns, cgamg_time)
pylab.xlabel('Unknowns')
pylab.ylabel('Time(sec)')
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.savefig('tmp_cpu.pdf')
pylab.show()
```

Exercise 1.1. *Implement the Richardson iteration of the 1D Poisson problem with homogeneous Dirichlet conditions, using finite differences and a multigrid preconditioner.*

Exercise 1.2. *Estimate the convergence factor for the Jacobi and Gauss-Seidel iteration for the 1D Poisson problem from Example 1.3.*

Exercise 1.3. *Test CG method without preconditioer, with ILU preconditioner and with AMG preconditioner for the Poisson problem in 1D and 2D with homogeneous Dirichlet conditions, with respect to different mesh resolutions. Do some of the iterations suggest spectral equivalence?*

Exercise 1.4. *Test CG, BiCGStab, GMRES with ILU, AMG, and Jacobi preconditioning for*

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

Where Ω is the unit square, $v = c\,sin(7x)$, 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 homogeneous Dirichlet conditions.

Exercise 1.5. *The following code snippet shows the assembly of the matrix and preconditioner for a Stokes problem:*

Python code

```
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 1.3:

Python code

```
# 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
```