Homework Assignment 11

Even though the lion share of the computational expense when solving a PDE is the solution of the discrete linear system $Ax = b$, we have never spent a lot of thought on how to solve these systems. So far, we have simply used some high-level commands like the backslash in GNU Octave / MATLAB and `solve` in FEniCS. These commands would first run some initial tests to try and detect certain structure in the matrix, and then they choose a suitable method.

This can be done more efficiently. Since we already have a lot of knowledge about properties of the matrix A, we can select the best possible solver ourselves so that no additional testing is necessary at runtime.

Therefore, you will now learn about numerical methods for solving linear systems that stem from discretisations of PDEs. To choose a method that is (i) guaranteed to converge and (ii) as efficient as possible, you will have to use all your knowledge about the matrix A.

Question 1 considers *direct solvers*, which are useful for linear systems of moderate size, e.g. discretised 1D or 2D PDEs. Question 2 deals with *iterative solvers*, which are needed for very large systems, e.g. from 3D PDE problems.

Question 1 | 2 marks Read about LU factorisation (aka Gaussian elimination) and Cholesky factorisation.

(a) Which of these direct solvers is most appropriate for solving the two linear systems

$$\left(M^h + (\theta\Delta tc)^2 K^h\right)\vec{u}_+^h = M^h\left(\vec{u}_\circ^h + \Delta t\vec{v}_\circ^h\right) - \left(\theta(1-\theta)(\Delta tc)^2\right)K^h\vec{u}_\circ^h \tag{1a}$$

$$M^h\vec{v}_+^h = M^h\vec{v}_\circ^h - \Delta tc^2K^h\left(\theta\vec{u}_+^h + (1-\theta)\vec{u}_\circ^h\right) \tag{1b}$$

in the finite-element discretisation of the wave equation (cf Assignment 10) and why?

Solution. LU factorisation is a general matrix factorisation algorithm which works for any matrix A. Cholesky factorisation is a special case of LU factorisation for the case of symmetric positive definite matrices A, factorising A into LL^T instead of the more general LU, and is approximately twice as efficient when applicable.

The matrices M^h and K^h are both sparse, symmetric, positive definite matrices when they arise from the discretisation of the homogeneous wave equation. Then, the matrix $M^h + (\theta\Delta tc)^2 K^h$ is also symmetric positive definite, and so the matrix inversion in both equations 1a and 1b can be computed using the more efficient Cholesky factorisation method.

□

(b) Make a copy of `hw10.py` (you may use the program from the model answers). The new script should integrate the wave equation as in Assignment 10, but it should solve the linear systems with the method you selected in part (a).

Hint: You can find some useful FEniCS commands on the enclosed cheat sheet. Create a solver object for (1a) and another solver object for (1b). Your code should run approximately three times faster than in Assignment 10.

Solution. See Appendix A for the modified `hw10.py` code. □

Question 2 | 3 marks There are two main classes of iterative solvers: KRYLOV subspace methods and multigrid methods. We will look at KRYLOV subspace methods here.

Read about the conjugate gradient method (CG), the minimal residual method (MINRES) and the generalised minimal residual method (GMRES).

(a) Which of these iterative solvers is most appropriate for the linear systems in (1) and why?

Solution. The basic requirements of the listed KRYLOV subspace solvers are:

- CG: matrix must be symmetric positive definite
- MINRES: matrix must be symmetric
- GMRES: no requirements on the matrix; solver is fully generic

These methods are, naturally, also listed in decreasing order of efficiency.

Now, since the matrices M^h and K^h are both symmetric positive definite matrices, and therefore so is the sum $M^h + (\theta \Delta t c)^2 K^h$, the matrix inversion in both equations 1a and 1b should be solved using the conjugate gradient method. □

(b) Modify your FEniCS script from Question 1 to now solve the linear systems with the iterative method you selected in part (a).

Hint: Comment out the lines where you defined the direct solver objects. Create two iterative solver objects instead.

Solution. See Appendix A for the modified `hw10.py` code.

Interestingly, when using the iterative solver, although the forward Euler solution diverges at the same time as for the direct solver, FENICS throws an error instead of continuing to propagate the infinity/NaN values.

In my opinion, this actually seems like an advantage, as it prevents your solution from diverging silently. The direct solver method returned a solution without complaining, which depending on how the solution is used next, could be problematic. □

(c) Iterative methods typically converge significantly faster if they are applied to a preconditioned problem: instead of

$$Ax = b, \tag{2}$$

one solves the mathematically equivalent, but numerically advantageous problem

$$P^{-1}Ax = P^{-1}b. \tag{3}$$

The invertible matrix P, the so-called preconditioner, should on the one hand approximate A as closely as possible, but on the other hand it should be easier to invert than A. Such preconditioners are designed based on the specific properties of the linear system or the original PDE.

Note that if $P \approx A$, then $P^{-1}A \approx \mathrm{id}$. This is what makes (3) more amenable to iterative solvers than (2).

Read about diagonal aka JACOBI preconditioning and incomplete CHOLESKY factorisation. Can you think of an even better preconditioner specifically for the mass matrix M^h than these two generic preconditioners?

Solution. The basic properties of the listed preconditioners are:

- JACOBI: The preconditioner P is given simply by the main diagonal of the matrix A, and therefore P^{-1} is extremely easy to compute. This method only requires that the diagonal entries of A are non-zero.
- INCOMPLETE CHOLESKY: The preconditioner P is given by an efficient to compute sparse approximation of the CHOLESKY factorisation, and similarly requires that the matrix A is symmetric positive definite. This P is a sparsely constructed LL^T factorisation, and therefore is inexpensive to invert.

Now, both of these generic preconditioners could be reasonably efficient for inverting the sparse symmetric positive definite mass matrix M^h. A more efficient preconditioner, however, would be the mass lumping matrix $\tilde{M}^h$, which is the diagonal matrix with entries $\tilde{M}^h_{ii} = \sum_j M^h_{ij}$. This matrix would be a more efficient preconditioner because not only is it an easy to invert diagonal matrix (with non-zero diagonal entries), but mass lumping only incurs a $\mathcal{O}(h^2)$ error in the first place and so we should expect $(\tilde{M}^h)^{-1}M^h \approx \mathrm{id}$ to be a good approximation.

□

Your Oral Presentation You can find a worksheet attached to the presentation assignment on Canvas which you may want to use to prepare for your talk. Also familiarise yourself with the marking criteria which can be found on the same page to make sure you will cover everything that is needed.

You do not necessarily have to include final results in your talk. If you already have results to share, this is only to your advantage as it allows me to give you some feedback before you submit your written work where they will be graded.

Come up with an interesting title and write a succinct mini-abstract of at most three short sentences. This will also be made available to the public and should thus assume no specialised knowledge, but arouse interest for your talk. Please use the presentation assignment on Canvas to submit your title and your mini-abstract.

Your Learning Progress What is the one most important thing that you have learnt from this assignment?

How to use iterative solvers in FENICS! Although, I have already played around with this for my project (as I cannot use direct solvers for the large 3D system I am solving).

Any new discoveries or achievements towards the objectives of your course project?

Yes, I've discovered that I need to more carefully choose an iteratively solver for the PDE I am solving! Hopefully it will speed things up a little bit.

What is the most substantial new insight that you have gained from this course this week? Any *aha moment*?

Definitely reading up on proper definitions of the various iterative solvers. I have known about CG and have heard of the others from the supplemental notes, but I haven't ever really had to consider anything other than (preconditioned-)conjugate gradient, so this was good experience in that regard.

Appendix A

hw11.py

```
# coding=utf-8
"""
FEniCS program: Solution of the wave equation with homogeneous Dirichlet (reflective)
boundary conditions.
"""

from __future__ import print_function
from fenics import *
from mshr import *
import time
import csv

def wave_eqn(THETA = 0.5, foldname = 'cn'):
    # Create a mesh on the unit disk
    disk = Circle(Point(0., 0.), 1.)
    mesh = generate_mesh(disk, 100) # h   1/50

    # Function space and boundary condition
    V = FunctionSpace(mesh, 'P', 1)

    def boundary(x, on_boundary):
        return on_boundary

    bc = DirichletBC(V, Constant(0.), boundary)

    # Problem data
    t0 = 0. # initial time
    T = 5. # final time
    t = t0 # current time
    c = Constant(1.) # propagation speed
    c2 = c**2 # square speed, for convenience

    # initial displacement and velocity
    u0 = interpolate(Expression("pow(x[0],2)+pow(x[1],2) < 1.0/16.0 ? "
        "pow(1.-16.*(pow(x[0],2)+pow(x[1],2)),2) : 0.0", degree=1), V)
    v0 = interpolate(Constant(0.0), V)

    # Parameters of the time-stepping scheme
    tsteps = 500 # number of time steps
    dt = T/tsteps # time step size
    dt2 = dt**2 # square time step, for convenience
    theta = Constant(THETA) # degree of implicitness (fenics constant object)

    # Define the variational problem
    w = TrialFunction(V) # w = u in the 1st equation and w = v in the 2nd equation
    z = TestFunction(V)
    B1 = (w*z + (theta*dt*c)**2 * dot(grad(w),grad(z)))*dx # LHS of the 1st equation
    B2 = w*z*dx # LHS of the 2nd equation

    # Assemble matrices
    B1 = assemble(B1)
    B2 = assemble(B2)
    bc.apply(B1)
    bc.apply(B2)

    # Create direct solver objects (Cholesky) for the linear systems B1 and B2
    # B1_solver = LUSolver(B1)
    # B1_solver.parameters.symmetric = True
    # B1_solver.parameters.reuse_factorization = True
    #
    # B2_solver = LUSolver(B2)
    # B2_solver.parameters.symmetric = True
    # B2_solver.parameters.reuse_factorization = True

    # Create iterative solver objects (incomplete cholesky preconditioned
```

```
    # conjugate gradient) for the linear systems B1 and B2
    B1_solver = KrylovSolver(B1,'cg','icc')
    B1_solver.parameters.nonzero_initial_guess = True

    B2_solver = KrylovSolver(B2,'cg','icc')
    B2_solver.parameters.nonzero_initial_guess = True

    # Set initial data
    u = Function(V, name='Displacement')
    v = Function(V, name='Velocity')
    u.assign(u0)
    v.assign(v0)
    T = assemble(0.5*v*v*dx)
    V = assemble(0.5*c2*dot(grad(u),grad(u))*dx)
    E = T + V

    # Write initial data to file
    # displacement = File('wave/' + foldname + '/u.pvd')
    # displacement << (u, t)
    energy = csv.writer(open('wave/' + foldname + '/energy.csv', 'w'))
    energy.writerow([t] + [E] + [T] + [V])

    # Time stepping
    for k in range(tsteps):

        # Current time
        t = t0 + (k+1)*dt
        print('Step = ', k+1, '/', tsteps , 'Time =', t)

        # System for the displacement
        L1 = (u0+dt*v0)*z*dx
        if abs(THETA - 0.0) > DOLFIN_EPS and abs(THETA - 1.0) > DOLFIN_EPS:
            L1 -= theta*(1.0-theta)*dt2*c2 * dot(grad(u0),grad(z))*dx

        # Apply boundary conditions & solve system using the solver object
        L1 = assemble(L1)
        bc.apply(L1)
        B1_solver.solve(u.vector(), L1)

        # System for the velocity
        L2 = v0*z*dx
        if abs(THETA - 0.0) > DOLFIN_EPS:
            L2 -= theta*dt*c2 * dot(grad(u),grad(z))*dx
        if abs(THETA - 1.0) > DOLFIN_EPS:
            L2 -= (1.0-theta)*dt*c2 * dot(grad(u0),grad(z))*dx

        # Apply boundary conditions & solve system using the solver object
        L2 = assemble(L2)
        bc.apply(L2)
        B2_solver.solve(v.vector(), L2)

        # Compute energy
        T = assemble(0.5*v*v*dx)
        V = assemble(0.5*c2*dot(grad(u),grad(u))*dx)
        E = T + V

        # Write data to file
        # displacement << (u, t) #don't need the solution
        energy.writerow([t] + [E] + [T] + [V])

        # Update
        u0.assign(u)
        v0.assign(v)

if __name__ == "__main__":
    wave_eqn(THETA = 0.5, foldname = 'cn')
    wave_eqn(THETA = 1.0, foldname = 'be')
    wave_eqn(THETA = 0.0, foldname = 'fe')
```