Second Year Engineering Mathematics Laboratory
Hilary Term 2000

Ian Reid, 15/12/1999

Exercise 2: Iterative solution of simultaneous equations

The purpose of this exercise is to explore the use of iterative algorithms for solving simultaneous equations — the "Jacobi" algorithm and close relatives. This is done in the context of a (simplified) engineering problem: modelling an elastic surface under load.

1 Preparation

Read the relevant parts of the lecture notes.

A. Simulating a springy surface

In this exercise we try to model a (one-dimensional) membrane on an elastic base. Consider Figure 1(a) in which the membrane is modelled as a set of n evenly spaced points held together by a tensile force of one unit in the horizontal direction, affected by a load b_j at the jth point, and with each point experiencing an elastic restoring force proportional to its deflection from zero. This might be a pausible model for an elastic deformation of a road surface as a vehicle moves over it.

Show by resolving forces vertically at each point that:

$$x_{j-1} - 2x_j + x_{j+1} = b_j + \lambda x_j, \;\; 1 \leq j \leq m, \tag{1}$$

(where $\lambda \geq 0$ is the constant that governs the relative strength of a restoring force). Exerting pressure at one point ($b_j = 0$ except for one value of j) causes an indentation of the surface as in figure 1(b).

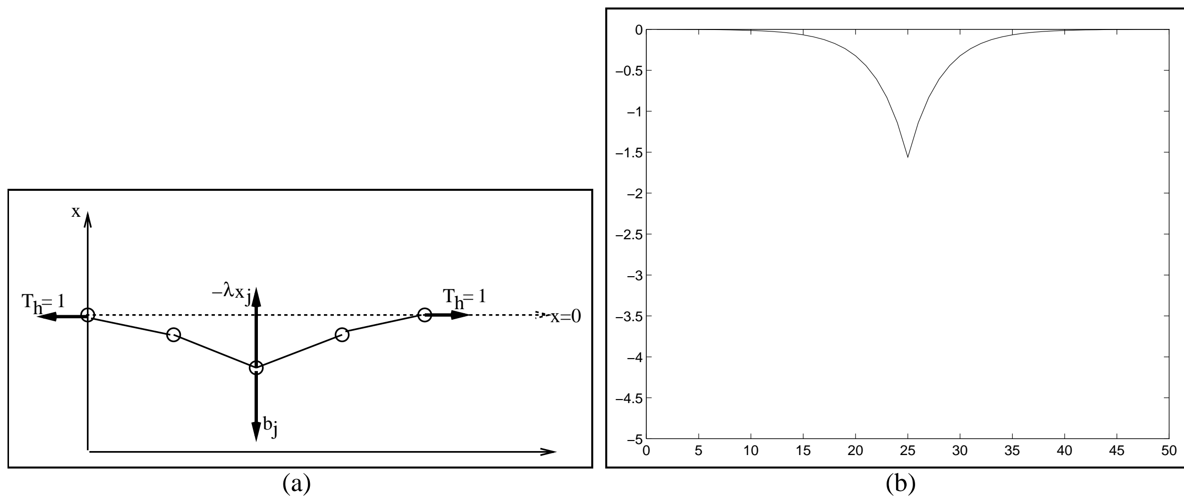
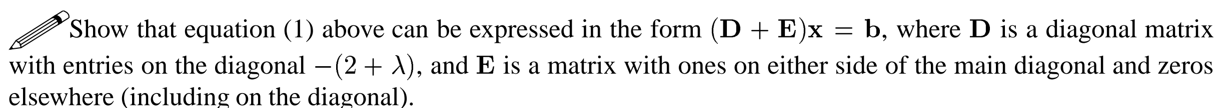
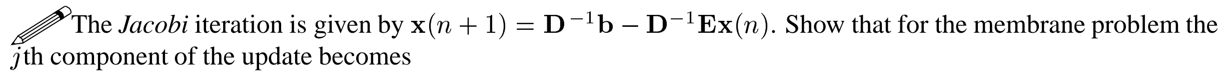


Figure 1: *Computing the shape of a membrane on an elastic base, deformed by a downward force acting at a point.*


Show that equation (1) above can be expressed in the form $(\mathbf{D} + \mathbf{E})\mathbf{x} = \mathbf{b}$, where $\mathbf{D}$ is a diagonal matrix with entries on the diagonal $-(2 + \lambda)$, and $\mathbf{E}$ is a matrix with ones on either side of the main diagonal and zeros elsewhere (including on the diagonal).

B. The Jacobi algorithm


The *Jacobi* iteration is given by $\mathbf{x}(n+1) = \mathbf{D}^{-1}\mathbf{b} - \mathbf{D}^{-1}\mathbf{E}\mathbf{x}(n)$. Show that for the membrane problem the jth component of the update becomes

$$x_j(n+1) = -(b_j - x_{j-1}(n) - x_{j+1}(n))/(2+\lambda)$$

C. The Gauss-Seidel algorithm

The Gauss-Seidel iteration for solution of $\mathbf{A}\mathbf{x} = \mathbf{b}$ is derived by expressing

$$\mathbf{A} = \mathbf{D} + \mathbf{L} + \mathbf{U}$$

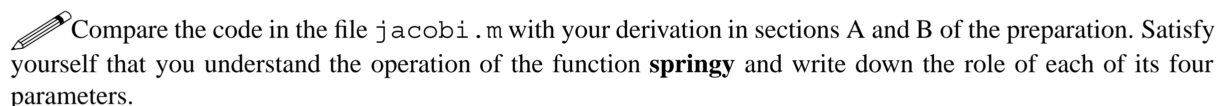
where $\mathbf{D}$ is diagonal, $\mathbf{L}$ is strictly lower triangular and $\mathbf{U}$ is strictly upper triangular. The update is then given by $\mathbf{x}(n+1) = \mathbf{D}^{-1}\mathbf{b} - \mathbf{D}^{-1}\mathbf{L}\mathbf{x}(n+1) - \mathbf{D}^{-1}\mathbf{U}\mathbf{x}(n)$. Explain the presence of the term $\mathbf{x}(n+1)$ on the right-hand side of the equation, and show that for the membrane problem the Gauss-Seidel update for the jth component is given by:

$$x_j(n+1) = -(b_j - x_{j-1}(n+1) - x_{j+1}(n))/(2+\lambda)$$

2 Lab Work

A. Simulating a springy surface

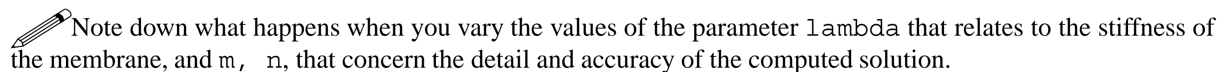
Copy the files `springy.m` and `jacobi.m` from `/packages/demo/comp/lab2` to a directory called `lab2` in your own area.


Compare the code in the file `jacobi.m` with your derivation in sections A and B of the preparation. Satisfy yourself that you understand the operation of the function **springy** and write down the role of each of its four parameters.

Try it out by typing

```
springy(0.1, 50, 10, 3)
```

First the program computes an accurate solution for the deformed surface under a point load, using many iterations. (The use of many iterations, in this way, is of course impractical in real applications, but is done here for tutorial purposes only.) Then (press RET) it computes some intermediate solutions, to show how the Jacobi algorithm converges.


Note down what happens when you vary the values of the parameter `lambda` that relates to the stiffness of the membrane, and `m, n`, that concern the detail and accuracy of the computed solution.

B. Using norms to look at convergence

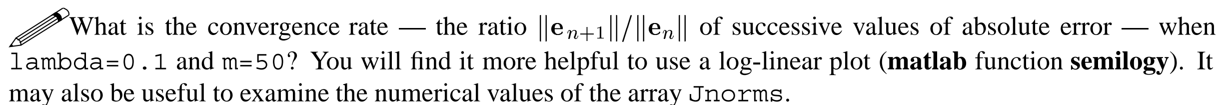
Now you use the "absolute error" $\mathbf{e}_n = \mathbf{x}_n - \mathbf{s}$ (where $\mathbf{s}$ is the true solution vector of the linear equations) to investigate convergence experimentally. The 2-norm $\|\ldots\|_2$, denoted here just $\|\ldots\|$ for brevity, is used throughout. It is computed by the **matlab** function `norm` (see `help norm` and look under "vectors"). Ideally, you would like your iterative algorithm to terminate when $\|\mathbf{e}_n\|$ becomes smaller than some pre-specified value.

Write a new function **springynorm** similar to **springy** which returns two arrays `iter` and `e`, each of length `n+1`. The array `iter` should contain the iteration number, and the array `e` the absolute error at each iteration. Some sample code is given below.

```
function [iter,e] = springynorm(lambda, m, n)
  % lambda:  relative elasticity
  % m: number of nodes
  % n: number of iterations

b = (1:m+2)*0;  % array of 0's - downward forces
b(m/2)=1;       % with 2 1's in middle - vehicle wheels
x = (1:m+2)*0;  % array of 0's - initial heights

% iterate 200 times to get "true" solution
xx = jacobi(lambda, x, b, 200);

iter = [0:n];           % iteration number 0,1,2,...n
e(1) = norm(x-xx);      % initial error

for k = 2:n+1
  x = jacobi(lambda, x, b, 1);
  e(k) = norm(x-xx);
end
```

Run your new function for 30 iterations and plot the results:

```
>> [iter, Jnorms] = springynorm(0.1, 50, 30);
>> plot(iter, Jnorms);
```


What is the convergence rate — the ratio $\|\mathbf{e}_{n+1}\|/\|\mathbf{e}_n\|$ of successive values of absolute error — when `lambda=0.1` and `m=50`? You will find it more helpful to use a log-linear plot (**matlab** function **semilogy**). It may also be useful to examine the numerical values of the array `Jnorms`.

C. Gauss-Seidel

Write a replacement for the function **jacobi**, called **GS**, that applies the Gauss-Seidel algorithm to the same problem of computing deformation under load. Use the function **jacobi** as a starting point and

- change the function name (obviously to **GS**);
- change the line `anew(j) = ...` (see preparation, section C);
- save the new function in a file called `GS.m`.

Alter your function **springynorm** to use **GS** in place of **jacobi**. A good way to do this is to "comment out" the relevant line in your function and replace it with the new line.

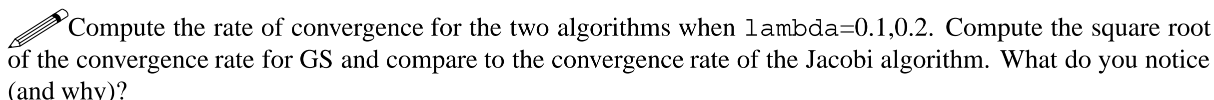
```
%    x = jacobi(lambda, x, b, 1);
     x = GS(lambda, x, b, 1);
```

That way you can easily switch between the two algorithms.

Using your program, create arrays `iter`, `Jnorms` and `GSnorms` such that you can plot the convergence of both the Jacobi and Gauss-Seidel algorithms, for parameter values `lambda=0.1` and `m=50`, over 30 iterations.

Generate (and record in your log-book) the plots, all on one set of log-linear axes as follows.

```
semilogy(iter, Jnorms, 'r', iter, GSnorms, 'g')
```


Compute the rate of convergence for the two algorithms when `lambda`=0.1,0.2. Compute the square root of the convergence rate for GS and compare to the convergence rate of the Jacobi algorithm. What do you notice (and why)?

D. Automatic termination of iteration (optional)

By monitoring the rate at which the *relative error*

$$\delta(n) = \mathbf{x}(n+1) - \mathbf{x}(n)$$

decreases, it should be possible to terminate the Gauss-Seidel algorithm automatically when the solution is accurate to within a chosen precision p. Clearly this is desirable — no user of computer code wants to have to say how many iterations should be used; instead, the required precision is a sensible thing to be asked to specify. The steps in deciding when to terminate are:

1. Estimate the asymptotic convergence ratio $\alpha = \lim_{n\to\infty} \alpha_n$ where $\alpha_n = ||\delta_{n+1}||/||\delta_n||$.
2. Use α to estimate the norm of the absolute error $\mathbf{e}_n$. To do this, use the fact that

$$-\mathbf{e}_n = \delta_n + \delta_{n+1} + \dots$$

   so that, by the triangle inequality for norms,

$$||\mathbf{e}_n|| \leq ||\delta_n|| + ||\delta_{n+1}|| + \dots = ||\delta_n||(1 + \alpha_n + \alpha_n\alpha_{n+1} + \dots)$$

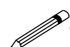
   and approximating $\alpha_n \approx \alpha$,

$$||\mathbf{e}_n|| \leq ||\delta_n||(1 + \alpha + \alpha^2 + \dots).$$

3. Terminate the algorithm when the average error $||\mathbf{e}_n||/\sqrt{m}$ is less than p (*Why the factor of* $\sqrt{m}$*?*)

Now write a new function **GSauto** that terminates automatically. A skeleton is given below:

```
function anew = GSauto(lambda,b,p);

  m = length(b) - 2;     % no of nodes, excl. bdys
  a = (1:m+2)*0;         % starting solution all zeros
  anew = a;
  alpha = 0;
  err = 1+p;             % ensure first time through loop happens

  for j = 2:m+1          % interior of array, missing boundaries
    anew(j) = (anew(j-1) + a(j+1) - b(j))/(2 + lambda);
  end
  delta = ???;
  a = anew;

  while err > p,
    for j = 2:m+1          % interior of array, missing boundaries
      anew(j) = (anew(j-1) + a(j+1) - b(j))/(2 + lambda);
    end
    deltanew = ???;
    alpha = ???;
    err = ???;

    a = anew;
    delta = deltanew;
  end
```

Call your function appropriately and plot the results.