CS 740, Fall 1996
Homework Assignment 6

Assigned: Nov. 21
Due: Thurs., Dec. 5, 1:00PM

1 Logistics

You may work in a group of up to 3 people in solving the problems for this assignment. You should turn in a single report for your entire group, identifying all of the group members.

Any clarifications and revisions to the assignment will be posted on Web page `assigns.html` in the class WWW directory.

You will hand in a hard copy document for this assignment. Formatted text is preferred to hand written.

Note: The first part of the following description gives background for the problem. The description of what you need to do for the assignment starts on page 9.

Simulating air flow over an airplane wing

This assignment involves simulating air flow over an airplane wing.[1] Air can be modeled as a fluid and the simplest model of fluid flow is Laplace's equation

$$\nabla^2 \phi = 0 \qquad (i.e.\ \frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = 0) \tag{1}$$

where the *velocity potential* $\phi(x, y)$ is related to the flow velocity $\vec{v}$ (air speed) by

$$\vec{v} = \nabla \phi \qquad (i.e.\ v_x = \frac{\partial \phi}{\partial x},\ \ v_y = \frac{\partial \phi}{\partial y}) \tag{2}$$

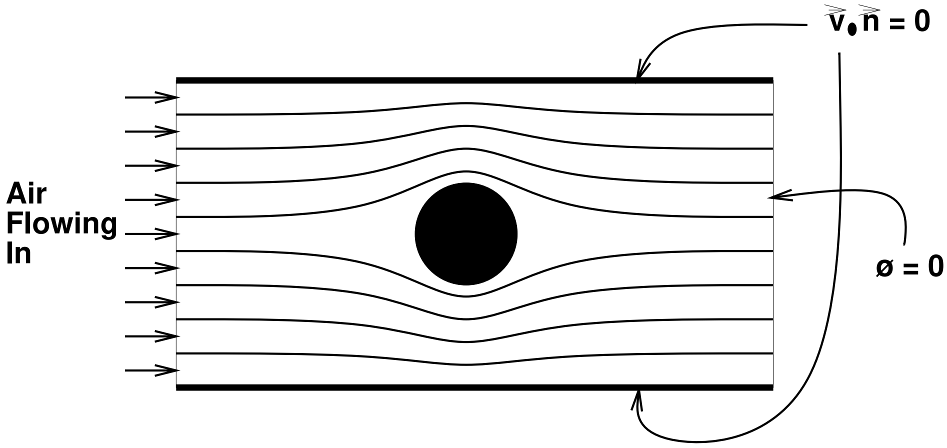
Note that if we solve for ϕ then we can easily find the flow velocity. In fact, the function ϕ has no physical interpretation other than being used indirectly for finding the flow velocities.

This equation makes several simplifying assumptions about fluid flow. In particular it assumes that the fluid is incompressible (i.e. the pressure is constant), that there is no viscosity (i.e. no friction) and that the fluid cannot rotate to form vortices. These assumptions are reasonable for simulating air flow at low speeds (no turbulence) and if you don't care about drag.

[1] This assignment was prepared by Professor Guy Blelloch with help from Professor Omar Ghattas from the Civil Engineering department.

The goal of this homework is to simulate fluid flow under these assumptions over an object, in particular an airplane wing. To do this we need to solve Laplace's equation over a region. We also need to set the boundary conditions to specify the velocities at a set of fixed boundaries. This will effectively simulate a wind tunnel.

Consider the following picture:

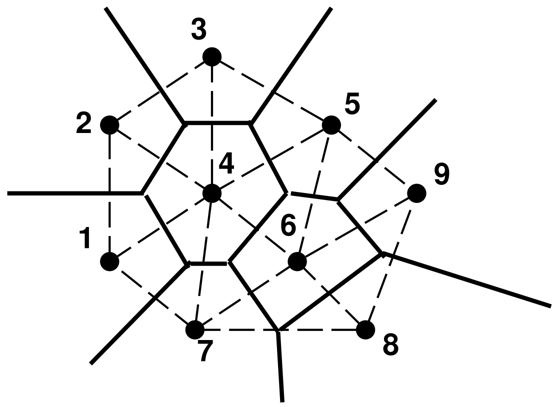


To simulate airflow throughout the region we will use the following boundary conditions. First we assume that the object itself is impenetrable so that the air flow at the surface of the wing has no component perpendicular to the surface ($\vec{v} \cdot \vec{n} = 0$ where $\vec{n}$ is the unit vector normal to the surface). Second we will assume that air is blowing in from the left, and assume that the component of flow entering on the left is constant. We assume that there is no flow through the top or bottom (as if it were a wind tunnel). Finally we will set $\phi = 0$ at one point. We do this since without specifying the value anywhere, the system is under-determined.

Discrete formulation of differential equations The first step in computer solutions of differential equations (such as the flow equation we are considering) is approximating continuous space with a set of discrete points (or cells). We then use this set of points to approximate the real solution. In particular we would like to know the value of our function ϕ at each of the points so that we can then derive the flow. The finer we make the points (closer together) the more accurate our simulations will be. For some simulations it is adequate to put the points on a regular grid such that the distance between points is constant over the whole grid. In other simulations it is important to get more accuracy in certain areas, in particular in areas where the function of interest is changing more rapidly. In such simulations one places more points in places of more interest.

The data we are going to be using for this assignment (an airplane wing with flaps) has already been represented as a set of points. This set of points is much denser in regions where airflow is likely to have more interesting effects, such as at the front of the wing and between the back of the wing and the flaps.

Finite volume method for solving differential equations: The method we are going to use to solve our differential equation and thus the air-flow is what is called a finite-volume method. We view space as being broken down into a set of volumes each of which surrounds one of our points. In particular the volume associated with each point is the region of space that is closer to that point than to any other point. Breaking space into such cells is called the *Voronoi Diagram* of a set of points. In a Voronoi diagram we have exactly one cell per point. Here is an example of a set of points along with their Voronoi cells:



Note that each cell is a convex polygon (except for boundary points which are partial polygons). We will call the edges of the polygons *Voronoi edges*, and each one separates two points. If we draw the line between every pair of adjacent points (points that share a Voronoi edge) we get what is called the *Delaunay triangulation* of the points. The edges of the triangles will be called *Delaunay edges*. In our solution we will consider both the Voronoi edges and the Delaunay edges.

We now consider how these cells can be used to solve the fluid flow problem. Consider the total flow that enters a cell through its boundaries. Since we have assumed the fluid is incompressible, the net flow into each cell (sum of flow in and out) has to be zero. This condition can be written formally as:

$$\int_{\Gamma_i} \vec{v} \cdot \vec{n} \; ds = 0 \qquad (3)$$

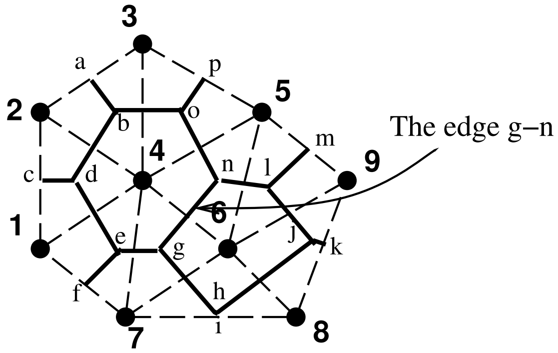
where Γ_i is the boundary of the i^{th} cell, $\vec{v}$ is our flow (the gradient of ϕ), and $\vec{n}$ is the vector normal to the surface. By the divergence theorem

$$\int_{\Gamma} \vec{v} \cdot \vec{n} \; ds = \int_{R} \nabla \cdot \vec{n} \; dv$$

this is the same as forcing Laplaces equation (equation 1) to hold over each volume.

The way we specify that the net flow into each cell is 0 is by writing a linear equation for each one. The variables in these equations are going to be the values of our function ϕ at the points (these are the variables we want to solve for). If we have n cells, we are going to write n equations. Since there are n unknowns we can simply solve the set of linear equations. We discuss how to solve these equations later but first discuss how to generate the equations.

Setting up the linear equations: To set up the equations lets consider a particular cell. The flow into the cell is the sum of the flow through its edges. We already mentioned that each of the cell's edges is associated with another point. Based on equation 3 we would like to determine the gradient of ϕ along each Voronoi edge in a direction perpendicular to the edge. This gradient can be determined to a good approximation by taking the difference in the value of ϕ at the two points associated with the edge and dividing this difference by the distance between the points (a gradient is simply a directional first derivative, i.e. a slope). Note that the line joining the two points is perpendicular to the edge. To approximate the flow across the Voronoi edge we then simply take this gradient and multiply it by the length of the edge. For example, consider the following cells:



The flow through the Voronoi edge g-n into cell 6 is $\frac{(\phi_4-\phi_6)\cdot l_{g-n}}{l_{4-6}}$, where l is the length of an edge. Now to generate our linear equation for the cell we simply add up all the flows into the cell and set this sum to zero. For the above example the equation for cell 6 would be

$$\begin{aligned} &\frac{(\phi_4-\phi_6)\cdot l_{g-n}}{l_{4-6}} + \frac{(\phi_7-\phi_6)\cdot l_{h-g}}{l_{7-6}} + \\ \frac{(\phi_8-\phi_6)\cdot l_{j-h}}{l_{8-6}} + \frac{(\phi_9-\phi_6)\cdot l_{l-j}}{l_{9-6}} + &\frac{(\phi_5-\phi_6)\cdot l_{n-l}}{l_{5-6}} \quad = \quad 0 \end{aligned}$$

We could write similar equations for each of the nine points (cells) in the diagram. This would give us a set of equations of the form

$$Ax = b$$

where the A would be a sparse matrix with the coefficients of our equations, the vector x would be the values of ϕ_i at each point, which we are trying to solve for, and the vector b would be all 0s.

The one thing we have not considered are the boundary conditions. Earlier we discussed how we assume that the flow through the surface of any object is 0. To deal with this we don't have to do anything special. For example, for cell 3 in the above diagram, if we say that the sum of the flows through edges $a-b$, $b-o$ and $o-p$ is 0 then we are implicitly assuming that the flow through the boundaries $2-3$ and $3-5$ is 0. Also as mentioned earlier we assume that the flow through the top and bottom of our space is 0. This can be dealt with in a similar way.

The only tricky boundary conditions we have to consider are the left and right hand sides of our space. To deal with them we set the flow through the boundary to a value that is proportional to the component of the boundary parallel to the y axis. For example, the equation for cell 2 would be:

$$\frac{(\phi_3 - \phi_2) \cdot l_{a-b}}{l_{3-2}} + \frac{(\phi_4 - \phi_2) \cdot l_{b-d}}{l_{4-2}} + \frac{(\phi_1 - \phi_2) \cdot l_{d-c}}{l_{1-2}} = b_2$$

where b_2 is the sum of the flow through edges $1-2$ and $2-3$.

The one last thing we must do is set one point i so that $\phi_i = 0$. We can do this with any point, but it is typical to pick one of the boundary points.

Solving linear equations: Once we have our set of equations $Ax = b$ we need to solve them. The matrix A is very sparse (typically only 6 nonzero entries per row), so we want to pick a method that works well with sparse matrices.

Solving linear equations: We now consider how to solve a linear system of the form $Ax = b$ for a sparse matrix A. There are two classes of methods for solving linear systems: *direct* and *iterative* methods. Gaussian elimination is a direct method, but in this assignment we are going to use an iterative method. This is because iterative methods work well with sparse matrices and are easy to parallelize. The basic idea of an iterative method is that we repeat a set of steps and each step generates a better approximation of the final answer. The improvement is made by trying to reduce the error caused by the guess. One of the simplest such techniques is Jacobi iteration. Each step in the Jacobi method picks a new set of variables based on the equation:

$$x_{\text{new}} = x + (b - Ax)$$

Let's consider an example—the equations:

$$\begin{aligned} 3x + y &= 9 \\ x + 2y &= 8 \end{aligned}$$

The matrix form for these equations is:

$$\begin{pmatrix} 3 & 1 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 9 \\ 8 \end{pmatrix}$$

Let's solve this linear system using Jacobi's method. It turns out that the convergence can be improved by scaling each row so that the the diagonals are 1. This gives:

$$\begin{pmatrix} 1 & 1/3 \\ 1/2 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 3 \\ 4 \end{pmatrix}$$

We start with the initial guess $x = y = 0$ —our initial vector is $(0, 0)$. We use $x_{\text{new}} = x + (b - Ax)$ to generate our new guess, which gives $(3, 4)$. We now use this again, which gives $(3, 4) + (3, 4) - (13/3, 11/2) = (5/3, 5/2)$. The next few iterations give:

$$(13/6, 19/6),\ (1.94, 2.92),\ (2.03, 3.03),\ (1.99, 2.99), \ldots.$$

This sequence is converging on the right answer, which is (2,3). The problem with Jacobi's method is that it is only guaranteed to converge if the matrix A is *diagonally dominant*. A matrix is diagonally dominant if the absolute value of the diagonal in each row is larger than the the sum of the absolute values of the off diagonals in the row (i.e. $|a[i,i]| > \sum_{j \neq i} |A[i,j]|, 0 \leq i < n$). For example, consider switching the two rows in the above matrix.

$$\begin{pmatrix} 1 & 2 \\ 3 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 8 \\ 9 \end{pmatrix}$$

The matrix is now not diagonally dominant. If we try Jacobi's method we get the sequence:

$$(8, 9),\ (-10, 6.6),\ (-5.2, 12),\ (-16, 10.6),\ (-13.1, 13.8),\ (-19.6, 12.9),\ \ldots$$

This is diverging from the correct answer, and as it turns out, will never converge.

Conjugate Gradient

To get much better convergence we need to use the conjugate gradient (CG) method. In this section we will assume the matrix A is symmetric. One way of thinking about solving a linear system is to pose it as a minimization problem. If we write the equation $r(x) = b - Ax$, the goal of a linear system solver is to determine the variables x such that $r(x)$ is zero everywhere. This can be solved by finding the minimum of the function $q(x) = (1/2)xAx - xb$ since $\nabla q(x) = r(x)$ and we know that when a function is at a minimum, the derivative in each dimension is 0. We can think of the function $q(x)$ as a contour map with the x as our dimensions and $q(x)$ as the altitudes. We want to find a minimum location on the contour map (the bottom of a valley). One way to do this is to start at a particular location and then keep taking steps downhill.

Jacobi's method basically always figures out the direction of the steepest slope and takes a step in that direction. The direction of the slope of $q(x)$ is simply $r(x)$, by definition. So therefore our equation

$$x_{\text{new}} = x + (b - Ax)$$

in Jacobi's method is exactly taking a step in the direction of the steepest downhill slope of $q(x)$. So why doesn't Jacobi's method always work? The problem is that we can overshoot and end up on the other side of the valley by taking a step that is too big. In the case that the matrix is not diagonally dominant we could in fact keep taking bigger steps and end up working our way higher and higher up the the walls of the valley, instead of towards the bottom.

Let's consider how to improve on this. In general we want to take a step of the form:

$$x_{new} = x + \alpha p$$

where α is a scalar step size and p is a direction vector. Based on this form, there are two motivations behind the CG (Conjugate Gradient) method.

1. Don't allow yourself to go back in the direction you came from (pick a p that does not point back to where you came from).
2. Take a step of the right size (pick α well).

The idea of the conjugate gradient (CG) method is therefore to pick a step in a direction that is perpendicular (conjugate) to all previous steps and to pick an α such that given the selected direction p it minimizes our function $q(x + \alpha p)$.

To select the direction vectors p on each step the CG method uses the rule

$$p_{next} = r_{new} + \frac{r_{new}^2 p}{r^2}$$

where p is the previous direction vector, $r = b - Ax$ (as previously defined), and $r_{new} = b - Ax_{new}$, where x_{new} is our new x. We pick and initial direction vector equal to b. It turns out that this method for picking direction vectors guarantees that each is conjugate (perpendicular relative to A) to all previous direction vectors even though it only needs to consider the last previous one.

Now that we have a direction vector p, to pick the scalar step size α we want to minimize $q(x + \alpha p)$ with respect to α:

$$\begin{array}{ll} \textbf{minimize}_\alpha & 1/2(x + \alpha p)A(x + \alpha p) - (x + \alpha p)b \\ \textbf{minimize}_\alpha & 1/2(xAx + 2\alpha pAx + \alpha^2 pAp) - xb - \alpha pb \end{array}$$

to solve this we can take the derivative with respect to α and set it to zero:

$$\begin{aligned} 0 &= 1/2(2pAx + 2\alpha pAp) - pb \\ &= pAx + \alpha pAp - pb \\ \alpha &= (pAx - pb)/(pAp) \\ &= pr/(pAp) \end{aligned}$$

The full algorithm is therefore:

$x = 0$
$r = b - Ax = b$
$p = b$

repeat until $r^2 < \epsilon$
$\alpha = \frac{pr}{pAp}$
$x_{new} = x + \alpha p$
$r_{new} = b - Ax_{new}$
$p = r_{new} + \frac{r_{new}^2}{r^2} p$
$x = x_{new}$
$r = r_{new}$

This routine requires 2 sparse-matrix vector multiplies, 4 vector dot products, and 3 vector adds per iteration. Typically the sparse-matrix vector multiplies are the most expensive steps. It turns out that the routine can be optimized to run with only a single matrix-vector multiply. This is based on the following equality

$$\begin{aligned} r_{new} &= b - Ax_{new} \\ &= b - A(x + \alpha p) \\ &= b - Ax - \alpha Ap \\ &= r - \alpha Ap \end{aligned}$$

This equality allows us to use the matrix-vector product Ap twice instead of having to do a second product Ax_{new}. This leads to the optimization.

$x = 0$
$r = b - Ax = b$
$p = b$
repeat until $r^2 < \epsilon$
$w = Ap$
$\alpha = \frac{pr}{pw}$
$x_{new} = x + \alpha p$
$r_{new} = r - \alpha w$
$p = r_{new} + \frac{r_{new}^2}{r^2} p$
$x = x_{new}$
$r = r_{new}$

With this modification each step of the conjugate gradient method is hardly any more expensive than a step of the Jacobi method. It is also possible to remove two of the 4 dot products.

What you need to do for the assignment

For this homework you should implement the conjugate gradient method in NESL and use the NESL interpreter to try it out. Refer to the SCANDAL project WWW page (`http://www.cs.cmu.edu/ scandal`) for code examples, instructions on running the interpreter, and documentation.

We give you the following data:

1. The x, y location of a set of points, each of which corresponds to one cell.
2. For each cell (point) we give the following information for each of its boundary edges.
   (a) The index of the point on the other side of the edge.
   (b) The length of the boundary edge
   (c) The distance from the point to the point on the other side of the edge.

   This is given as a nested sequence of the form

```
[[(j,length,distance),(j,length,distance),...]     %first point %
 [(j,length,distance),(j,length,distance),...]     %second point %
 ...
]
```

3. We give the vector b. This is 0 for all points except for the outside boundary points.
4. We give a node number for which ϕ should be set to 0.

To read in the data you can use the function `read_data`, which you can find in the file `/afs/cs/academic/class/15740-f96/public/asst/asst6/asst6.nesl`. This will return the quadruple `(points, cell_info, b, num)`.

We also give you some routines for plotting the results and for sparse vector matrix multiply . In particular we give you the following functions, which can be found in the same file:

```
wininfo = create_grid_window(points,cell_info)

_ = draw_grid_values(x,wininfo)

_ = wait_until_mouse_click(wininfo)

vect = sparse_mvmult(A,b)
```

The _ indicates that those functions do not return anything.

Your job is to write a linear system's solver to solve the airflow problem over the data we give you. This will require generating the appropriate matrix A from the data and implementing the Conjugate Gradient iterative method for solving the equation $Ax = b$. Your final program, which should be called `solve_wing`, should load the data, generate the matrix A, create a window (using `create_grid_window`), iterate through the Conjugate Gradient method and replot the values of x every 50 iterations (using `draw_grid_values`). Your program should go through 6 cycles of 50 iterations (300 iterations total). Your initial guess for x should be all 0s.

Since NESL does not have loops, you will need to use recursion (NESL implements recursion efficiently). If you get an `Unable to allocate color` message, this means you are using too much of your color map. You might have to kill some windows (*e.g.* Netscape is a color hog).

You can run a demo of what your program should do by running `cg_airflow`, or for a tutorial, run `airflowdemo`.