Finite Element Treatment of the Navier Stokes Equations: Part V

John Burkardt
https://people.sc.fsu.edu/∼jburkardt/fem_2005/fem_ns5.pdf

16 August 2021

1 Introduction

In this section, we consider sparse storage for the Jacobian matrix that arises in our treatment of the Navier Stokes equations. Recall that, for both the time dependent and steady state cases, it is necessary to apply Newton's method in order to find a satisfactory approximate solution of the finite element equations.

This Jacobian matrix is naturally sparse, and so it is of great advantage to determine a sparse storage format suitable for this setting.

Given that a sparse format is to be used, it will also be necessary to determine an algorithm, whether direct or iterative, that can be reliably applied to solve the resulting linear system. If a direct method is used, then presumably it will require that the sparse storage format include additional storage for fill-in entries.

2 The Problem Setting

We assume that we are solving the incompressible steady Navier-Stokes equations for velocity and pressure over an arbitrary connected 2D domain. We assume that a set of nodes have been chosen, and a corresponding set of 6-node triangular elements constructed, so that the entire region of interest is covered by the elements.

However, we assume that the mesh is "irregular", in the sense that, because of geometric constraints or other reasons, the size, shape, and connectivity of the elements varies somewhat. We mean to state that the triangulation at hand is *not*, for instance, the regular tessellation of a simple square - in such special cases, it is possible to write down beforehand the location of every node, the indices that make up each element, and even the bandwidth of the Jacobian matrix. Compare the images of the simple "square" region against that of the "hex" region.

We also assume that each node has associated with it a set of variables, corresponding to unknown coefficients in the finite element representation of a flow solution. We assume that every node has an unknown velocity variable, while only nodes corresponding to triangle vertices have an unknown pressure variable. (Nodes along the boundary are also assumed to satisfy this condition; boundary conditions are treated in a way that formally creates corresponding unknowns.) We assume that all the unknowns have been assigned unique indices.

We assume that, given the index of any node, we can look up its physical location, the number of unknowns associated with the node, and the indices of those unknowns.

We assume that, given any element, we can look up the indices of the six nodes that make up that element, as well as the order in which these nodes occur in the element.

3 Node and Variable Indices

In the finite element method, a scalar function $f(x, y)$ is represented by constructing a set of basis functions $\psi_i(x, y)$, with local support, associated with the $NodeNum$ mesh nodes. The appropriate coefficients c_i are then determined to yield a representation of the function:

$$f(x, y) = \sum_{i=1}^{NodeNum} c_i \quad \psi_i(x, y). \tag{1}$$

For Navier Stokes calculations using the Taylor Hood element, there are several complications:

- the velocity is a vector function
- the pressure is approximated using only part of the mesh, namely, the nodes associated with vertices
- depending on the treatment of the boundary conditions and other constraints, the representation of a function could require a more complex form in certain regions.

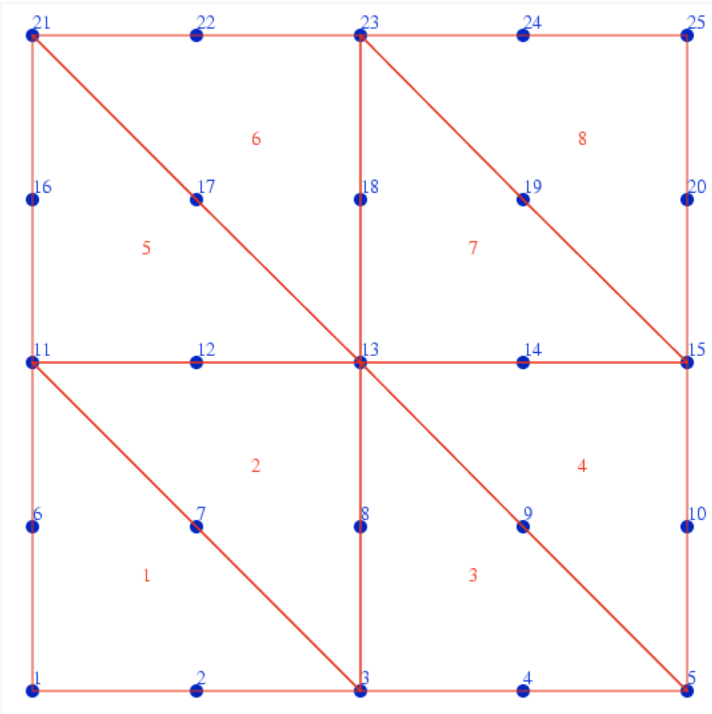

Figure 1: The “square” region with an unstructured 6-node triangular mesh.

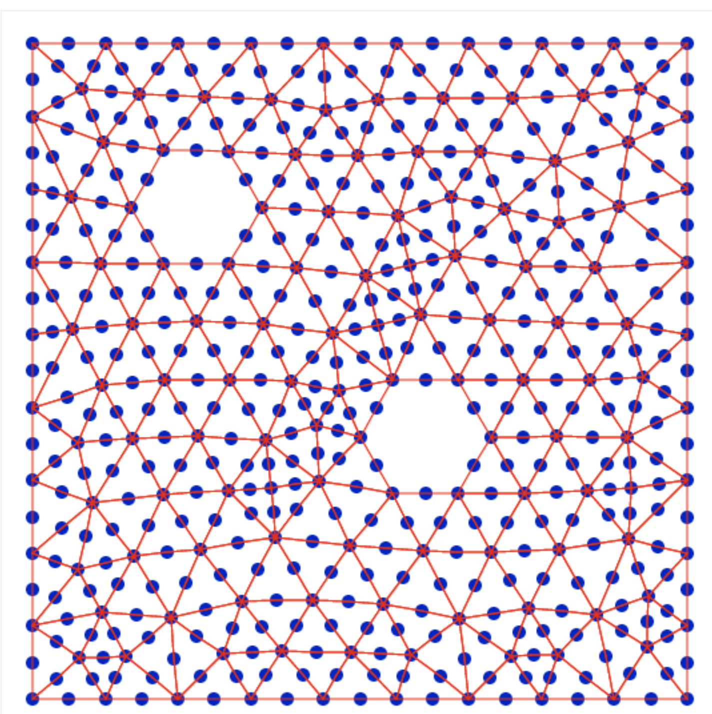

Figure 2: The “hex” region with an unstructured 6-node triangular mesh.

For our particular applications:

- we treat the velocity as two component scalar functions U and V, thus, in a sense, we have two copies of the basis function ψ_i.
- we generate a separate set ϕ_i of basis functions for the pressures P, using only the element vertices
- we are careful to treat boundary conditions and constraints in such a way that we preserve the simple relationship between nodes and coefficients.

Let us suppose, then, that we have numbered the nodes. We now propose to number the variables, that is, the undetermined coefficients associated with the two velocity components and the pressure. A natural procedure is to consider each node in order of its index, and to generate new variable indices for the two or three coefficients associated with that node. Unless the mesh is very regular, the resulting set of indices is very difficult to set up in advance, or without going through such a step-by-step procedure.

For our simple square region. the resulting variable assignment is

Node	U	V	P
1	1	2	3
2	4	5	...
3	6	7	8
4	9	10	...
5	11	12	13
6	14	15	...
7	16	17	...
8	18	19	...
9	20	21	...
10	22	23	...
11	24	25	26
12	27	28	...
13	29	30	31
14	32	33	...
15	34	35	36
16	37	38	...
17	39	40	...
18	41	42	...
19	43	44	...
20	45	46	...
21	47	48	49
22	50	51	...
23	52	53	54
24	55	56	...
25	57	58	59

4 A Simpler Case: The Poisson Equation

For the finite element treatment of Poisson's equation, we begin by representing the scalar state function as a linear combination of finite element basis functions $u(x,y) = \sum_{i=1}^{N} c_i \quad \phi_i(x,y)$ Each basis function $\phi_i(x,y)$ is associated with a mesh node n_i of the same index, and ϕ_i is nonzero only over elements which include n_i as one of the nodes.

In formulating the Galerkin system for the coefficients c_i, we end up with a series of algebraic equations involving integrals involving products of basis functions, which can be reformulated as a set of linear equations $K * c = f$, where K is the stiffness matrix. If we concentrate on row i of this matrix, then we note that entry $K_{i,j}$ has the value $\int_\Omega \frac{\partial \phi_i}{\partial x}\frac{\partial \phi_j}{\partial x} + \frac{\partial \phi_i}{\partial y}\frac{\partial \phi_j}{\partial y} \quad d(\Omega)$. Now we note that, if there is no element that contains both nodes n_i and n_j, then the matrix entry $K_{i,j}$ must be zero. To see this, regard the integral over the region as a sum of integrals over elements. In each element that we consider, at least one of the two nodes does not occur, hence the corresponding basis function is zero over the entire element, hence so are its derivatives, making the integrand identically zero over the element. Hence the entire integral is zero, and the matrix element is zero.

For Poisson's equation, the geometry of the finite element mesh controls the zero structure of the stiffness matrix. To get a vivid visual image of this sparsity, start with a plot of a mesh, choose any particular node n_i, and highlight the (few) elements that contain that node. The nonzero entries in row i can only occur in locations corresponding to the other nodes included in those elements. All other entries must be zero.

5 Banded Matrix Techniques

The simple two dimensional storage of a full matrix seems a very natural scheme (although inefficient for our large sparse problems). We understand that it is worth looking at other schemes for storing the matrix. But even though these schemes can sometimes be described very simply in words, it can be surprisingly awkward or laborious to correctly implement the corresponding storage scheme as a computer code.

Our ultimate aim is a fairly general sparse format, but before we get there, it is worth "practicing" by considering the changes needed to go from a full storage matrix format to a standard general band matrix format, such as the **GB** format used by the LINPACK factor and solve routines DGBFA and DGBSL, or the corresponding LAPACK routines DGBTRF and DGBTRS.

We begin with a simple definition. We say a matrix is *a banded matrix* if there exist number **ml** and **mu** such that, in every row **i** of the matrix, the only nonzero entries lie between columns $i - ml$ and $i + mu$. If **ml** and **mu** are the smallest possible values with this property, we call **ml** the lower bandwidth of the matrix, **mu** the upper bandwidth of the matrix, and **m = ml+1+mu** the (total) bandwidth of the matrix.

Obviously, in a trivial sense, *every* matrix is banded. However, we are only interested in cases where the bandwidth **m** is much less than the order **n** of the matrix. In such a case, we will see that the corresponding finite element matrix can be stored using much less storage, and that Gauss elimination can be carried out more rapidly.

5.1 Computing the Bandwidth

We presume that we have access to the array **ElementNode(1:ElementOrder,1:ElementNum)** which lists, for each element, the indices of the associated nodes. For now, let us assume that we are working with the Poisson equation; and that there is a 1-1 correspondence between nodes and variables; and that the variable and node indices are equal.

For this simpler case, we can easily determine the pattern of nonzeros in the stiffness matrix. Starting with the first element, simply examine the list of node indices, which might be $(3, 12, 4)$. Since node 3 appears in an element with node 12, $K(3, 12)$ will be nonzero, as will $K(12, 3)$. In fact, we now have some information about 3 rows of the matrix, in particular, that in row 3 we have nonzero entries at (3,3), (3,12), (3,4), in row 4 we have (4,3), (4,4) and (4,12), and in row 12 we have (12,3), (12,4) and (12,12).

Of course, this is just a start. Each element we examine will give us more information like this, about several rows of the matrix. It sounds like a very haphazard process, but as we examine each element, we are sure to encounter every single nonzero entry of the matrix. Now the lower bandwidth ML is simply the maximum value of $I - J$ for all possible entries. So to compute it, we can simply examine every element, as we suggested, compute the value of $I - J$ for each pair of nodes in that element, and keep the maximum value of this quantity that we observe. Similarly, if we compute $J - I$ for every pair of nodes in each element, we get MU. Since these two quantities are equal, we can simply compute the maximum $|I - J|$ that we encounter.

Therefore, we have a simple procedure for computing the bandwidth in this case:

ML = 0 MU = 0 ...

For the Navier-Stokes equations, the procedure is similar. We still go from element to element, but now we need to look not at the node indices themselves, but at the indices of the variables associated with each node. Thus, in an element with 6 nodes, we will have 15 variables whose indices we need to retrieve, so that we can compute their differences, keeping the maximum.

5.2 Banded Storage Formats

There are two related formats for storing the banded matrix which have an additional extremely important property, namely, we can store a matrix in these formats, and carry out Gaussian elimination without ever needing to access a location of the full matrix that has not been set aside in the banded format of the matrix. What we are talking about here is called *fill in.* In Gaussian elimination, we add multiples of one row to another. Although our matrix is initially banded, you can see that the elimination process could easily destroy the banded structure of the matrix. This would be a disaster, since we would then need to store numbers for which we have not allocated any space.

The first format assumes that we can carry out Gauss elimination without doing any row interchanges; that is, we do no pivoting. In that case, Gauss elimination does not need access to any matrix entries outside the band. Logically, the data within the band is a sort of parallelogram. To store it computationally, we need to reshape it as a rectangular array, whose width will be the bandwidth, and whose height will be the number of equations. This rearrangement is suggested pictorially:

```
11 12  0  0  0  0          0 11 12
21 22 23  0  0  0         21 22 23
 0 32 33 34  0  0  -----> 32 33 34
 0  0 43 44 45  0         43 44 45
 0  0  0 54 55 56         54 55 56
 0  0  0  0 65 66         65 66  0
```

The saving in storage seems small for this small example, but becomes considerable as the number of equations grows. As a cost of this saving, the value that we want to think of $A(I, J)$, is still stored in row I of the band storage matrix, but the column index must be recomputed.

The second banded format modifies the above storage scheme to allow for the possibility of partial pivoting. In this format, we essentially behave as though the matrix had twice as many subdiagonal nonzeros, to accommodate possible row swapping. Thus, for our simple example, the storage scheme is suggested by:

```
11 12  0  0  0  0         0  0 11 12
21 22 23  0  0  0         0 21 22 23
 0 32 33 34  0  0  -----> 0 32 33 34
 0  0 43 44 45  0         0 43 44 45
 0  0  0 54 55 56         0 54 55 56
 0  0  0  0 65 66         0 65 66  0
```

where some of the zero entries in the initial columns will become nonzero as part of the pivoting process.

It's clear that these schemes can be helpful if the system is actually banded. But to take advanatage of them, we need to determine the bandwidth of the system. We will now consider the issue of determining this bandwidth, for a structured or unstructured mesh.

6 The Sparseness of the Equations

The Navier Stokes equations are nonlinear. In approximating these nonlinear differential equations, the finite element method produces a set of nonlinear algebraic equations with the property that the corresponding Jacobian matrix is relatively sparse. In typical examples, sparseness may mean that each equation never involves more than 45 variables, no matter how large the total number of variables N becomes.

Full storage of the matrix would require N^2 entries. Simply storing the entries intelligently might only require $45 * N$ entries. If we can actually achieve a storage cost that grows linearly with N, we can handle far bigger problems than if our storage grows linearly. Thus, we must investigate and exploit the opportunity that the inherent sparseness of the finite element equations is offering us.

Of course, the storage of the Jacobian entries is not generally all that is required. At the very least, we need not just the values, but some way of recording the indices associated with these values. Moreover, if we are doing a traditional direct linear factorization and solution, we will surely need even more storage in order to handle the inevitable fill in. Nonetheless, we may find that the storage cost of this scheme only grows linearly with N, instead of quadratically, which means that we can construct and store Jacobians for much larger values of N.

6.1 The Simpler Case of Poisson's Equation

To see why the Jacobian matrix is sparse, it may be helpful to start with a simpler situation, in which we are solving the scalar Poisson's equation. Instead of a Jacobian matrix, we work with the stiffness matrix, but the story is essentially the same. The advantage of considering the Poisson equation is that we can ignore for a moment the complications of having two state variables, (one a vector function), of having two separate sets of basis functions, and of having to treat vertex nodes differently from midside nodes. None of those details affect the sparseness argument in any interesting way, but they certainly confuse the analysis!

6.2 The Navier-Stokes Case

The same argument can be made, after a number of unimportant adjustments, to show that the Jacobian matrix of the finite element residual functions for the Navier Stokes equations is also sparse. Assuming we have explained how this sparseness arises in the simpler case of Poisson's equation, we may now venture a more detailed examination of exactly *how sparse* a typical row of the Navier Stokes Jacobian might be.

Recall, then, that row i of the Jacobian matrix is associated with some basis function, which we generically denote ω_i, and that this basis function, in turn, is associated with some node which we will denote n_i.

Under the reasonable assumption that the mesh was created by starting with a logically rectangular mesh and splitting each rectangle into two triangles, and splitting all the triangles along the same diagonal, then the mesh will be extremely regular. In this case, it is not hard to verify the following facts, which may be checked by referring to the figure of the mesh of the "square region". (But please don't miss the point that we are not talking here about a general unstructured mesh!)

If basis function ω_i is associated with a vertex node n_i:

- there are usually 6 (but never more) elements containing node n_i
- there are usually a total of 19 (but never more) distinct nodes in these elements
- there are usually a total of 45 (but never more) basis functions or degrees of freedom associated with these nodes. (38 velocity components and 7 pressures)

If basis function ω_i is associated with a midside node n_i:

- there are usually 2 (but never more) elements containing node n_i
- there are usually a total of 9 (but never more) distinct nodes in these elements

- there are usually a total of 22 (but never more) basis functions or degrees of freedom associated with these nodes. (18 velocity components and 4 pressures)

So this tells us that, for a typical structured mesh, no matter how big the region or how many nodes and elements, or even how we number the nodes, elements, and variables, there are at most 45 nonzero entries in each row of the Jacobian. (The actual number of nonzeros is somewhat less, since some of the equations that could involve another variable don't actually do so; in particular, the continuity equation).

Using a basically rectangular mesh is really for illustration. Even if the mesh is unstructured, it will generally be the case that every row of the Jacobian will be sparse. The only way to keep this from happening requires an artificial arrangement, such as having many elements share a single node. Then, even if some individual rows of the Jacobian are not sparse (perhaps the number of entries in one row is a significant fraction of N), it must be the case that in the larger sense, the whole Jacobian is sparse (the total number of entries is very small compared to N^2.)

In short, typical finite element systems of equations, in which the basis functions are developed from the mesh, will involve only local coupling of variables, therefore the stiffness matrix (for a linear problem) or the Jacobian matrix (for a nonlinear problem) will be sparse, so that it makes sense to seek an intelligent way of handling sparse linear systems when working with finite element techniques.

7 Sparse Triplet Storage

There are many special storage formats for sparse matrices. One of the simplest schemes simply saves the row, column and value of each nonzero entry in the matrix, and is known as the *sparse triplet* format.

Thus, the data associated with sparse triplet matrix storage might be

- N, the number of variables and equations;
- $NZNUM$, the number of nonzero entries in the matrix;
- $IA(1 : NZNUM)$, $JA(1 : NZNUM)$, $A(1 : NZNUM)$, the row, index, column index, and value of each nonzero matrix entry.

Now suppose we are given the data associated with a sparse triplet matrix, and we are asked to retrieve the value $A_{i,j}$. With no other information, we would have to examine each pair of values (IA_k, JA_k) for k from 1 to $NZNUM$. This lookup can be very time-consuming, and there must be a better way. Since we will surely be looking up every entry of the matrix, we must find a better system.

Suppose, then, that we rearrange the data so that the entries in column 1 come first, followed by the entries in column 2, and so on. Then, at least, once we pass a certain point we will know we've gone too far.

We could do better. Since we've arranged our data by columns, we could keep track of where each column begins, and jump there directly. That is, if we create a vector COL of length $N + 1$, and set the extra entry $COL(N + 1)$ to be equal to $NZNUM + 1$, then we will have the property that if we're searching for matrix entries in column J, we only have to look at entries k between $COL(J)$ and $COL(J + 1) - 1$.

We still need to examine every entry in this range. But, we can do a little better if we also sort these values by their row index. In that case, we can apply binary search to find the right row index very quickly. These improvements lead to another sparse matrix storage format, to be discussed next.

8 Compressed Column Storage

Compressed column storage can be thought of as an optimized version of the sparse triplet format. We still store the nonzero entries and their row indices as before, but we do not store the column indices. Instead, we rely on the COL vector and the sorting of the data to implicitly give us the column information. Moreover, we assume that the entries associated with each column have been ascending sorted by row.

Thus, the data associated with compressed column storage might be:

- N, the number of variables and equations;
- $NZNUM$, the number of nonzero entries in the matrix;
- $COL(1:N+1)$, so that the entries for column J occur between $COL(J)$ and $COL(J+1)-1$;
- $ROW(1:NZNUM)$, $A(1:NZNUM)$, the row index and the value of each nonzero matrix entry.

Note that the important thing about compressed column storage is not the slight savings in size, in replacing $JA(1:NZNUM)$ by $COL(1:N+1)$, but rather, the speed with which entries can be looked up. In particular, if we want to know the value of the (I,J) entry of the matrix, the steps involved are as follows:

- Set $KLO = COL(J)$ and $KHI = COL(J+1)-1$.
- Using binary search for efficiency, determine the value K satisfying $KLO <= K <= KHI$ and $ROW(K) = I$.
- The (I,J) entry of the matrix is stored in $A(K)$.

The real work for a sparse matrix storage scheme is not in using the structure, but rather in setting it up. On paper, this may seem easy: simply take all the nonzeros in the first column, followed by all the nonzeros in the second column, and so on. But, particularly in the finite element case, this is not the proper way to think about the problem.

The full matrix is never actually constructed. Instead, by examining each individual element of the mesh, we discover that certain entries of the matrix are nonzero, and it is only after we have examined all of the entries that we know the complete matrix structure. Moreover, a given entry may be accessed by many elements, and thus we also face the problem of redundant information. Thus, it is worth while explaining in a little detail how the seemingly simple operation of collapsing the full matrix into the sparse structure can actually take place.

9 Building the Sparse Data Structures

Let us now suggest how, using information available in a typical finite element program, we can analyze the elements one by one, determine the Jacobian entries that must be nonzero by virtue of the geometry of this element, put all this information into one list, and then process that list so that we can arrive at the compressed column sparse storage format from which we may proceed with relative ease.

Again, we prefer to consider the related but simpler case of the stiffness matrix for the Poisson equation, rather than the Jacobian matrix of the Navier Stokes equations. As in previous discussions, the essential and interesting features of the argument will be the same, but we can avoid the complications arising from trying to deal with a vector variable and a scalar variable defined using separate basis functions.

So let us begin our considering by assuming that we are given the raw data defining a mesh of quadratic triangles in 2D:

- $NodeNum$, the number of nodes;
- $NodeXY(2, NodeNum)$, the coordinates of the nodes;
- $ElementNum$, the number of elements;
- $ElementNode(6, ElementNum)$, the indices of the nodes making up each element

As the following computations become fairly complicated, it may be helpful to fix a simple example, to which we may refer. Unfortunately, even the seemingly tiny example shown here will end up showing a fair amount of complexity. But working out the details for this example may help you to follow the algorithms, and to be convinced that they will apply in the more general and humanly unfathomable cases.

Although we will not need to know the coordinates of the nodes in this section, we will suppose that for the example problem, the $NodeNum = 15$ nodes have the following simple coordinates $NodeXY$:

and that the $ElementNum = 4$ elements have the following $ElementNode$ array:

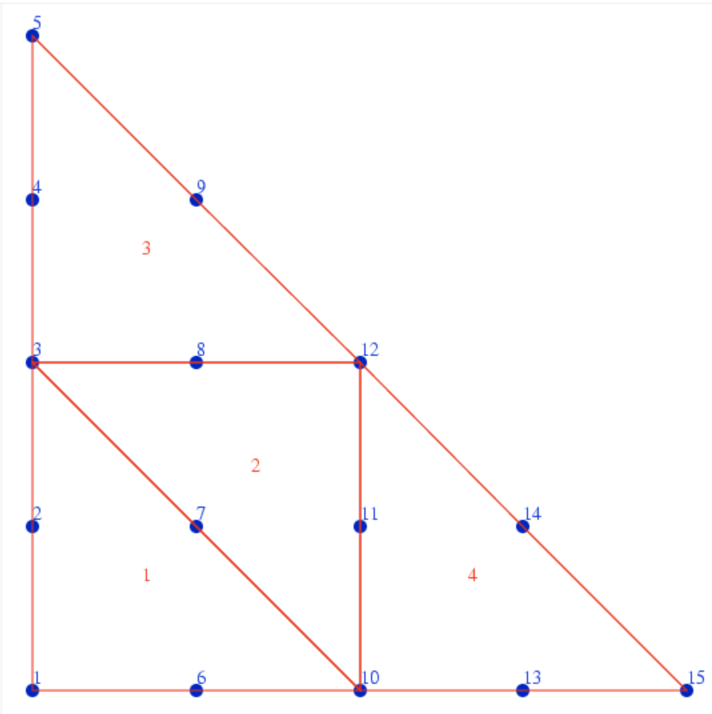

Figure 3: A triangular region to be analyzed for variable adjacencies.

9.1 Task One:Create the Element Neighbor Array

Our first task is to compute the *element neighbor array*

- $ElementNeighbor(3, ElementNum)$, lists the neighboring element on each side of the given element, or -1 if that is a boundary side.

You should think about this for a moment and agree that this array can be constructed entirely from the information in $ElementNode$. For instance, let us suppose we are trying to find the neighbor of an element along the side defined by vertex nodes labeled 17 and 99. We could simply examine the first three nodes (the vertices) of every element, and look for a second occurrence of 17 and 99. If that occurs, we have found the neighboring element, and if it does not, then we have encountered a boundary side.

We need to compute the **ElementNeighbor** array in a consistent, precise, and efficient way. We do so in three steps:

- Construct a list of edges. For each triangle, write down three records, listing the nodes on an edge, the index of the edge, and the index of the element. To make matching possible, always list the two nodes in ascending order.

- Sort the edge list. Then if two elements share an edge (I, J), there will be two consecutive entries in the edge list of the form $(I, J, side1, element1)$ and $(I, J, side2, element2)$.

- Process the edge list. Initialize all entries of **ElementNeighbor** to -1. Examine successive pairs of records in the edge list. When you spot a pair of consecutive items whose first two entries match, note that

```
ElementNeighbor(side1,element1) = element2
```

Node	X	Y
1	0.0	0.0
2	0.0	0.5
3	0.0	1.0
4	0.0	1.5
5	0.0	2.0
6	0.5	0.0
7	0.5	0.5
8	0.5	1.0
9	0.5	1.5
10	1.0	0.0
11	1.0	0.5
12	1.0	1.0
13	1.5	0.0
14	1.5	0.5
15	2.0	0.0

Element	N1	N2	N3	N4	N5	N6
1	1	10	3	6	7	2
2	12	3	10	8	7	11
3	3	12	5	8	9	4
4	10	15	12	13	14	11

```
ElementNeighbor(side2,element2) = element1
```

The array **ElementNeighbor** is now complete.

For our tiny example, we would have computed the following *ElementNeighbor* array:

Element	E1	E2	E3
1	-1	2	-1
2	3	1	4
3	2	-1	-1
4	-1	-1	2

Note that we temporarily needed a work array to store 4 items for every element's edge. It is always useful to be able to know beforehand the size of such temporary arrays, so that the appropriate amount of memory can be allocated. In this case, the size of the array is $ElementNum * 3 * 4$ because, for each element, we have 3 sides, and we are storing 4 items of information per side.

9.2 Task Two: Counting the Adjacencies and Setting the COL Vector

As we suggested just a moment ago, it is usually helpful, and sometimes absolutely necessary, to determine the amount of memory needed for arrays that you create as you go in a computer program. Our goal is to set up the adjacency information for this problem.

Since we haven't actually counted the number of variables, let us first note that by examining the *ElementNode* array, we can determine that nodes that appear in the first three entries are *pressure nodes*, which have 3 variables associated with them, while the remaining nodes are *velocity nodes* which have only 2. If we make a temporary list of all the nodes, and then scan the *ElementNode* array, we can assign a 3 or a 2 to each node, and at the end, sum up these values to get the number of variables. (Of course, some nodes will be encountered more than once, but, as long as this is a legal triangulation, a node will always appear as only one kind of node or the other.

Thus, for our example, if we process each element in order, here is the state of our knowledge about the number of variables per node: and by summing, we discover that we have 36 variables, namely 15 horizontal

Element	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
1	+3	+2	+3	?	?	+2	+2	?	?	+3	?	?	?	?	?
2	3	2	3	?	?	2	2	+2	?	3	+2	+3	?	?	?
3	3	2	3	+2	+3	2	2	2	+2	3	2	3	?	?	?
4	3	2	3	2	3	2	2	2	2	3	2	3	+2	+2	+3

velocities, 15 vertical velocities, and 6 pressures.

There are many ways of assigning indices to the variables. One approach starts by listing all the horizontal velocities, another starts by assigning all the variables associated with node 1. Careful numbering can make a big difference if we are defining a banded matrix, but is of little importance for a sparse matrix.

Following the second numbering convention, we have the following relationship between nodes and variables for our tiny example.

Node	U	V	P
1	1	2	3
2	4	5	-1
3	6	7	8
4	9	10	-1
5	11	12	13
6	14	15	-1
7	16	17	-1
8	18	19	-1
9	20	21	-1
10	22	23	24
11	25	26	-1
12	27	28	29
13	30	31	-1
14	32	33	-1
15	34	35	36

We can store this information in a convenient way by creating three arrays, NodeUVariable, NodeVVariable, and NodePVariable.

Now we are going to need to set up a sparse matrix that is logically of order VariableNum. In order to set up the sparse matrix, we need to *count* the number of nonzero entries that will occur. Each of these nonzeros corresponds to a "variable adjacency", which is similar to the geometric adjacency of two nodes. Here, we say that two variables are "adjacent" if the nodes they are associated with both occur in some triangle.

In our example, therefore, variable 1 and variable 16 are "adjacent" because they are associated with nodes 1 and 7, respectively, and these nodes both occur in triangle 1. However, variable 26 and variable 5 are not adjacent, because they are associated with nodes 11 and 2, which are never in the same triangle.

Now I repeat that we need to know a lot about this adjacency relationship, but to start with, we need to know the number of adjacency relationships that exist. We will denote this number as **ADJNUM**, but it will be equivalent to what is usually called **NZNUM** for sparse matrices, since it represents the number of nonzero entries to be set aside.

It will turn out that as we count the adjacencies, we can also keep track of information that will allow us to set up the sparse compressed column pointer array that we denoted by **COL**.

To do these tasks, we must consider every possible adjacency. This information is readily available, implicitly, in the **ELEMENTNODE** and the NodeUVariable, NodeVVariable and NodePVariable arrays that we just constructed.

The problem is that the adjacency of any pair of variables may be recorded just once, twice, or many times, depending on the geometry of the mesh. Therefore, we have to go through the geometry information very carefully, so that we examine all the adjacencies, but ignore multiplicities. Essentially, we only want to count each adjacency the very first time we see it.

To do so, we proceed as follows. First, we set **ADJNUM** to 0. We also set up and zero out an array **COUNT** of length **VARIABLENUM**. We will use this array to keep a little finer tally on the adjacencies. Every time we find an adjacency, we will increment the overall counter **ADJNUM**, but we will also bump up the value of **COUNT** for the two variables involved.

The first adjacency information to process is easy: every variable is presumed to be adjacent to itself. So the VARIABLENUM entries of COL each go up by 1, and ADJNUM goes up by VARIABLENUM.

The next adjacency information states that each variable is adjacent to the other variables that share its node. So we loop over the nodes. If the node is a pressure node, then the 3 variables at that node each add 2 adjacencies, so ADJNUM goes up by 6, and the 3 entries of COL each go up by 2. If the node is a velocity node, the 2 variables each add 1 adjacency, so ADJNUM goes up by 2, and the 2 entries of COL go up by 1.

Now we must examine the adjacency information describing variables at different nodes. We proceed by looping over each triangle. So assume we are now considering triangle T.

There are some adjacencies that are absolutely certain not to be duplicated. In particular, triangle T is the only triangle in which it will be possible for the following adjacencies to occur: (N1,N5), (N2,N6), (N3,N4). Therefore, these can be added to the data.

(Please note my shorthand here. I am referring to node adjacency, although I am ultimately interested in variable adjacency. If two nodes are adjacent, then the variables at each node are to be considered pairwise adjacent. Therefore, when I say in shorthand that N1 is adjacent to N5, I am actually saying the the three variables (U1,V1,P1) are each adjacent to the two variables (U5,V5), making for a total of 12 new adjacencies of variables.)

All the other potential adjacencies involve nodes that share a side of the triangle. Because of that, the adjacency could already have been encountered because, perhaps, we already processed the other triangle that might share that side. Therefore, we consider the remaining data one edge at a time.

We suppose that triangle T has sides E1, E2, and E3. Sharing each side is a neighboring triangle, T1, T2 and T3, whose index is recorded in ElementNeighbor. That index is -1 in the case where the edge is a boundary edge.

We loop over the three edges. Edge 1 has the possible adjacencies of (N1,N2), (N1,N4) and (N2,N4). Have we seen these adjacencies before? Only if the Edge 1 is not a boundary edge (T1 is not -1) AND if T1 is less than T (because we are processing the triangles in order. So if T1 was less than T, we'd have seen edge E1 already when we worked on T1.) So we make the check, and if the data is new, we record it, and update ADJNUM and COUNT.

Similarly, if Edge 2 is "new", then we record the adjacencies of (N2,N3), (N2,N5) and (N3,N5), and if Edge 3 is "new", we record adjacencies of (N1,N3), (N1,N6), and (N3,N6).

We then move on to the next triangle, until we have exhausted them all.

This is a complicated process, but at the end we are guaranteed to have worked through the adjacency information represented by ELEMENTNODE and recorded exactly once the adjacency of every variable.

Moreover, as long as we were careful to increment our COUNT vector correctly each time we added adjacencies, we can set the **COL** array needed by the sparse compressed column storage scheme: **COL(VARIABLE)** is 1 plus the sum of **COUNT(1:VARIABLE-1)**.

At the end of this task, the ElementNeighbor array has served its purpose, and may be discarded.

9.3 Task Three: Setting the ROW vector

We now go back and fill the entries of ROW...

And we can allocate A.

As mentioned before, if the mesh is fixed, then the sparsity structure does not change, and the sparse matrix data structure only needs to be computed once.