CS 294-73
Software Engineering for Scientific Computing

Lecture 8: Unstructured grids and sparse matrices
Back to Poisson’s equation.
Some Vector Calculus

Gradient operator: \( \nabla \Psi = \left( \frac{\partial \Psi}{\partial x}, \frac{\partial \Psi}{\partial y} \right) \)

Divergence operator: \( \nabla \cdot (F_x, F_y) = \frac{\partial F_x}{\partial x} + \frac{\partial F_y}{\partial y} \)

Laplacian: \( \Delta \phi = \nabla \cdot (\nabla \phi) = \frac{\partial^2 \Psi}{\partial x^2} + \frac{\partial^2 \Psi}{\partial y^2} \)

- Green’s theorem (aka integration by parts)
  \[ - \int_{\Omega} \Psi(x)(\nabla \cdot (\nabla \phi))(x) \, dx = \int_{\Omega} \nabla \Psi \cdot \nabla \phi \, dx + \int_{\partial \Omega} \Psi(x)(\nabla \phi)(x) \, dS \]

- If \( \Psi \equiv 0 \) on \( \partial \Omega \), then
  \[ - \int_{\Omega} \Psi(x)(\nabla \cdot (\nabla \phi))(x) \, dx = \int \nabla \Psi \cdot \nabla \phi \, dx \]
We want to solve Poisson’s equation (note the sign convention)

\[-\Delta \phi = f \text{ on } \Omega\]
\[\phi = 0 \text{ on } \partial \Omega\]

We want find a weak solution, i.e.

\[\int_\Omega (-\Delta \phi)(x) \Psi(x) dx = \int_\Omega f(x) \Psi(x) dx \text{ on } \Omega\]

For all continuous piecewise smooth test functions

- \(\Psi(x)\) with \(\Psi = 0\) on \(\partial \Omega\)

Applying Green’s Theorem, this is the same as

\[\int_\Omega \nabla \phi \cdot \nabla \Psi dx = \int_\Omega f(x) \Psi(x) dx, \Psi \in V\]
Finite element discretization

Step 1: we discretize our domain as a union of triangles.

Step 2: We replace $V$ by $V^h$, a finite-dimensional space of test functions. For this exercise, we will use linear combinations of continuous, piecewise linear functions, indexed by interior nodes, linear on each triangle containing the node. A basis for this space is given by $\{\Psi_n^h(x) : n \in N_I\}$.

Step 3: We also approximate the solution as a linear combination of the elements in $V^h$.

\begin{align*}
\Psi_n^h(x_n') &= \delta_{nn'} , n' \in N
\end{align*}
We apply the weak form of the equations to the finite-dimensional subspace $V^h$

$$\phi(x) \approx \phi^h(x) = \sum_{n \in N_I} a_n \Psi_n^h(x)$$

$$\int_\Omega \nabla \phi^h \cdot \nabla \Psi_n^h \, dx = \int_\Omega f(x) \Psi_n^h(x) \, dx, \quad n \in N_I$$

$$\sum_{n' \in N_I} a_{n'} \int_\Omega \nabla \Psi_{n'}^h \cdot \nabla \Psi_n^h \, dx = \int_\Omega f(x) \Psi_n^h(x) \, dx$$

$$(La)_n = \sum_{n' \in N_I} L_{n,n'} a_{n'} = b_n$$

$$L_{n,n'} = \int_\Omega \nabla \Psi_{n'}^h \cdot \nabla \Psi_n^h \, dx, \quad b_n = \int_\Omega f(x) \Psi_n^h(x) \, dx$$
Elements

Two issues:
• Computing L.
• Quadrature for computing b.

\[ L_{n,n'} = \int_{\Omega} \nabla \Psi^h_{n'} \cdot \nabla \Psi^h_n \, d\mathbf{x} \ , \ (n, n' \in \mathcal{N}_I) \]

\[ = \sum_{e=0 \ldots E-1} \int_{K_e} \nabla \Psi^h_{n'} \cdot \nabla \Psi^h_n \, d\mathbf{x} \]

\[ = 0 \text{ unless } n, n' \in K_e \]

*Fig 1.9 The basis function \( \varphi_j \).*
Matrix Assembly

Pseudocode: Interior Nodes $= \mathcal{N}_I$, Elements $e = 0, \ldots E - 1$

Initialize $L = 0$

for $e = 0 \ldots E - 1$

\[
\begin{align*}
&\text{for } (x_n, x_{n'}) \in K_e : (n, n') \in \mathcal{N}_I \\
&L_{n,n'} + = \int_{K_e} \nabla \Psi^h_{n'} \cdot \nabla \Psi^h_n d\mathbf{x}
\end{align*}
\]

endfor

endfor

- $L$ is a matrix with mostly zero entries. But it is nice: symmetric, positive-definite, M-matrix.

- $\nabla \Psi^h_n$ is a constant vector, easily computed.

- We’re building a matrix dimensioned by nodes by iterating over elements and building it up incrementally.

Fig 1.9 The basis function $\varphi_j$. 
Getting the right-hand side

Quadrature for $b$: midpoint rule on each element.

$$\int_{\Omega} \Psi_n^h f \, d\mathbf{x} = \sum_{K_e} \int_{K_e} \Psi_n^h f \, d\mathbf{x}$$

$$\int_{K_e} \Psi_n^h f \, d\mathbf{x} \approx \text{Area}(K_e) f(x_{e\,\text{centroid}}) \Psi_n^h(x_{e\,\text{centroid}})$$

Initialize $b = 0$

for $e = 0 \ldots E - 1$

for $x_n \in K_e : n \in \mathcal{N}_I$

$$b_n + = \text{Area}(K_e) f(x_{e\,\text{centroid}}) \Psi_n^h(x_{e\,\text{centroid}})$$

endfor

endfor

More element magic.
Point Jacobi Iteration

Motivation: to solve $La = b$, we compute it as a steady-state solution to an ODE.

$$\frac{da}{dt} = -La + b$$

If all of the eigenvalues of $L$ are positive, then

$$La_\infty = b, \quad a_\infty = \lim_{t \to \infty} a(t)$$

Point Jacobi: use forward Euler to solve ODE.

Stop when the *residual* has been reduced by a suitable amount.

$$a^{l+1} = a^l + \lambda(b - La^l), \quad l = 0, 1, \ldots; \quad a^0 = 0 \quad \lambda > 0$$

$$||b - La^l|| \leq \epsilon||b||$$
Our matrix has the following properties:

- Symmetric, positive-definite: \( L = L^T \), \( \forall \mathbf{v}, (L\mathbf{v}) > 0 \) if \( \mathbf{v} \neq \mathbf{0} \)
- Positive along diagonal.
- Rows sum to a non-negative number: \( L_{k,k} \geq -\sum_{k'} L_{k,k'} \)
- For triangles sufficiently close to equilateral, the nonzero off-diagonal elements are non-negative, i.e. \( \nabla \psi^h_n \cdot \psi^h_{n'} \leq 0 \) on \( K_e \).
Choosing a Relaxation Parameter

This leads to the following choice for our relaxation parameter.

\[ \lambda = \alpha \frac{1}{\max \limits_k L_{k,k}} , \quad \alpha < 1 \]

If your grid is strongly-varying, may want to use a local relaxation parameter (you will not be doing this in the present assignment).

\[ a^{l+1}_k = a^l_k + \lambda_k (b - L a^l)_k \]
\[ \lambda_k = \alpha \frac{1}{L_{k,k}} \]
Sparse Matrices.

- Compact basis function space results in a linear operator (Matrix) that has mostly zero entries.

Typical non-zero entries in A matrix from a finite element problem.
RectMDArray can hold this matrix, but wasteful

- Wasteful in several ways
  - You waste memory storing the number 0 in a lot of places
  - You waste floating point instructions performing multiplication with 0
  - You waste processor bandwidth to memory
  - You waste hits in your cache
We represent a sparse matrix as two vectors of vectors:

\[
\begin{bmatrix}
1.5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 2.3 & 0 & 1.4 & 0 & 0 & 0 & 0 \\
0 & 0 & 3.7 & 0 & 0 & 0 & 0 & 0 \\
0 & -1.6 & 0 & 2.3 & 9.9 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 5.8 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 7.4 & 0 & 0 \\
0 & 0 & 1.9 & 0 & 0 & 0 & 4.9 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 3.6 \\
\end{bmatrix}
\]

We represent a sparse matrix as two vectors of vectors: `vector<vector<double>>` to hold the matrix elements, `vector<vector<int>>` to hold the column indices.

Compressed-sparse-row (CSR) representation.
SparseMatrix Class

class SparseMatrix
{
public:
    /// set up an M rows and N columns sparse matrix
    SparseMatrix(int a_M, int a_N);
    /// Matrix Vector multiply.  a_v.size()==a_N, returns vector of size a_M
    vector<double> operator*(const vector<double>& a_v) const;
    /// accessor functions for get and set operations of matrix elements
    double& operator[](const array<int,2>&);

private:
    int m_m, m_n;
    float m_zero;
    vector<vector<double> > m_data;
    vector<vector<int> > m_colIndex;
};

Part of your homework 2 will be to implement this class, with a few more functions

If necessary, sparse matrix automatically adds a new matrix element when you reference that location, and initializes it to zero.

For each non-zero entry in ‘A’ we keep one float, and one int indicating which column it is in
Setup for Homework 2

• Build an operator corresponding to a triangular element discretization of the Poisson equation.
• Use an iterative solver to solve the equation.
• What we will provide:
  - Triangular grids, stored in files.
  - Classes for reading those files, and storing and manipulating computing geometric information.
  - A class for writing out the solution in a form that can be viewed by VisIt.
• You will write:
  - A class FEPoissonOperator that generates and stores the sparse matrix, and applies the operator to the right-hand side.
  - The SparseMatrix class.
  - An implementation of point Jacobi iteration to solve the resulting linear system.

We will discuss the details of these in the next few slides.
Node, Element, and FEGrid

class Node
{
public:
    Node();
    Node(array<double,DIM> a_position,
         const int& a_interiorNodeID,
         const bool& a_isInterior);
    /// Constant access to node Location in space.
    const array<double,DIM>& getPosition() const;
    const int& getInteriorNodeID() const;
    const bool& isInterior() const;
private:
    array<double,DIM> m_position;
    bool m_isInterior;
    int m_interiorNodeID;
};

Three different integer ID’s for nodes:
- Where they are in the vector of all nodes making up the triangular grid;
- Where they are in the vector making up the interior nodes;
- Where they are in the vector making up the nodes on an element (localNodeNumber)
Node, Element, and FEGrid

#define VERTICES 3
class Element
{
    public:
        Element();
        /// Constructor.
        Element(array<int, VERTICES>& a_tr);
        /// Destructor.
        ~Element();
        /// local indexing to get nodeNumber.
        const int& operator[](const int& a_localNodeNumber) const;
    private:
        array<int, VERTICES> m_vertices;
};

Local node numbers for element $i_{Elt}$. 
Node, Element, and FEGrid

We’re implementing this one (along with Node and Element) for you – you just have to use them correctly.

```cpp
class FEGrid { public:
    FEGrid();
    // Constructor by reading from file.
    FEGrid(char* a_nodeFileName, char* a_elementFileName);
    // Destructor.
    ~FEGrid();
    // Get number of elements, nodes, interior nodes.
    int getNumElts() const;
    int getNumNodes() const;
    int getNumInteriorNodes() const;

    Read in the file names from argv.
```
Element , Element, and FEGrid

... Element-centered calculus.
/// Compute gradient of basis function at node
/// a_localNodeNumber = 0,...,VERTICES-1, on element
a_eltNumber.
array<double,DIM> gradient(const int& a_eltNumber,
                          const int& a_localNodeNumber) const;

/// Compute centroid of element.
array<double,DIM> centroid(const int& a_eltNumber) const;

/// Compute area of element.
float elementArea(const int& a_eltNumber) const;

/// Compute value of basis function.
float elementValue(const array<double,DIM>& a_xVal,
                   const array<double,DIM>& a_gradient,
                   const int& a_eltNumber,
                   const int& a_localNodeNumber) const;
Node, Element, and FEGrid

...  
/// get reference to node on an element.
const Node& getNode(const int& a_eltNumber,
    const int& a_localNodeNumber) const;
/// Get reference to a Node given its global index.
const& Node& getNode(const int& a_nodeNumber) const;

private:
    vector<Node > m_nodes;
    vector<Element> m_elements;
    int m_numInteriorNodes;
};

Notice what we don’t have: neither an explicit mapping that gives all of the elements touching a given node, nor one that maps interiorNodes into nodes. The first one we don’t need, and the second is encoded implicitly in Node.
class FE_Poisson_Operator
{
    public:
        FE_Poisson_Operator();
        FE_Poisson_Operator(const FEGrid& a_grid);
    void applyOperator(vector<float>& a_LoPhi, const vector<double>& a_phi) const;
    void makeRHS(vector<double>& a_rhsAtNodes, const vector<float>& a_rhsAtCentroids) const;
    const FEGrid& getFEGrid() const;
    const SparseMatrix& getSparseMatrix() const;
    ~FE_Poisson_Operator();

    private:
        SparseMatrix m_matrix;
        FEGrid m_grid;
};

Note that a PHI is defined only on the interior nodes, as is a_LoPhi, a_rhsAtNodes.
Building the Sparse Matrix (**FE\text{Poisson}::\text{FE\text{Poisson}}(\ ...\ )**)

- Our sparse matrix has dimensions $N_I \times N_I$ ($\text{getNumInteriorNodes}()$)
- To compute the inner product on each element, you need \textit{gradient, elementArea}.
- Fill in $L_{n,n'}$ incrementally, by incrementing matrix elements corresponding to pairs of interior nodes in each element, then iterating over elements ($\text{getNode}(\ ...\ ), \text{Node::InteriorNodeID}()$).

\[
L_{n,n'} = \int_{\Omega} \nabla \Psi_{n'}^h \cdot \nabla \Psi_n^h \, d\mathbf{x}, \quad (n, n' \in N_I)
\]

\[
= \sum_{e=0 \ldots E-1} \int_{K_e} \nabla \Psi_{n'}^h \cdot \nabla \Psi_n^h \, d\mathbf{x}
\]

\[
= 0 \text{ unless } n, n' \in K_e
\]

Initialize $L = 0$

for $e = 0 \ldots E - 1$

\[
L_{n,n'} + \int_{K_e} \nabla \Psi_{n'}^h \cdot \nabla \Psi_n^h \, d\mathbf{x}
\]

endfor

Sparse matrix automatically adds new matrix element when you index that location, and initializes it to zero.
Building the Right-hand Side (**makeRHS**)  

- Our right-hand side is an $\mathcal{N}_I$-dimensional vector (``getNumInteriorNodes()``), while our input $f$ a vector of values evaluated at the centroids of elements (``getNumElements()``, ``centroid(...)``).

- Fill in $b$ incrementally, by iterating over elements, then computing interior nodes in each element (``getNode(...)``, ``Node::InteriorNodeID()``).

- Use `elementValue(...)`, `elementArea(...)` to compute contribution from each node in an element.

\[
\int_{\Omega} \Psi^h_n f \, dx = \sum_{K_e} \int_{K_e} \Psi^h_n f \, dx
\]

\[
\int_{K_e} \Psi^h_n f \, dx \approx \text{Area}(K_e) f(x_e^{\text{centroid}}) \Psi^h_n(x_e^{\text{centroid}})
\]

Initialize $b = 0$

for $e = 0 \ldots E - 1$
  
  for $x_n \in K_e : n \in \mathcal{N}_I$
    
    $b_n += \text{Area}(K_e) f(x_e^{\text{centroid}}) \Psi^h_n(x_e^{\text{centroid}})$
  
endfor

endfor