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

\[- \int_{\Omega} \Psi(x)(\nabla \cdot (\nabla \phi))(x) \, dx = \int_{\Omega} \nabla \Psi \cdot \nabla \phi \, dx \]
Weak Form of Poisson’s Equation

We want to solve Poisson’s equation (note the sign convention)

$$-\Delta \phi = f \quad \text{on } \Omega$$

$$\phi = 0 \quad \text{on } \partial \Omega$$

We want find a weak solution, i.e.

$$\int_{\Omega} (-\Delta \phi)(\mathbf{x}) \Psi(\mathbf{x}) d\mathbf{x} = \int_{\Omega} f(\mathbf{x}) \Psi(\mathbf{x}) d\mathbf{x} \quad \text{on } \Omega$$

For all continuous piecewise smooth test functions

$$\cdot \quad \Psi(\mathbf{x}) \quad \text{with } \Psi = 0 \quad \text{on } \partial \Omega$$

Applying Green’s Theorem, this is the same as

$$\int_{\Omega} \nabla \phi \cdot \nabla \Psi d\mathbf{x} = \int_{\Omega} f(\mathbf{x}) \Psi(\mathbf{x}) d\mathbf{x} \quad , \Psi \in V$$
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 $\{\Psi_n^h(x) : n \in \mathcal{N}_I\}$.

Step 3: We also approximate the solution as a linear combination of the elements in $V^h$.
Weak form -> matrix equation.

We apply the weak form of the equations to the finite-dimensional subspace $V^h$:

$$\phi(\mathbf{x}) \approx \phi^h(\mathbf{x}) = \sum_{n \in \mathcal{N}_I} a_n \Psi^h_n(\mathbf{x})$$

$$\int_{\Omega} \nabla \phi^h \cdot \nabla \Psi^h_n d\mathbf{x} = \int_{\Omega} f(\mathbf{x}) \Psi^h_n(\mathbf{x}) d\mathbf{x}, \ n \in \mathcal{N}_I$$

$$\sum_{n' \in \mathcal{N}_I} a_{n'} \int_{\Omega} \nabla \Psi^h_{n'} \cdot \nabla \Psi^h_n d\mathbf{x} = \int_{\Omega} f(\mathbf{x}) \Psi^h_n(\mathbf{x}) d\mathbf{x}$$

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

$$L_{n,n'} = \int_{\Omega} \nabla \Psi^h_{n'} \cdot \nabla \Psi^h_n d\mathbf{x}, \ b_n = \int_{\Omega} f(\mathbf{x}) \Psi^h_n(\mathbf{x}) d\mathbf{x}$$
Elements

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

$$L_{n,n'} = \int_{\Omega} \nabla \Psi_{n'}^h \cdot \nabla \Psi_n^h \, d\mathbf{x} \ , \ (n, n' \in \mathcal{N}_I)$$

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

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

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 $e = 0 \ldots E - 1$
  for $(x_n, x_{n'}) \in K_e : (n, n') \in \mathcal{N}_I$
    $$L_{n, n'} + = \int_{K_e} \nabla \psi^h_{n'} \cdot \nabla \psi^h_n dx$$
  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.
Getting the right-hand side

Quadrature for \( b \): midpoint rule on each element.

\[
\int_{\Omega} \Psi_n^h f \, dx = \sum_{K_e} \int_{K_e} \Psi_n^h f \, dx
\]

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

Initialize \( b = 0 \)

for \( e = 0 \ldots E - 1 \)

for \( x_n \in K_e : n \in N_I \)

\[
b_n + = \text{Area}(K_e) f(x_e^{\text{centroid}}) \Psi_n^h(x_e^{\text{centroid}})
\]

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

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

Point Jacobi: use forward Euler to solve ODE.

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

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

$$\|b - La^l\| \leq \epsilon \|b\|$$
Matrix Properties

Our matrix has the following properties:

- Symmetric, positive-definite: \( L = L^T \), \( v \cdot (Lv) > 0 \) if \( v \neq 0 \)
- 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. \( \nabla \Psi_n^h \cdot \Psi_{n'}^h \leq 0 \) on \( K_e \).
Choosing a Relaxation Parameter

This leads to the following choice for our relaxation parameter.

\[ \lambda = \alpha \frac{1}{\max_k L_{k,k}} , \alpha < 1 \]

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

\[ a_{k}^{l+1} = a_{k}^{l} + \lambda_{k} (b - La_{k}^{l})_{k} \]

\[ \lambda_{k} = \alpha \frac{1}{L_{k,k}} \]
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 was floating point instructions performing multiplication with 0
  - You waste processor bandwidth to memory
  - You waste hits in your cache
Sparse Matrix representation using vectors

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.
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;
};
Node, Element, and FEGrid

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

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_PoissonOperator
{
public:
    FE_PoissonOperator();
    FE_PoissonOperator(const FE_Grid& a_grid);
    void applyOperator(vector<float>& a_LOfPhi, const vector<double>& a_phi) const;
    void makeRHS(vector<double>& a rhsAtNodes, const vector<float>& a rhsAtCentroids) const;
    const FE_Grid& getFE_Grid() const;
    const Sparse_Matrix& getSparse_Matrix() const;
    ~FE_PoissonOperator();
private:
    Sparse_Matrix m_matrix;
    FE_Grid m_grid;
};

Note that a_phi is defined only on the interior nodes, as is a_LOfPhi, a_rhsAtNodes.
Building the Sparse Matrix (\texttt{FE\textcolor{red}{Poisson}::FE\textcolor{red}{Poisson}( ... )})

- Our sparse matrix has dimensions \( N_I \times N_I \)
  \( (\text{getNumInteriorNodes}()) \)
- To compute the inner product on each element, you need \texttt{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 \( L = 0 \)

\[
\text{for } e = 0 \ldots E - 1 \quad \\
\quad \text{for } (\mathbf{x}_n, \mathbf{x}_{n'}) \in K_e : (n, n') \in N_I \quad \\
L_{n,n'} += \int_{K_e} \nabla \Psi_{n'}^h \cdot \nabla \Psi_n^h \, d\mathbf{x} \\
\text{endfor} \\
\text{endfor}
\]

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

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

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

- Use $\text{elementValue}(...)$, $\text{elementArea}(...)$ to compute contribution from each node in an element.

\[
\int_{\Omega} \Psi_n^h f \, dx = \sum_{K_e} \int_{K_e} \Psi_n^h f \, dx
\]
\[
\int_{K_e} \Psi_n^h f \, dx \approx \text{Area}(K_e) f(x_e^{\text{centroid}}) \Psi_n^h(x_e^{\text{centroid}})
\]

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