Briefing Notes — Parametric Surfaces and Surface Integrals

Supporting Material for MECH 222 Computer Lab 3

IDEA 1: *Double integrals and surface integrals from triangulations*

This week we are going to build something you can use as-is: a function that returns a computed approximation to the double integral

$$I = \iint_D f(x,y)\,dA(x,y).$$

In fact, the ideas involved are powerful enough to calculate something more versatile—the surface integral

$$J = \iint_{\mathcal{S}} f(x,y,z)\,dS. \tag{1}$$

The interpretation for J should sound familiar. A certain surface, named $\mathcal{S}$, is covered in some kind of goo with density $f(x,y,z)$ gunits per square metre. We imagine splitting $\mathcal{S}$ into tiny patches, each with infinitesimal area dS square metres. Then the product $f\,dS$ tells the amount of goo on that patch, and the symbol $\iint_{\mathcal{S}}$ calls for the generalized sum of all the little bits just mentioned. The result is J, the total amount of goo (in gunits) on $\mathcal{S}$. In particular, when f is the constant function $f(x,y,z) = 1$,

$$\iint_{\mathcal{S}} dS = \text{Area}(\mathcal{S}), \tag{2}$$

and *this is something you have already computed in Computer Lab 2*!

The above interpretation for J is just what we have always said about double integrals like I: the added versatility in J comes from allowing any flying surface in space to be the domain of integration. The double integral I covers the special case where the surface of interest happens to be a flat patch of the (x,y)-plane named D. (We write an infinitesimal patch of area as dA when we know it lies in one of the coordinate planes, and as dS when it is allowed to be flying anywhere in space. Using two different symbols for almost the same thing will help organize our hand calculations later in the course, but it makes very little difference in this lab.)

To invent an approximation for J in (1), start with our approximation for $\text{Area}(\mathcal{S})$ in Computer Lab 2. We just split $\mathcal{S}$ into many small triangles and added their areas. Symbolically,

$$\text{Area}(\mathcal{S}) = \sum_{i=1}^{N_{\text{tri}}} A(T_i), \tag{3}$$

where N_{tri} is the number of triangles involved; T_i is the name for triangle number i; and $A(T_i)$ is the area of T_i. The corresponding approximation for J is this natural extension:

$$J \approx \sum_{i=1}^{N_{\text{tri}}} \overline{f}_i\, A(T_i). \tag{4}$$

We simply multiply each triangle's area by a representative value for the function f. On triangle T_i, we use $\overline{f}_i$, the average of the three f-values at the vertices. In symbols,

$$\overline{f}_i = \frac{f\left(x_1^{(i)}, y_1^{(i)}, z_1^{(i)}\right) + f\left(x_2^{(i)}, y_2^{(i)}, z_2^{(i)}\right) + f\left(x_3^{(i)}, y_3^{(i)}, z_3^{(i)}\right)}{3}. \tag{5}$$

(Here the vertices T_i are denoted $\mathbf{r}_1^{(i)}$, $\mathbf{r}_2^{(i)}$, $\mathbf{r}_3^{(i)}$.)

The approximation in (4)–(5) makes sense because *if f happens to be linear on T_i, we have the exact equation*

$$\iint_{T_i} f(x,y,z)\,dS = \overline{f}_i\,A(T_i). \tag{7}$$

It follows that our scheme will be exact for every linear integrand, and pretty good for integrands that are approximately linear on each little triangle making up the surface $\mathcal{S}$.

For the general surface integral J, two approximations are active in (4)–(5): first, the approximation of the true smooth surface $\mathcal{S}$ by a collection of triangles; second, the approximation of the true smooth function f by a different linear approximation on each little triangle.

Representing a Surface. We met one standard data structure for a triangulated mesh surface in Computer Lab 2. In general, it involves a list of vertex coordinates presented using three column vectors `X`, `Y`, `Z`, and a matrix `FACELIST` that tells which vertices make up each triangle. It's an effective choice for this task as well.

A Strong Start. A key product of Computer Lab 2 was a script to calculate the area of a triangulated surface given in the workspace variables `X`, `Y`, `Z`, and `FACELIST`. In Lab 2 we expressed this as in equation (3) above; the step up from (3) to (4) is not a large one. All we need is another vector `F` with the property that entry `F(i)` tells the value of f at the point `(X(i),Y(i),Z(i))`. And we can test our extension by supplying

```
F = ones(size(X));
```

IDEA 2: *Triangulating a Plane Rectangle.*

The simplest double integrals have a domain that is a rectangle in the (x,y)-plane, like the set $D = \{(x,y) : 1 \le x \le 5,\ 11 \le y \le 13\}$ shaded gray below. (All figures in this writeup are in colour.)

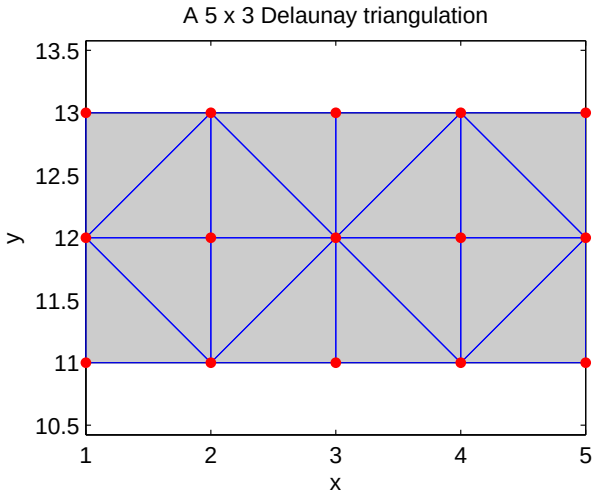

Figure 1: A Simple Rectangle, with Delaunay Triangulation

Overlayed on the rectangle D is a mesh of 16 triangles, with vertices in a simple rectangular array. Let's discuss how to build such a mesh and give it a standard surface representation.

Vertices in Mesh Matrices. Look at the red dots in Figure 1. These are the triangle vertices for the mesh we will build in the next step. Matlab has a built-in command for generating a rectangular grid of vertices like this. Figure 1 shows 15 vertex points, at (x, y) locations that Matlab can generate from given lists of five x-values and three y-values. The `meshgrid` command does this:

```
xnodes = 1:5;                                % specify x-values for nodes in grid
ynodes = 11:13;                              % specify y-values for nodes in grid
[Xmesh,Ymesh]  = meshgrid(xnodes,ynodes);    % Build mesh matrices of x-vals and y-vals
```
(8)

The two 3×5 matrices produced by this sequence of commands are

$$\texttt{Xmesh} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{bmatrix}, \qquad \texttt{Ymesh} = \begin{bmatrix} 11 & 11 & 11 & 11 & 11 \\ 12 & 12 & 12 & 12 & 12 \\ 13 & 13 & 13 & 13 & 13 \end{bmatrix}.$$

Each matrix has the same shape, and for any index pair (i, j), selecting the (i, j) entries from `Xmesh` and `Ymesh` give the (x, y)-coordinates of one of the vertices we want. In detail, `meshgrid` uses the given x-values to make a matrix `Xmesh` with constant *rows*; it repeats the given y-values in a matrix `Ymesh` with constant *columns*. The organization of the matrix entries matches layout of the (x, y)-pairs in our grid.

Reshaping. Our standard data structure wants long lists of vertices, not rectangular arrays. No problem: Matlab's `colon` operator will turn any given matrix into an enormous column vector by stacking the columns on top of each other. Doing this to both `Xmesh` and `Ymesh` preserves the correspondence between x- and y-coordinates. To get vertex lists we can use, and even extend to 3D, we follow the commands in line (8) with

```
X = Xmesh(:);                % Reshape x-coords in mesh into a tall column
Y = Ymesh(:);                % Reshape y-coords in mesh into a tall column
Z = zeros(size(X));          % Generate z-coords of compatible dimension
```
(9)

Triangulating. Matlab supplies a powerful built-in function that will build a triangular mesh whose vertices are N given points in the plane. The function, named `delaunay`, is extremely versatile: its only limitation is that all the given points must be distinct. (Using `delaunay` on a rectangular grid is like using a sledgehammer to open a peanut, but ... it's a free sledgehammer.) To make it work, simply pack the vertex coordinates into two Matlab vectors, `X = [`$x_1, \ldots, x_N$`]` and `Y = [`$y_1, \ldots, y_N$`]`, and say

```
FACELIST = delaunay(X,Y);
```
(10)

The code blocks (8)–(10), in order, produce all the workspace variables we need to describe the mesh of triangles shown in Figure 1 above. To produce that sketch, we could say

```
triplot(FACELIST,X,Y,'b');              % Draw triangle mesh just built in blue
hold on;                                % Set up to add more to this plot
plot(X,Y,'r.','MarkerSize',14);         % Put a red dot at each vertex
```
(11)

IDEA 3: *Adding Altitude.*

Lifting the triangulation from Idea 2 out of the plane $z = 0$ is as simple as providing a nonzero z-coordinate to go with the given nodes in the (x, y)-plane. For example, suppose we are interested in the part of the surface

$$z = \ln(y) - \tfrac{1}{2}(x - 3)^2 \tag{12}$$

that lies above the rectangle $D = [1,5] \times [11,13]$ shaded in Figure 1 above. With the vertices listed in vectors `X` and `Y` from line (9), we could generate corresponding z-coordinates using the vectorized calculation

```
Z = log(Y) - 0.5*(X-3).^2;                % Vectorized calculation uses known X, Y
```
(13)

Now the vectors `X`, `Y`, `Z` describe points on the surface (12), and the edge-building instructions coded in the matrix `FACELIST` need no change to give a flying-triangle approximation to the surface of interest. The following commands produce a picture:

```
trimesh(FACELIST,X,Y,Z,'EdgeColor','m');   % Magenta mesh in space
plot3(X,Y,Z,'k.','MarkerSize',14);         % Put a black dot at each vertex
```

(Note that `triplot` works in the plane, and `trimesh` works in 3D.) The result is shown below

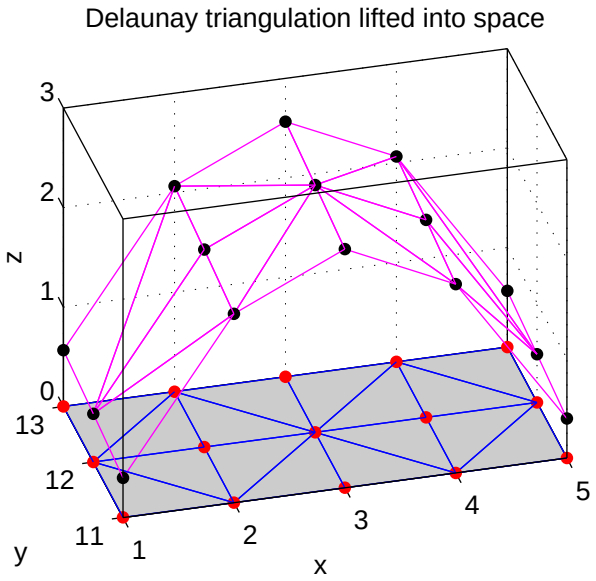

Figure 2: Triangulation of $z = \ln(y) - \frac{1}{2}(x-3)^2$, $(x,y) \in D$

IDEA 4: *Parametric Triangulation of Simple Plane Regions*

Change of Variables. Imagine using the methods above to triangulate a rectangular mesh in a coordinate system whose axes are labelled u and v instead of x and y. Then we could use mathematical functions to map each (u,v) vertex to some new point in the (x,y)-plane. The connection scheme (in matrix `FACELIST`) that identifies the triangles in (u,v)-space would carry forward without change to produce corresponding triangles in (x,y)-space. The simplest definitions, namely,

$$x = u, \qquad y = v,$$

would re-produce exactly the sort of triangulations shown in Idea 3. Let's explore some more interesting options.

A Vertical Ribbon. Imagine a subset of the (x,y)-plane with the form

$$D = \{(x,y) \,:\, a \le x \le b,\ f(x) \le y \le g(x)\}\,. \tag{14}$$

To fill D with triangles, use the ribbon edges to set the u-values, but choose $0 \le v \le 1$:

$$R = [a,b] \times [0,1] = \{(u,v) \,:\, a \le u \le b,\ 0 \le v \le 1\}\,.$$

Clearly a point (x,y) lies in D if and only if it has the form

$$x = u, \quad y = (1-v)f(x) + vg(x) \qquad \text{for some } (u,v) \in R. \tag{15}$$

This provides a recipe for using each point of the reference rectangle R to generate a point in the desired domain D.

Figure 3 below illustrates this procedure for the desired domain

$$D = \{(x,y) \,:\, -1 \le x \le 1,\ |x| - 1 \le y \le \cos(\pi x/2)\}\,.$$

The commands that produced Fig. 3(b) are shown here. Please read and understand every line.

```
unodes = linspace(-1,1,11);                % List of nodes along u-axis:  a=-1, b=1
vnodes = linspace( 0,1,11);                % List of nodes along v-axis
[Umesh,Vmesh] = meshgrid(unodes,vnodes);   % Build mesh matrices
U = Umesh(:); clear Umesh                  % Build vertex columns in (u,v)-space;
V = Vmesh(:); clear Vmesh                  %  discard matrices to free up memory
FACELIST = delaunay(U,V);                  % Hard Work:  join vertices with triangles
X = U;                                     % Horizontal interval is OK as-is
Y = (1-V).*(abs(X)-1) + V.*cos(pi*X/2);    % Apply x-dependent stretch to each vert fibre
triplot(FACELIST,X,Y,'r');                 % Draw plane triangulation in red
hold on; axis equal;                       % Set up for more plotting
plot(X,Y,'r.');                            % Add a dot at each vertex location
```
(16)

The results are in Figure 3(b), below. (Minor decorations have been added.) Each triangle the reference region produces a corresponding triangle in the desired domain.

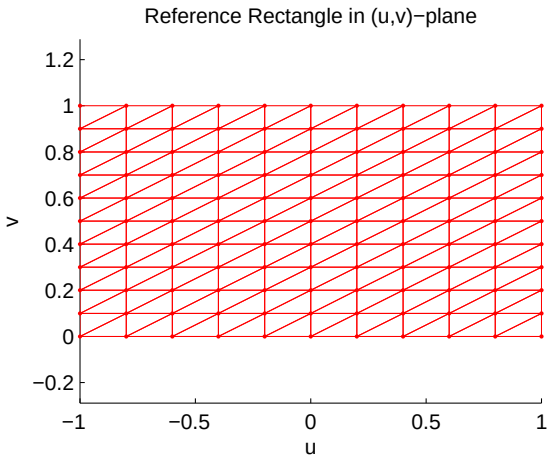

Fig. 3(a): Reference Rectangle $R = [-1,1] \times [0,1]$

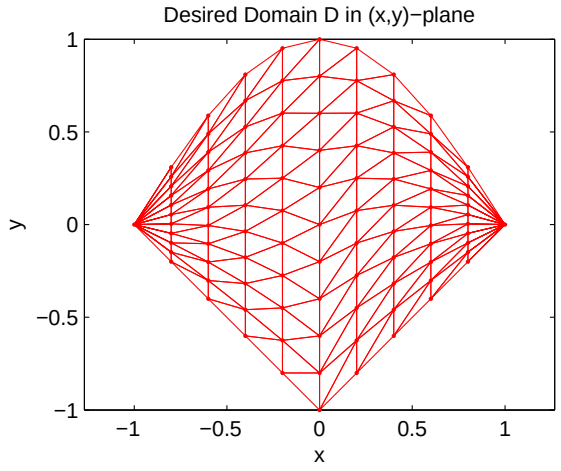

Fig. 3(b): Desired Domain

Fine points. In Fig. 3(a), there are 11 vertices with $u = -1$ on the left edge of the reference rectangle R. Each of these 11 points gets mapped onto the same target, namely, the left corner of D at $(x,y) = (-1,0)$. So the `delaunay` triangle-generating command would probably fail if we asked it to work on the vectors `X` and `Y`. It was smart to build the triangles before mapping them.

A Horizontal Ribbon. Similar ideas can be used to triangulate a set of the form

$$D = \{(x, y) \,:\, p(y) \le x \le q(y),\ c \le y \le d\}\,. \tag{17}$$

We would start with a reference rectangle $R = [0, 1] \times [c, d]$ whose vertical dimensions match the ribbon wrapping D, then match $y = v$, and finally choose x to take on some value between $p(y)$ and $q(y)$ depending on the value of u. Expressions analogous to (15) and code similar to (16). *Try to derive and write them out now, because they will be needed in Computer Lab 3.*

Adding altitude. Many surfaces in 3D can be described in the form

$$\mathcal{S} = \{(x, y, z) \,:\, (x, y) \in D,\ z = f(x, y)\}\,,$$

where D is some given subset of the (x, y)-plane and f is some given function of two variables. If we have a triangulation for the domain D, it is very easy to "lift it" and produce a triangulation for the surface: we just take the vertex lists `X`, `Y` for points in D and associate with them the z-coordinates defined by `Z = f(X,Y)`. (The function f must be correctly "vectorized" for this to work.) To illustrate, let's look at the piece of $z = x^2 + y^2$ lying above the set D suggested in Fig. 3 above. In detail, this is the surface

$$\mathcal{S} = \left\{(x, y, z) \,:\, -1 \le x \le 1,\ |x| - 1 \le y \le \cos(\pi x/2),\ z = x^2 + y^2\right\}.$$

We can sketch it by adding just a few commands to the block of code above:

```
Z = X.^2 + Y.^2;              % Vectorized calculation of z-vals
trisurf(FACELIST,X,Y,Z,Z);    % Second 'Z' is for colour map
triplot(FACELIST,X,Y);        % Draw mesh on plane for emphasis
```

Here are the results (lightly decorated):

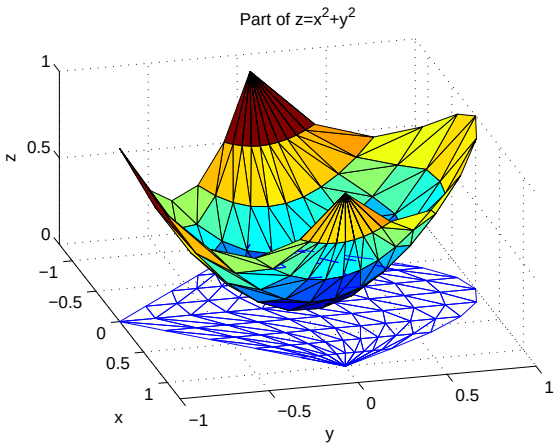

Fig. 4(a): Graph surface $\mathcal{S}$

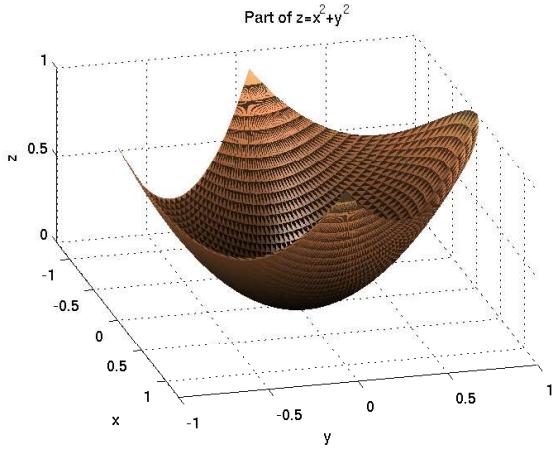

Fig. 4(b): More triangles ⇒ nicer picture!

APPENDIX [OPTIONAL]: *Order of Accuracy and Richardson Extrapolation.*

Approximate Error Analysis. This material should be familiar from MECH 221. Let $J(h)$ denote the computed approximation to a surface integral that we get when we use a triangulation made of triangles with characteristic length h. This is defined for many small values of $h > 0$, but not when $h = 0$. Let's *define* $J(0)$ to be the theoretical exact value of the true surface integral. Our approximation scheme is said to have "order p" if there is some constant C such that

$$J(h) \approx J(0) + Ch^p \qquad \text{for } 0 < h \ll 1. \tag{A1}$$

The exponent p can be predicted analytically, and observed in computational experiments. It is a characteristic of the method, the same for all reasonable choices of $\mathcal{S}$ and f; the coefficient C, however, typically varies from one situation to another.

Finding the Order. To find p experimentally, we start with a problem where $J(0)$ is known. Then we calculate $J(h)$ for several values of h, gathering several input-output pairs for the function

$$E(h) \stackrel{\text{def}}{=} J(h) - J(0) \approx Ch^p. \tag{A2}$$

(The scalar $E(h)$ tells the error associated with a grid of "width" h.) Taking logarithms of the magnitudes on both sides in (4) gives

$$\log|E(h)| \approx \log|C| + p\log(h). \tag{A3}$$

Thus the relationship between $y = \log|E(h)|$ and $x = \log(h)$ has the form

$$y \approx px + \log|C|.$$

Plotting y vs x should produce a straight line whose slope equals the method's order, p. So we transform our observed pairs of $(h, E(h))$-values into pairs of the form $(x, y) = (\log(h), \log|E(h)|)$, and then plot the latter. If they lie on or near a straight line, then the approximations in (A1)–(A3) are credible and the line's slope reveals p.

Exploiting A Known Order. Once we know p in (A1), we can use it to extrapolate our calculations even when the exact value $J(0)$ is unknown. We first rearrange (A1) as

$$\frac{J(h) - J(0)}{h^p} \approx C \qquad \text{for } 0 < h \ll 1.$$

Then we take two numerical results, with grid spacings h_1 and h_2, and compare

$$\frac{J(h_1) - J(0)}{h_1^p} \approx \frac{J(h_2) - J(0)}{h_2^p}.$$

Rearranging this gives an approximate formula for $J(0)$ that we can evalulate using only the known numbers h_1, h_2, $J(h_1)$, and $J(h_2)$. This process is known as **Richardson Extrapolation.**