Lecture 16

Interpolation in 2D

1 Introduction

The two-dimensional (2D) interpolation problem is as follows. We are given n samples (x_i, y_i, z_i) assumed to be drawn from some function $z = f(x, y)$. How can we best estimate $z = f(x, y)$ for arbitrary x and y values?

In many practical cases our samples are drawn from a *uniform rectangular grid* which allows us to separate the bookkeeping for x and y values. A digital photograph, with its distinct rows and columns, is an example. We will limit consideration to this case and restate the problem as follows. Given $n \cdot m$ samples of a 2D function

$$z_{ij} = f(x_i, y_j), \; i = 1,2,\ldots,m, \; j = 1,2,\ldots,n \tag{1}$$

where

$$\begin{aligned} x_i &= x_1 + (i-1)\Delta x \\ y_j &= y_1 + (j-1)\Delta y \end{aligned} \tag{2}$$

how can we best estimate $z = f(x, y)$ for arbitrary x,y values?

2 Bookkeeping

For a uniform rectangular grid a point (x, y) will fall inside of a single rectangle $x_i \le x < x_{i+1}, y_j \le y < y_{j+1}$ as shown in Fig. 1. We will call each of these rectangles a *unit cell*. Even more than in the 1D case, it is convenient to define *normalized* coordinates

$$u = \frac{x - x_i}{x_{i+1} - x_i}, \; v = \frac{y - y_j}{y_{j+1} - y_j} \tag{3}$$

so that $x_i \le x < x_{i+1}, y_j \le y < y_{j+1}$ corresponds to $0 \le u < 1, 0 \le v < 1$.

As shown in Fig. 1, the distance $x - x_1$ can be broken up into an integer number of Δx steps plus a fractional Δx step

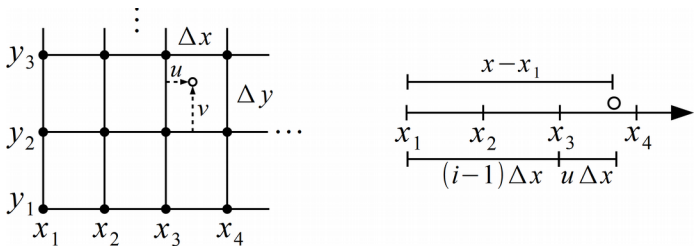


Fig. 1 Left: A point (x, y) will fall within a unit cell, in the case shown cell $i=3$, $j=2$. Position in the cell is specified by (u, v). Right: The integer part of $(x - x_1)/\Delta x$ determines i while the fractional part determine u.

$$x - x_1 = (i-1)\Delta x + u\,\Delta x \tag{4}$$

Rearranging we have

$$(i-1) + u = \frac{x - x_1}{\Delta x} \tag{5}$$

where $(i-1)$ is an integer and $0 \le u < 1$ is a fractional value. In Scilab the command `int(z)` returns the integer part of a real number. Given an *x* value we can solve for *i* and *u* using

$$i = 1 + \text{int}\left(\frac{x - x_1}{\Delta x}\right)\ ,\ u = \frac{x - x_1}{\Delta x} + 1 - i \tag{6}$$

Likewise in the *y* direction we have

$$j = 1 + \text{int}\left(\frac{y - y_1}{\Delta y}\right)\ ,\ v = \frac{y - y_1}{\Delta y} + 1 - j \tag{7}$$

Function `interpUnitCell` in the Appendix implements this bookkeeping. Below, we will assume that the values *i,j,u,v* have been calculated for the given (x, y) at which we are computing the interpolation.

3 Nearest neighbor interpolation

This is an obvious extension of the 1D case. We find the grid point closest to (x, y) and use the *z* value at that grid point as our interpolation. That grid point will be one of the corners of the unit cell. With our bookkeeping this reads

Nearest-neighbor interpolation

if $u \le 0.5$ then $k = i$ else $k = i + 1$

if $v \le 0.5$ then $l = j$ else $l = j + 1$

$z = z_{kl}$

This interpolation is piecewise constant and discontinuous. Function `interpNeighbor2D` in the appendix implements this algorithm.

4 Linear interpolation over triangles

A 2D linear function has the form

$$z = f(u, v) = a + b\,u + c\,v \tag{8}$$

and defines a plane in 3D space. A plane cannot pass through four arbitrary points, so a function of this form cannot represent $f(x, y)$ over a unit cell with four corners. However, we can divide a unit cell into two triangles, as shown in Fig. 2. Each of those triangles has three vertices through which we can pass a plane. For the "lower" triangle ($v \le u$) this requires

$$\begin{aligned} z_{i,j} &= a \\ z_{i+1,j} &= a + b \\ z_{i+1,j+1} &= a + b + c \end{aligned} \tag{9}$$

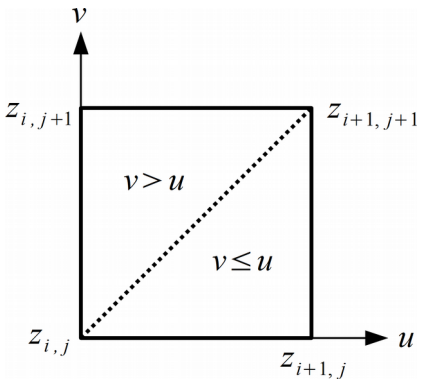


Fig. 2: Dividing a unit cell into two triangles.

corresponding to the three vertices $(u,v)=(0,0),(1,0),(1,1)$. The solution is

$$\begin{aligned} a &= z_{i,j} \\ b &= z_{i+1,j} - z_{i,j} \\ c &= z_{i+1,j+1} - z_{i+1,j} \end{aligned} \tag{10}$$

For the "upper" triangle ($v > u$) the equations are

$$\begin{aligned} z_{i,j} &= a \\ z_{i,j+1} &= a + c \\ z_{i+1,j+1} &= a + b + c \end{aligned} \tag{11}$$

corresponding to the three vertices $(u,v)=(0,0),(0,1),(1,1)$. The solution is

$$\begin{aligned} a &= z_{i,j} \\ c &= z_{i,j+1} - z_{i,j} \\ b &= z_{i+1,j+1} - z_{i,j+1} \end{aligned} \tag{12}$$

The result is summarized as follows.

Linear interpolation over triangles

if $v \le u$ then $z = z_{i,j} + u(z_{i+1,j} - z_{i,j}) + v(z_{i+1,j+1} - z_{i+1,j})$

else $z = z_{i,j} + v(z_{i,j+1} - z_{i,j}) + u(z_{i+1,j+1} - z_{i,j+1})$

This form of linear interpolation is a key component of the *Finite Element Method* (FEM) for solving partial differential equations. In that case, the z_{ij} samples values are treated as unknowns whose values are solved by requiring the interpolation to satisfy some equations describing the physics of the system. In the FEM the triangles are called *finite elements*. Function `interpTriangle` in the appendix implements this algorithm.

5 Bilinear interpolation

The linear function (8) cannot pass through the four corners of a unit cell. However a *bilinear* function

$$z = f(u,v) = a + b\,u + c\,v + d\,uv$$

can with proper choice of the coefficients *a,b,c,d*. The equations are

$$\begin{aligned} z_{i,j} &= a \\ z_{i+1,j} &= a+b \\ z_{i,j+1} &= a+c \\ z_{i+1,j+1} &= a+b+c+d \end{aligned} \tag{13}$$

corresponding to the four corners $(u,v)=(0,0),(1,0),(0,1),(1,1)$. The solution is

$$\begin{aligned} a &= z_{i,j} \\ b &= z_{i+1,j}-z_{i,j} \\ c &= z_{i,j+1}-z_{i,j} \\ d &= z_{i+1,j+1}+z_{i,j}-z_{i+1,j}-z_{i,j+1} \end{aligned} \tag{14}$$

and we have the algorithm

Bilinear interpolation

$$z = z_{i,j}+(z_{i+1,j}-z_{ij})u+(z_{i,j+1}-z_{i,j})v+(z_{i+1,j+1}+z_{i,j}-z_{i+1,j}-z_{i,j+1})uv$$

A bilinear function

$$a+b\,x+c\,y+d\,xy \tag{15}$$

is linear in *x* if *y* is held constant, say $y=y_0$

$$(a+c\,y_0)+(b+d\,y_0)x \tag{16}$$

and is linear in *y* if *x* is held fixed, say $x=x_0$

$$z=(a+b\,x_0)+(c+d\,x_0)\,y \tag{17}$$

Indeed one way to think of bilinear interpolation is illustrated in Fig.3. First perform linear interpolation in *u* along the top and bottom sides of the cell to get the values

$$\begin{aligned} z_j &= z_{i,j}+(z_{i+1,j}-z_{i,j})u \\ z_{j+1} &= z_{i,j}+(z_{i+1,j}-z_{i,j})u \end{aligned} \tag{18}$$

at the locations marked X. Now perform linear interpolation in *v* between those values to get

$$z=z_j+(z_{j+1}-z_j)v \tag{19}$$

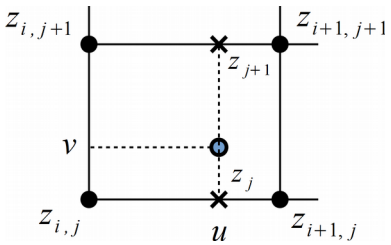


Fig.3:Bilinear interpolation can be thought of as a series of 1D linear interpolations.

Substituting (18) into (19) and do a bit of algebra results in the bilinear interpolation formula.

This idea is easily extended into 3 or more dimensions, and Scilab provides a function for performing this calculation. In 2D we execute

```
[xp,yp] = ndgrid(xx,yy);
zp = linear_interpn(xp,yp,x,y,z);
```

where *x* and *y* are the arrays of sample x_i, y_j values and *z* is the array of sample $z_{i,j}$ values. The (two-dimensional) arrays `xp, yp` are the coordinates of the points were we want interpolated values and `zp` is the array of those values. In this example we used the `ndgrid` function to convert one-dimensional arrays `xx` and `yy` into two-dimensional arrays `xp` and `yp`. Function `interpBilinear` in the appendix implements this algorithm.

6 Bicubic interpolation

In the 1D case piecewise cubic interpolation offered dramatic improvements over piecewise linear interpolation at the expense of extra calculation and bookkeeping. We might expect a similar improvement in the 2D case. In 2D we have to consider *bicubic* functions over rectangles in the *x,y* plane. Just as a bilinear expression has a "cross term" *uv*, a bicubic expression has various cross terms of the form

$$\begin{matrix} 1 & v & v^2 & v^3 \\ u & uv & uv^2 & uv^3 \\ u^2 & u^2v & u^2v^2 & u^2v^3 \\ u^3 & u^3v & u^3v^2 & u^3v^3 \end{matrix}$$

There are sixteen such terms, and each requires its own coefficient to form the bicubic interpolation

$$z = f(u,v) = \sum_{k=1}^{4}\sum_{l=1}^{4} c_{kl} u^{k-1} v^{l-1} \tag{20}$$

There are four corners to a rectangle (Fig. 2), so we need four equations at each corner to get a total of sixteen equations in sixteen unknowns. One way to obtain these equations is to specify the four values

$$z, \frac{\partial z}{\partial x}, \frac{\partial z}{\partial y}, \frac{\partial^2 z}{\partial x \partial y}$$

at each sample point. Although the bookkeeping is messy, it's then straight-forward to set up equations to solve for the coefficients c_{kl} for each rectangle.

As in 1D, in 2D we most often do not have known derivative values at the sample points, so we must estimate these somehow. A simple approach is to approximate the derivatives using *central differences* of the *z* values (we will study central differences in the numerical derivatives lecture). The *x* and *y* partial derivatives are approximated by

$$\frac{\partial z}{\partial x} \approx \frac{z(x+\Delta x, y) - z(x-\Delta x, y)}{2\Delta x} = \frac{z_{i+1,j} - z_{i-1,j}}{2\Delta x} \tag{21}$$

and

$$\frac{\partial z}{\partial y} \approx \frac{z(x, y+\Delta y)-z(x, y-\Delta y)}{2\Delta y} = \frac{z_{i, j+1}-z_{i, j-1}}{2\Delta x} \tag{22}$$

The cross derivative is approximated by

$$\frac{\partial^2 z}{\partial x \partial y} \approx \frac{1}{(2\Delta y)(2\Delta x)}\left[z_{i+1, j+1}+z_{i-i, j-1}-z_{i-1, j+1}-z_{i+1, j-1}\right] \tag{23}$$

Another approach is to pass 1D splines through the z data in various ways and use those splines to estimate the derivative values. This leads to the idea of *bicubic splines*. In Scilab we perform bicubic spline interpolation as follows

```
[xp,yp] = ndgrid(xx,yy);
zp = interp2d(xp,yp,x,y,splin2d(x,y,z));
```

The `x,y,z,xp,yp,zp` arrays are the same as in `linear_interpn` above. The `splin2d` function calculates the c_{kl} coefficients for each rectangle which then becomes the last argument of the `interp2d` function.

7 Lanczos interpolation

The Lanczos interpolation method readily translates from 1D to 2D. In 2D we write

$$z = \sum_{k=i+1-a}^{i+a} \sum_{l=j+1-a}^{j+a} z_{k,l}\, L\left(\frac{x-x_k}{\Delta x}\right) L\left(\frac{y-y_l}{\Delta y}\right) \tag{24}$$

where as before

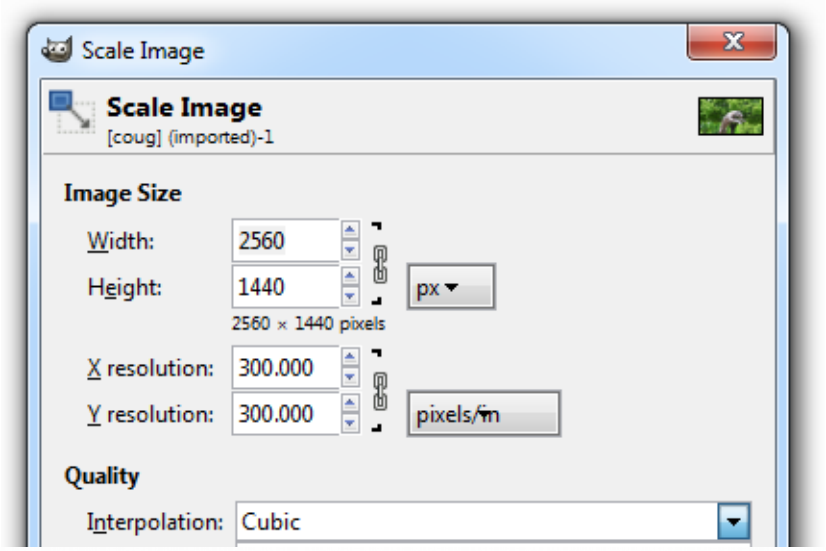
$$L(x) = \begin{cases} \operatorname{sinc}(x)\operatorname{sinc}\left(\dfrac{x}{a}\right) & |x| \le a \\ 0 & |x| > a \end{cases} \tag{25}$$

and

$$\operatorname{sinc}(x) = \frac{\sin \pi x}{\pi x} \tag{26}$$

Typically $a = 3$ is used. In (24) we take $z_{k,l} = 0$ if either k or l falls outside the grid. Function `interpLanczos2D` in the appendix implements this algorithm.

The various 2D interpolation methods we have looked at are commonly used for image resizing ("resampling" is just a form of interpolation) in graphics manipulation programs such as Photoshop and Gimp. Fig. 4 shows a screen shot of the Interpolation menu of Gimp. Finally Fig. 5 compares interpolation performed by the various methods we have been discussing.



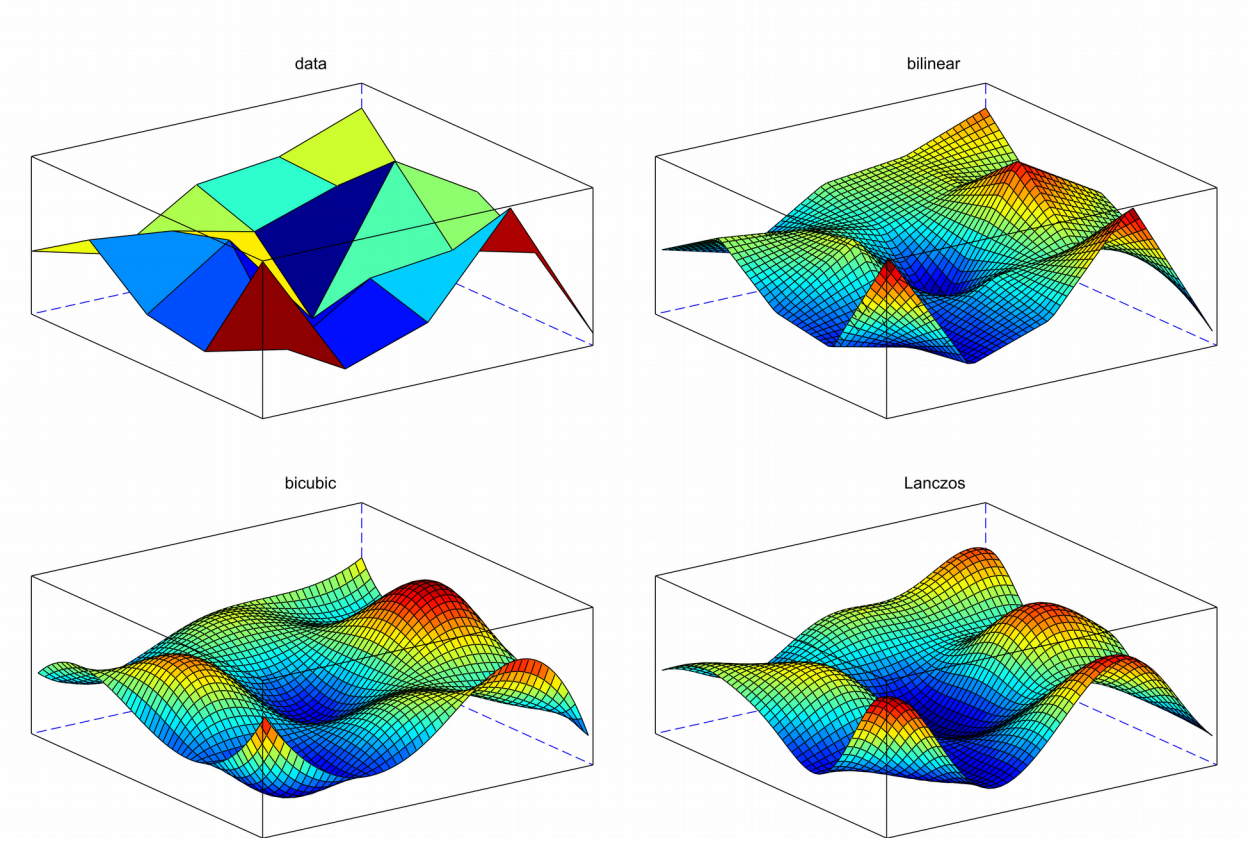


Fig. 5: Comparison of various 2D interpolation methods.

8 Appendix – Scilab code

8.1 unit cell bookkeeping

```
0001  ////////////////////////////////////////////////////////////////////
0002  // interpUnitCell.sci
0003  // 2014-11-07, Scott Hudson, for pedagogic purposes
0004  // Given sample locations x(i),y(j) 1<=i<=m , 1<=j<=n
0005  // with x and y sorted in ascending order x(1)<x(2)<x(3) etc.
```

```
0006  // and a interpolation location xp,yp
0007  // calculate the indices of the unit cell i,j
0008  // and the location u,v within the unit cell
0009  ////////////////////////////////////////////////////////////////////
0010  function [i, j, u, v]=interpUnitCell(x, y, xp, yp);
0011    m = length(x);
0012    n = length(y);
0013    Dx = (x(m)-x(1))/(m-1);
0014    Dy = (y(n)-y(1))/(n-1);
0015    t = (xp-x(1))/Dx;
0016    i = 1+int(t);
0017    u = t+1-i;
0018    t = (yp-y(1))/Dy;
0019    j = 1+int(t);
0020    v = t+1-j;
0021    // if xp or yp are outside the x,y grid, move to the grid edge
0022    if (i>=m)
0023      i = m-1;
0024      u = 1;
0025    elseif (i<=0)
0026      i = 1;
0027      u = 0;
0028    end
0029    if (j>=n)
0030      j = m-1;
0031      v = 1;
0032    elseif (j<=0)
0033      j = 1;
0034      v = 0;
0035    end
0036  endfunction
```

8.2 Nearest-neighbor interpolation

```
0001  ////////////////////////////////////////////////////////////////////
0002  // interpNeighbor2D.sci
0003  // 2014-11-07, Scott Hudson, for pedagogic purposes
0004  // Given samples (x(i),y(j),z(i,j)) 1<=i<=m , 1<=j<=n
0005  // use nearest-neighbor interpolation over triangles to estimate
0006  // zp(k,l) = f(xp(k),yp(l)) 1<=k<=p , 1<=l<=q
0007  // x and y must be sorted in ascending order x(1)<x(2)<x(3) etc.
0008  ////////////////////////////////////////////////////////////////////
0009  function zp=interpNeighbor2D(x, y, z, xp, yp);
0010    m = length(x);
0011    n = length(y);
0012    p = length(xp);
0013    q = length(yp);
0014    zp = zeros(p,q);
0015    Dx = (x(m)-x(1))/(m-1);
0016    Dy = (y(n)-y(1))/(n-1);
0017    for k=1:p
0018      for l=1:q
0019        [i,j,u,v] = interpUnitCell(x,y,xp(k),yp(l));
0020        if (u<=0.5)
0021          kk = i;
0022        else
0023          kk = i+1;
0024        end
0025        if (v<=0.5)
0026          ll = j;
0027        else
0028          ll = j+1;
0029        end
0030        zp(k,l) = z(kk,ll);
0031      end
0032    end
0033  endfunction
```

8.3 Linear interpolation over triangles

```
0001  ////////////////////////////////////////////////////////////////////
0002  // interpTriangle.sci
0003  // 2014-11-07, Scott Hudson, for pedagogic purposes
0004  // Given samples (x(i),y(j),z(i,j)) 1<=i<=m , 1<=j<=n
0005  // use linear interpolation over triangles to estimate
0006  // zp(k,l) = f(xp(k),yp(l)) 1<=k<=p , 1<=l<=q
0007  // x and y must be sorted in ascending order x(1)<x(2)<x(3) etc.
0008  ////////////////////////////////////////////////////////////////////
0009  function zp=interpTriangle(x, y, z, xp, yp);
0010    m = length(x);
0011    n = length(y);
0012    p = length(xp);
0013    q = length(yp);
0014    zp = zeros(p,q);
0015    Dx = (x(m)-x(1))/(m-1);
0016    Dy = (y(n)-y(1))/(n-1);
0017    for k=1:p
0018      for l=1:q
0019        [i,j,u,v] = interpUnitCell(x,y,xp(k),yp(l));
0020        if ((i>=1)&(i<=m-1)&(j>=1)&(j<=n-1))
0021          if (v<=u)
0022            zp(k,l) = z(i,j)+u*(z(i+1,j)-z(i,j))+v*(z(i+1,j+1)-z(i+1,j));
0023          else
0024            zp(k,l) = z(i,j)+v*(z(i,j+1)-z(i,j))+u*(z(i+1,j+1)-z(i,j+1));
0025          end
0026        end
0027      end
0028    end
0029  endfunction
```

8.4 Bilinear interpolation

```
0001  ////////////////////////////////////////////////////////////////////
0002  // interpBilinear.sci
0003  // 2014-11-07, Scott Hudson, for pedagogic purposes
0004  // Given samples (x(i),y(j),z(i,j)) 1<=i<=m , 1<=j<=n
0005  // use bilinear interpolation to estimate
0006  // zp(k,l) = f(xp(k),yp(l)) 1<=k<=p , 1<=l<=q
0007  // x and y must be sorted in ascending order x(1)<x(2)<x(3) etc.
0008  ////////////////////////////////////////////////////////////////////
0009  function zp=interpBilinear(x, y, z, xp, yp);
0010    m = length(x);
0011    n = length(y);
0012    p = length(xp);
0013    q = length(yp);
0014    zp = zeros(p,q);
0015    Dx = (x(m)-x(1))/(m-1);
0016    Dy = (y(n)-y(1))/(n-1);
0017    for k=1:p
0018      for l=1:q
0019        [i,j,u,v] = interpUnitCell(x,y,xp(k),yp(l));
0020        if ((i>=1)&(i<=m-1)&(j>=1)&(j<=n-1))
0021          zp(k,l) = z(i,j)+u*(z(i+1,j)-z(i,j))+v*(z(i,j+1)-z(i,j))..
0022          +u*v*(z(i+1,j+1)+z(i,j)-z(i+1,j)-z(i,j+1));
0023        end
0024      end
0025    end
0026  endfunction
```

8.5 Lanczos interpolation

```
0001  ////////////////////////////////////////////////////////////////////
0002  // interpLanczos2D.sci
0003  // 2014-11-07, Scott Hudson, for pedagogic purposes
0004  // Given samples (x(i),y(j),z(i,j)) 1<=i<=m , 1<=j<=n
0005  // use Lanczos3 interpolation to estimate
0006  // zp(k,l) = f(xp(k),yp(l)) 1<=k<=p , 1<=l<=q
0007  // x and y must be sorted in ascending order x(1)<x(2)<x(3) etc.
0008  ////////////////////////////////////////////////////////////////////
0009  function zp=interpLanczos2D(x, y, z, xp, yp);
0010    m = length(x);
0011    n = length(y);
0012    p = length(xp);
0013    q = length(yp);
0014    zp = zeros(p,q);
0015    Dx = (x(m)-x(1))/(m-1);
0016    Dy = (y(n)-y(1))/(n-1);
0017    a = 3;
0018
0019    function w=L(z) //Lanczos kernel
0020      if abs(z)<1e-6
0021        w = 1;
0022      elseif (abs(z)>=a)
0023        w = 0;
0024      else
0025        w = sin(%pi*z)*sin(%pi*z/a)/(%pi^2*z^2/a);
0026      end
0027    endfunction
0028
0029    for k=1:p
0030      for l=1:q
0031        [i,j,u,v] = interpUnitCell(x,y,xp(k),yp(l));
0032        for kk=i+1-a:i+a
0033          if ((kk>=1)&(kk<=m))
0034            hx = L((xp(k)-x(kk))/Dx);
0035            for ll=j+1-a:j+a
0036              if ((ll>=1)&(ll<=n))
0037                hy = L((yp(l)-y(ll))/Dy);
0038                zp(k,l) = zp(k,l)+z(kk,ll)*hx*hy;
0039              end
0040            end
0041          end
0042        end
0043      end
0044    end
0045  endfunction
```