R.MAPCALC
An Algebra for GIS and Image Processing

Michael Shapiro and Jim Westervelt
U.S. Army Construction Engineering Research Laboratory
Champaign, Illinois 61801, U.S.A.

March 24, 1992

Abstract

This paper describes a map algebra language that can form the foundation for overlay and neighborhood analyses in both raster geographic information systems and image processing systems. The language combines maps and images in expressions with arithmetic and logical operators and mathematical functions to produce new maps and images. This algebra has been implemented in the GRASS *r.mapcalc* module. The language syntax is first described and then examples relevant to GIS and image processing applications are presented.

1 Introduction

1.1 Background

Geographic information systems (GIS) have been evolving from basic spatial data storage, retrieval, and display systems to more powerful spatial modeling systems. This transition is mainly due to more robust query capabilities. The creation of spatial query languages that empower the end-user to design and create spatial models will advance this development even further. Tomlin (1990) describes a spatial modeling language for raster data with three major classes of map operations: functions of individual cells (e.g. map recoding and Boolean overlay); functions of cells within neighborhoods (e.g. filters and slope-aspect generation); and functions of cells within regions or zones (e.g. area calculations). This classification could be applied to image processing operations as well because image processing systems, which are inherently raster-based, employ analysis methods that are very simliar to some of these GIS operations (Burrough 1986).

1.2 Purpose

This document describes a map algebra language, that is flexible enough to accommodate many operations in Tomlin's first two classes and allows both developers and end-users to construct a solid core of GIS analysis and image processing operations.

1.3 Approach

In the following material, the language syntax is first described and then illustrated with examples relevant to GIS and image processing applications.

1.4 Mode of Technology Transfer

The map algebra has been implemented in the the Geographic Resources Analysis Support System (GRASS)[1] module *r.mapcalc*. Many of the examples employ the Spearfish data base, which is distributed with the GRASS software, and can be reproduced by users of GRASS.

---

[1] GRASS is a raster based GIS and image processing system developed at the Army Corps of Engineers Construction Engineering Research Laboratory.

2 Language Syntax

The algebra employs mathematical operators and functions in expressions involving maps or images. The maps and images, which are stored in raster grid-cell format, are two-dimensional matrices of integer values.

The syntax for the algebra is *result=expression*, where *expression* is built using *maps* and *images*, mathematical *operators*, *functions*, and temporary *variables.* The *result* map is produced by evaluating the expression for each cell in the matrix.

For example, the expression

$$sum = map1 + map2$$

would produce a map *sum* where each cell is the sum of the values of the corresponding cells in *map1* and *map2.*

2.1 Maps and Images

Maps and images are data base files stored in raster format, i.e., two-dimensional matrices of integer values. The values stored in a map may represent categorical data (e.g., soil types), or continuous data (e.g., elevation). The values stored in images usually represent continuous data (e.g., satellite sensor data). Although the information represented by the values within a map or an image varies, the data format is the same. For this reason, the distinction between maps and images will be dropped and the term *map* will be used for either.

Map naming rules are quite flexible. A map name must not contain any special characters (operators, parentheses, etc.), and must be distinguishable from numbers. Common names include *elevation, soils, vegetation, roads, band.1*, etc., although *12oct89* is also acceptable.

Maps may be followed by a *neighborhood* modifier that specifies a relative offset from the current cell being evaluated. The format is *map[r,c]*, where *r* is the row offset and *c* is the column offset. For example, *map[1,2]* refers to the cell one row below and two columns to the right of the current cell, *map[-2,-1]* refers to the cell two rows above and one column to the left of the current cell, and *map[0,1]* refers to the cell one column to the right of the current cell. This syntax permits the development of neighborhood-type filters within a single map or across multiple maps.

At present, GRASS map files may contain only integer values. However, GRASS does permit floating point values to be associated with each integer value in a map.[2]

---

[2] These values are stored in a separate attribute file known as the category label file.

Maps may be prefixed by the @ modifier that translates the integer values in a map to their associated floating point values.

2.2 The Operators

The operators for the map algebra are:

operator	meaning	type	precedence
*	multiplication	arithmetic	4
/	division	arithmetic	4
%	modulus	arithmetic	4
+	addition	arithmetic	3
−	subtraction	arithmetic	3
==	equal	comparison	2
!=	not equal	comparison	2
<	less than	comparison	2
<=	less than or equal	comparison	2
>	greater than	comparison	2
>=	greater than or equal	comparison	2
&&	and	logical	1
||	or	logical	1

The operators are applied from left to right, with those of higher precedence applied before those of lower precedence. Parentheses may be used to control the order of evaluation. The arithmetic operations have their usual meanings, except that division by zero equals zero. If both operands are integer the result is integer, otherwise the result is floating point. The comparisons evaluate to 1 if the comparison is true, otherwise they evaluate to 0. The || operation evaluates to 1 if either operand is non-zero (true) and to 0 otherwise. The && operation evaluates to 1 only if both operands are non-zero (true) and to 0 otherwise.

2.3 The Functions

Functions perform some specific operation on a list of expressions and result in a single operand. The *r.mapcalc* algebra includes the following functions:

abs(x)	$\lvert x \rvert$
exp(x)	e^y
exp(x,y)	x^y
log(x)	$\ln x$
log(x,y)	$\log_y x$
sqrt(x)	$\sqrt{x}$
atan(x)	$\tan^{-1} x \; [-90°, 90°]$
atan(x,y)	$\tan^{-1} y/x \; [0°, 360°]$
cos(x)	$\cos x°$
sin(x)	$\sin x°$
tan(x)	$\tan x°$
min(x,y,...)	minimum of $\{x, y, ...\}$
max(x,y,...)	maximum of $\{x, y, ...\}$
float(x)	convert x to floating point
round(x)	round x to nearest integer
int(x)	convert x to largest integer less than or equal to x
if(x)	1, if x != 0; 0 otherwise
if(x,y)	y, if x != 0; 0 otherwise
if(x,y,z)	y, if x != 0; z otherwise
if(x,p,z,n)	p, if $x > 0$; z, if $x == 0$; n otherwise
eval(a,b,...,x)	x (but all arguments are evaluated)

2.4 Temporary Variables

Sometimes expressions can become complicated and values that are computed in one part of the expression must be used in a later part. A value can be assigned to a temporary *variable* that is then used in a later part of the expression. For example, *result=(map+2)*(map+2)* may be expressed, using a temporary variable x, as *result=(x=map+2)*x*. Additional examples are given on pages 8 and 12.

3 GIS Examples

The algebra supports a variety of GIS operations. A number of these operations will be described, but it should be noted that these examples do not exhaust the flexibility and power of the language. The following maps are used:

elevation 30-meter digital elevation, range 1066-1840 meters

rushmore Camp Rushmore (a fictitious military installation)

0	outside the installation
1	inside the installation

slope slope in degrees

vegcover vegetation cover

1	irrigated agriculture
2	rangeland
3	coniferous forest
4	deciduous forest
5	mixed forest
6	disturbed

3.1 Map Recoding

A standard GIS operation is the recoding of values in a map. For example, a map of forest versus non-forest can be created from the *vegcover* map by recoding forest categories to 1 and non-forest categories to 2 as follows:

$$\begin{aligned} simple.cover \;=\; & if(vegcover == 3 \;||\; vegcover == 4 \;||\; vegcover == 5,\; 1) + \;\backslash \\ & if(vegcover == 1 \;||\; vegcover == 2 \;||\; vegcover == 6,\; 2) \end{aligned}$$

Note that it makes sense to add the results of both *if* functions because each *if* selects a distinct subset of the input data. Also, note the use of \ to indicate that the expression continues on multiple lines.

To make a map of elevations between 1200 and 1375 meters (with elevations outside this range recoded to 0):

$$upland = if(elevation >= 1200 \;\&\&\; elevation <= 1375, elevation)$$

3.2 Selections

An operation related to map recoding is the selection or identification of cells that meet specified criteria in one or more maps. Selection differs from recoding in two respects: (1) it may involve more than one map and (2) the resulting map often has only the values 0 and 1; the value 1 indicates cells that meet the criteria and the value 0 indicates cells that do not. The logical operators (&& and ||) together with the comparison operators (==, !=, <, <=, >, and >=) provide this selection capability.

For example, to create the map *upland*, where elevations between 1200 and 1375 meters are coded as 1 and elevations outside this range are coded as 0:

$$upland = elevation >= 1200 \;\&\&\; elevation <= 1375$$

To create the map *forest*, indicating where forests are found:

$$forest = vegcover == 3 \;||\; vegcover == 4 \;||\; vegcover == 5$$

Then, to create the map *upland.forest*, indicating where upland forests are found:

$$upland.forest = forest \;\&\&\; upland$$

Of course, the map *upland.forest* can be created directly without first creating the maps *upland* and *forest* as follows:

$$\begin{aligned} upland.forest &= (elevation >= 1200 \;\&\&\; elevation <= 1375) \;\&\& \;\; \backslash \\ &\quad (vegcover == 3 \;||\; vegcover == 4 \;||\; vegcover == 5) \end{aligned}$$

3.3 Region Growing

Region growing is a local neighborhood-type operation that adds a one-cell border around regions in a map. This operation can be implemented with a simple algorithm. Each zero value in the map is replaced by a non-zero value from the cell to the left, to the right, above, or below.

For example, to add a one-cell border to the Camp Rushmore map:

$$\begin{aligned} rushmore.grow = \; & if(rushmore, rushmore, & \backslash \\ & \quad if(rushmore[0,-1], rushmore[0,-1], & \backslash \\ & \quad\quad if(rushmore[0,1], rushmore[0,1], & \backslash \\ & \quad\quad\quad if(rushmore[-1,0], rushmore[-1,0], & \backslash \\ & \quad\quad\quad\quad if(\mathrm{rushmore[1,0]}, rushmore[1,0]))))) & \end{aligned}$$

The border itself can now be extracted simply by subtracting the original map from the new map:

$$rushmore.border = rushmore.grow - rushmore$$

Note that a count of just the border cells can be used to approximate the perimeter of the installation.[3]

3.4 Slope and Aspect

The neighborhood syntax of the algebra allows the determination of slope gradient (maximum rate of change in altitude) and slope aspect (compass direction of the slope) from elevation. The basic formulas (Dozier and Strahler 1983) are:

$$\begin{aligned} tan(slope) &= \sqrt{(\delta z/\delta x)^2 + (\delta z/\delta y)^2} \\ tan(aspect) &= \frac{\delta z/\delta x}{\delta z/\delta y} \end{aligned}$$

where $\delta z/\delta x$ and $\delta z/\delta y$ are the partial derivatives in the east-west and north-south directions. Numerical methods can be used to estimate these derivatives (Skidmore 1989). A method given by Horn (1981) is:

$$\begin{aligned} [\delta z/\delta x]_{y,x} &= \left(z_{y-1,x-1} + 2z_{y,x-1} + z_{y+1,x-1} - z_{y-1,x+1} - 2z_{y,x+1} - z_{y+1,x+1}\right)/8\Delta x \\ [\delta z/\delta y]_{y,x} &= \left(z_{y-1,x-1} + 2z_{y-1,x} + z_{y-1,x+1} - z_{y+1,x-1} - 2z_{y+1,x} - z_{y+1,x+1}\right)/8\Delta y \end{aligned}$$

where $z_{y,x}$ is the elevation value at row y column x, Δx is the east-west (i.e., column) grid spacing, and Δy is the north-south (i.e., row) grid spacing. This can be expressed more clearly with a matrix of coefficients:

$$\delta z/\delta x = \frac{\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}}{8\Delta X} \quad \delta z/\delta y = \frac{\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}}{8\Delta Y}$$

The *elevation* map, which has a 30-by-30-meter horizontal grid spacing, can be processed with *r.mapcalc* as follows:

---

[3] Cell counting, a regional operation, is not supported by *r.mapcalc*. Cell counts must be determined using another tool (e.g., the GRASS *r.stats* command).

$$
\begin{array}{rcllll}
slope & = & eval(& x = & (elevation[-1,-1] + 2 * elevation[0,-1] + elevation[1,-1] & \backslash \\
 & & & & -elevation[-1,1] - 2 * elevation[0,1] - elevation[1,1] & \backslash \\
 & & & &)/(8.0 * 30.0) \ , & \backslash \\
 & & & y = & (elevation[-1,-1] + 2 * elevation[-1,0] + elevation[-1,1] & \backslash \\
 & & & & -elevation[1,-1] - 2 * elevation[1,0] - elevation[1,1] & \backslash \\
 & & & &)/(8.0 * 30.0) \ , & \backslash \\
 & & & \multicolumn{2}{l}{atan(sqrt(x * x + y * y))} & \backslash \\
 & &) & & &
\end{array}
$$

$$
\begin{array}{rcllll}
aspect & = & eval(& x = & (elevation[-1,-1] + 2 * elevation[0,-1] + elevation[1,-1] & \backslash \\
 & & & & -elevation[-1,1] - 2 * elevation[0,1] - elevation[1,1] & \backslash \\
 & & & &)/(8.0 * 30.0) \ , & \backslash \\
 & & & y = & (elevation[-1,-1] + 2 * elevation[-1,0] + elevation[-1,1] & \backslash \\
 & & & & -elevation[1,-1] - 2 * elevation[1,0] - elevation[1,1] & \backslash \\
 & & & &)/(8.0 * 30.0) \ , & \backslash \\
 & & & a = & round(atan(x, y)) \ , & \backslash \\
 & & & \multicolumn{2}{l}{if(x \| y, if(a, a, 360))} & \backslash \\
 & &) & & &
\end{array}
$$

Note the use of the temporary variables a, x, and y to capture intermediate results for use in the latter part of the expression.

Slope is calculated using the single-argument version of atan(), which will produce angles in the range 0° to 90°. However, for terrain with low relief, degrees may not give enough information. One modification would be to multiply the result by 10 to get 10ths of a degree. Another would be to omit the atan() function to get tan(slope) instead.

Aspect is calculated using the two-argument version of atan(), which will produce angles in the range 0° to 360°. However, because aspect is undefined if both x and y are zero (i.e. flat terrain), additional care is required. The code:

$$
\begin{array}{l}
a = round(atan(x, y)) \\
if(x \| y, if(a, a, 360))
\end{array}
$$

produces values from 1 to 360 for cells that have non-zero slope and a zero value for flat terrain.

An alternate technique to compute $\delta z / \delta x$ and $\delta z / \delta y$ is used by Frank (1988):

$$\delta z/\delta x = \frac{\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 8 & 0 & -8 & -1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}}{12\Delta X} \quad \delta z/\delta y = \frac{\begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 8 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -8 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 \end{bmatrix}}{12\Delta Y}$$

This is expressed in *r.mapcalc* as:

```
slope  = eval( x = (elevation[0,-2] + 8 * elevation[0,-1]   \
                   -elevation[0,2] - 8 * elevation[0,1]     \
                   )/(12.0 * 30.0) ,                        \
               y = (elevation[-2,0] + 8 * elevation[-1,0]   \
                   -elevation[2,0] - 8 * elevation[1,0]     \
                   )/(12.0 * 30.0) ,                        \
               atan(sqrt(x * x + y * y))                    \
             )

aspect = eval( x = (elevation[0,-2] + 8 * elevation[0,-1]   \
                   -elevation[0,2] - 8 * elevation[0,1]     \
                   )/(12.0 * 30.0) ,                        \
               y = (elevation[-2,0] + 8 * elevation[-1,0]   \
                   -elevation[2,0] - 8 * elevation[1,0]     \
                   )/(12.0 * 30.0) ,                        \
               a = round(atan(x,y)) ,                       \
               if(x||y,if(a,a,360))                         \
             )
```

3.5 Hydrologic Simulation

r.mapcalc can be used to construct a simple hydrologic model that iteratively “flows” water across a landscape. A constant amount of water is first deposited on each cell in the landscape. Then, at each time slice, a portion of the water in each cell is drained into its eight surrounding neighbors. The basic logic is as follows:

```
for each cell
    for each neighbor
        let height difference = (cell elevation + water height)
                    minus (neighbor elevation + water height)
        if the height difference is positive
        then
            if the cell elevation is greater than
                the neighbor's elevation + water height
            then
                drain out a portion of the water
            otherwise
                drain out a portion of the height difference
        otherwise
            if the neighbor's elevation is greater than
                the cell's elevation + water height
            then
                drain in a portion of the neighbor's water
            otherwise
                drain in a portion of the height difference
    end
end
```

The outer loop is for each cell in the landscape and is done automatically by *r.mapcalc*. The inner loop applies to the eight neighboring cells and must be explicitly coded by the user.

Before running this model, the elevation map must be converted to the same units as the water map and filtered to smooth the elevation. In this example, we assume that the water will be in units of 0.1 inches. To convert the elevation from meters to 0.1 inches:

$$elev = elevation * 393.7$$

Then, the map *elev* can be smoothed with the following filter:

$$\frac{\begin{bmatrix} 1 & 2 & 1 \\ 2 & 8 & 2 \\ 1 & 2 & 1 \end{bmatrix}}{20}$$

as follows:

$$\begin{array}{lcll} elev & = & (& elev[-1,-1] + 2 * elev[-1,0] + elev[-1,1] & \backslash \\ & & + & 2 * elev[0,-1] + 8 * elev + 2 * elev[0,1] & \backslash \\ & & + & elev[1,-1] + 2 * elev[1,0] + elev[1,1] & \backslash \\ & &) & /20 & \end{array}$$

(This filter produces a map with smoother contours. It also tends to create dams at terrain choke points, which will form little lakes during the simulation.)

The model itself is coded as follows:

```
water =  water + eval(x = elev + water,                                  \
         if  (x > (y = elev[-1,0] + water[-1,0]),                        \
               -.15 * if  (elev > y,          water,          x - y),    \
                .15 * if  (elev[-1,0] > x,    water[-1,0],    y - x))+   \
         if  (x > (y = elev[1,0] + water[1,0]),                          \
               -.15 * if  (elev > y,          water,          x - y),    \
                .15 * if  (elev[1,0] > x,     water[1,0],     y - x))+   \
         if  (x > (y = elev[0,-1] + water[0,-1]),                        \
               -.15 * if  (elev > y,          water,          x - y),    \
                .15 * if  (elev[0,-1] > x,    water[0,-1],    y - x))+   \
         if  (x > (y = elev[0,1] + water[0,1]),                          \
               -.15 * if  (elev > y,          water,          x - y),    \
                .15 * if  (elev[0,1] > x,     water[0,1],     y - x))+   \
         if  (x > (y = elev[-1,1] + water[-1,1]),                        \
               -.10 * if  (elev > y,          water,          x - y),    \
                .10 * if  (elev[-1,1] > x,    water[-1,1],    y - x))+   \
         if  (x > (y = elev[1,1] + water[1,1]),                          \
               -.10 * if  (elev > y,          water,          x - y),    \
                .10 * if  (elev[1,1] > x,     water[1,1],     y - x))+   \
         if  (x > (y = elev[1,-1] + water[1,-1]),                        \
               -.10 * if  (elev > y,          water,          x - y),    \
                .10 * if  (elev[1,-1] > x,    water[1,-1],    y - x))+   \
         if  (x > (y = elev[-1,-1] + water[-1,-1]),                      \
               -.10 * if  (elev > y,          water,          x - y),    \
                .10 * if  (elev[-1,-1] > x,   water[-1,-1],   y - x)))
```

The result map *water* represents the water depth after a single iteration. Note that there are eight similar sections in this model (beginning with $if(x > \ldots)$), each handling a neighboring cell. The first four drain 15% of the water to or from the cells above, below, and to either side; the last four drain 10% of the water to or from the cells at the diagonal positions.

This model has been applied to a section of the Spearfish data base with realistic looking results. Water drains away from the uplands forming temporary streams and ponds. For those wishing to repeat the experiment with GRASS, enter the model code in a file called *water.mapcalc*, then create a controlling shell script:

```
#!/bin/sh
r.mapcalc water=120 # start with 12 inches of water in each cell
d.rast water        # display the water map
```

```
i=1
while [ $i != 100 ]
do
        n=1
        while [ $n != 10 ]
        do
                r.mapcalc < water.mapcalc # run the simulation
                d.rast water              # display the water map
                n=‘expr $n + 1‘
        done
        g.copy rast=water,water.$i        # save a snapshot
        i=‘expr $i + 1‘
done
```

This script will run 1000 iterations. Note that the depth starts at 120 (12 inches of water) across the entire map. After every tenth iteration, the result map *water* is copied to a map with a unique name[4] for later rapid display. Run the shell scriptand watch the water "flow" off the slopes of South Dakota's Black Hills.

This model could be improved by considering additional factors, such as (1) *evaporation and transpiration*—in each time-step (iteration), some water will evaporate, which could be modeled as either a constant value or as a simple function of ground cover type; (2) *flow impedance*—the amount of water that can leave a cell in a given time step could be modified using ground cover information; and (3) *ground saturation*—during a rain event, the rate at which soils absorb water can be estimated based on soil type and length of exposure to water.

4 Imagery Examples

The algebra also supports many image processing operations. A number of these will be described here. Again, the examples are not exhaustive. They are intended to illuminate the power of the algebra.

4.1 Spectral Ratios

Ratio transformation of spectral data is one technique used in the analysis of remotely sensed data (Lillesand and Kiefer 1987). Between-band ratios, which eliminate multiplicative effects, are easily created with *r.mapcalc*:

---

[4] These maps will be named *water.1*, *water.2*,...,*water.100*.

$$ratio = band.1/band.2$$

Ratios of between-band differences, which eliminate additive effects, are also straightforward:

$$ratio = (band.1 - band.2)/(band.3 - band.2)$$

A normalized vegetation index for AVHRR data is computed as:

$$nvi = (avhrr.2 - avhrr.1)/(avhrr.2 + avhrr.1)$$

These results could be produced with a very simple algebra that uses only arithmetic operators. The *transformed vegetative index* for the Landsat Thematic Mapper illustrates the use of functions in *r.mapcalc*:

$$tvi = 100 * sqrt((tm.4 - tm.3)/(tm.4 + tm.3) + 0.5)$$

Note that 0.5 is added to help ensure that sqrt() isn't applied to a negative value. This can be guaranteed by using the max() function:

$$tvi = 100 * sqrt(max(0,\ (tm.4 - tm.3)/(tm.4 + tm.3) + 0.5))$$

If it is desired that the results be in the range 0-255, this ratio will produce such results:

$$ratio = 255.0/90.0 * atan(float(tm.1)/float(tm.2))$$

Note that the float() function is used to ensure that the division is floating point division, not integer division. Also note that because the output map can hold only integer values, *r.mapcalc* will round the results to integers before they are stored in the map.

4.2 Spatial Filters

A common image processing operation is local neighborhood filtering. A small window is moved over the image, and the center pixel is replaced by a combination of all the pixels in the window.

Low-frequency filters (also called low-pass filters) attempt to de-emphasize high spatial frequencies. These involve performing a weighted average of all the pixels in the window. A simple average, which smoothes the image, is based on the filter:

$$\frac{\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}}{9}$$

This is expressed with *r.mapcalc* as:

$$\begin{array}{lcll} ave3x3 & = & (& image[-1,-1] + image[-1,0] + image[-1,1] & \backslash \\ & & + & image[0,-1] + image[0,0] + image[0,1] & \backslash \\ & & + & image[1,-1] + image[1,0] + image[1,1] & \backslash \\ & &) & /\ 9 & \end{array}$$

High-frequency filters (high-pass filters) attempt to emphasize high spatial frequencies and can be used for edge enhancement. The technique is the same as for low-frequency filters, except that some of the weights are negative. One such filter is:

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

This is expressed with *r.mapcalc* as:

$$\begin{array}{lcll} high3x3 & = & (& -image[-1,-1] - image[-1,0] - image[-1,1] & \backslash \\ & & & -image[0,-1] + 9 * image[0,0] - image[0,1] & \backslash \\ & & & -image[1,-1] - image[1,0] - image[1,1] & \backslash \\ & &) & & \end{array}$$

The Sobel filter is a non-linear edge enhancement filter:

$$\sqrt{\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}^2 + \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}^2}$$

It is similar to the method used to determine slope from elevation and is expressed in *r.mapcalc* as:

$$\begin{array}{llll} sobel3x3 = eval(& x = & image[-1,1] + 2 * image[0,1] + image[1,1] & \backslash \\ & & -image[-1,-1] - 2 * image[0,-1] - image[1,-1]\ , & \backslash \\ & y = & image[-1,-1] + 2 * image[-1,0] + image[-1,1] & \backslash \\ & & -image[1,-1] - 2 * image[1,0] - image[1,1]\ , & \backslash \\ & sqrt(x * x + y * y) & & \backslash \\ ) & & & \end{array}$$

Another type of filter is an adaptive filter, which replaces the center pixel only if some criteria is met. Eliason and McEwen (1990) describe two adaptive filters. The center pixel is replaced by the average value in the neighborhood if the difference between the pixel value and the average value exceeds a threshold. One filter sets the threshold to a multiple of the statistical variance in the neighborhood. The other sets the threshold to a multiple of the statistical variance in the entire image. Both filters can be coded with *r.mapcalc*; the variance within the entire image is a regional calculation and must first be computed by another command[5] and then inserted into the *r.mapcalc* code. The first filter is illustrated.

Let P be the value of the center pixel, s the sum of all values in the neighborhood, n the number of pixels in the neighborhood, and ss the sum of the squares of the values in the neighborhood. The average is given by:

$$\mu = \frac{s}{n}$$

and the variance is given by:

$$\sigma^2 = \frac{ss}{n} - \mu^2$$

The center pixel is replaced if

$$(P - \mu)^2 > C\sigma^2$$

where C is usually between 1.0 and 4.0. The *r.mapcalc* code for a 3x3 window with C set to 2.25 is:

$$\begin{array}{llcll}
filter = eval(& s & = & image[-1,-1] + image[-1,0] + image[-1,1]+ & \backslash \\
 & & & image[0,-1] + image[0,0] + image[0,1]+ & \backslash \\
 & & & image[1,-1] + image[1,0] + image[1,1]\ , & \backslash \\
 & ss & = & image[-1,-1] * image[-1,-1]+ & \backslash \\
 & & & image[-1,0] * image[-1,0]+ & \backslash \\
 & & & image[-1,1] * image[-1,1]+ & \backslash \\
 & & & image[0,-1] * image[0,-1]+ & \backslash \\
 & & & image[0,0] * image[0,0]+ & \backslash \\
 & & & image[0,1] * image[0,1]+ & \backslash \\
 & & & image[1,-1] * image[1,-1]+ & \backslash \\
 & & & image[1,0] * image[1,0]+ & \backslash \\
 & & & image[1,1] * image[1,1]\ , & \backslash \\
 & ave & = & s/9.0\ , & \backslash \\
 & var & = & ss/9.0 - ave * ave\ , & \backslash \\
 & x & = & image - ave\ , & \backslash \\
 & if(x * x > 2.25 * var, ave, image) & & & \backslash \\
\quad) & & & &
\end{array}$$

---

[5] The GRASS *r.stats* command could be used to compute the variance across the entire image.

4.3 Radiometric Calibration

Radiometric calibration is the conversion of digital values recorded by a remote sensing system to radiance values at the top of the atmosphere, possibly followed by conversion to surface reflectance values. Radiance values can be computed using the following formula (Chavez 1989):

$$rad_i(x,y) = (dn_i(x,y) - offset_i)/gain_i$$

where *rad* is the radiance for band *i* at pixel *x,y*; *dn* is the value recorded by the sensors; and *offset* and *gain* are the sensor offset and gain for band *i*. This formula is handled easily by *r.mapcalc*. For example, using the values for the offset and gain as reported by Chavez for October 3, 1988 Thematic Mapper data, radiance images are constructed as follows:

$$\begin{aligned} rad.1 &= (tm.1 - 2.4899)/16.5993 \\ rad.2 &= (tm.2 - 2.3871)/8.5104 \\ rad.3 &= (tm.3 - 1.4815)/12.4074 \\ rad.4 &= (tm.4 - 1.8418)/12.2790 \\ rad.5 &= (tm.5 - 3.4240)/92.5292 \\ rad.7 &= (tm.7 - 2.6323)/175.4878 \end{aligned}$$

The radiance formula for the Spot sensor appears in a slightly modified form (Spot, 1989):

$$rad_i(x,y) = dn_i(x,y)/A_i$$

where A_i are the absolute calibration coefficients. The May 27, 1989 Spot multispectral image in the Spearfish data base can be converted to radiance, using *r.mapcalc*, as follows:

$$\begin{aligned} rad.1 &= spot.ms.1/1.05586 \\ rad.2 &= spot.ms.2/1.12140 \\ rad.3 &= spot.ms.3/0.97244 \end{aligned}$$

Surface reflectance calculations are more involved because the formulas must incorporate atmospheric effects. There are various methods for modeling these effects (Richards 1986, Price 1987, Chavez 1989). A formula given by Chavez is:

$$R_i(x,y) = \frac{\pi dist^2[rad_i(x,y) - haze_i]}{E_i slope(x,y) \cdot sun \cdot mhaze_i}$$

where R is the surface reflectance image; *rad* is the radiance image; *slope* is a slope map; *dist* is the Earth-Sun distance; *haze* and *mhaze* are the additive and multiplicative atmospheric haze factors; E is exoatmospheric spectral irradiance; and *sun* is the sun elevation at the time the image was captured. Combining the non-map factors into single constants, the formula can be written in the form:

$$R_i(x,y) = \frac{A_i[rad_i(x,y) - B_i]}{slope(x,y)}$$

which, after substituting the appropriate values for the constants A and B, is expressed with *r.mapcalc* as:

$$\begin{array}{rcl} R.1 & = & A_1 * (rad.1 - B_1)/slope \\ R.2 & = & A_2 * (rad.2 - B_2)/slope \\ R.3 & = & A_3 * (rad.3 - B_3)/slope \\ & \cdot & \\ & \cdot & \\ & \cdot & \end{array}$$

4.4 Principal Components

Principal components analysis involves transforming a set of correlated variables into a new, uncorrelated set. In this case the variables are the band files from a multispectral sensor. The transformation is a linear combination of the band data:

$$component_i = \sum_{j=1}^{N} w_{ij} * band_j \quad i = 1, \ldots, N$$

where N is the number of band files and w_{ij} are the eigenvectors for the between-band covariance matrix. While neither the covariance computation nor the determination of the eigenvectors can be accomplished using the map algebra, the components can be computed. For example, suppose the three multispectral bands from a Spot image had the following covariance matrix:

$$\begin{bmatrix} 462.88 & 480.41 & 281.76 \\ 480.41 & 513.02 & 278.92 \\ 281.76 & 278.91 & 336.33 \end{bmatrix}$$

A set of eigenvectors for this matrix (in decreasing order of spectral variance) is:

$$\begin{array}{lrrr} \text{vector}_1 & 21.24 & 22.15 & 14.77 \\ \text{vector}_2 & 2.91 & 4.46 & -10.87 \\ \text{vector}_3 & 1.82 & -1.62 & -0.18 \end{array}$$

To compute the second component with *r.mapcalc*:

$$pc.2 = 2.91 * spot.ms.1 + 4.46 * spot.ms.2 - 10.87 * spot.ms.3$$

4.5 Merging Panchromatic Data with Multispectral Data

The French SPOT satellite produces a 10-meter panchromatic image plus three spectral bands with 20-meter resolution. Landsat Thematic Mapper produces six spectral bands with 30-meter resolution (as well as a 120-meter resolution thermal band). A sharpened color image can be produced by merging the high resolution panchromatic image with the lower resolution spectral bands. Assuming that the spectral data have been registered and resampled to the 10-meter panchromatic data, a number of techniques have been used to perform the merger, most of which can be done with the map algebra.

Welch (1987) describes two direct methods:

$$\begin{aligned} M_i' &= a_i(w_1 M_i + w_2 P) + b_i \\ M_i' &= a_i(M_i P)^{\frac{1}{2}} + b_i \end{aligned}$$

where M_i is spectral band i; P is the panchromatic band; w_1 and w_2 are weights; and a_i and b_i are chosen to make the results fall within the range 0-255. Both of these methods can be implemented using the algebra.

A simple average, based on the first method, is coded as:

$$\begin{aligned} merge.1 &= (spot.ms.1 + spot.pan)/2 \\ merge.2 &= (spot.ms.2 + spot.pan)/2 \\ merge.3 &= (spot.ms.3 + spot.pan)/2 \end{aligned}$$

Or a combination of both methods can be used:

$$\begin{aligned} merge.1 &= sqrt(spot.pan * spot.ms.1) \\ merge.2 &= sqrt(spot.pan * spot.ms.2) \\ merge.3 &= (spot.pan + 3 * spot.ms.3)/4 \end{aligned}$$

A third method is to transform three selected bands from red, green, and blue color space to intensity, hue, and saturation color space. The intensity is replaced by the panchromatic image, and the new intensity, hue, saturation combination is transformed back into red, green, and blue.

4.6 Intensity, Hue, Saturation

Conversion from red, green, and blue colors to intensity, hue, and saturation involve formulas that can be represented using the algebra. The following is based on the hue, saturation, and value (HSV) algorithm found in Foley and Van Dam (1984). Assuming that R, G, and B represent the red, green, and blue components of an image, the value (V), saturation (S), and hue (H) can be formed as follows:

$$
\begin{array}{rcll}
V &=& max(R,G,B) & \\
S &=& 255.0*(V-min(R,G,B))/V & \\
H &=& if(S, eval(& \backslash \\
 & & \quad m = float(min(R,G,B)) \ , & \backslash \\
 & & \quad r = (V-R)/(V-m) \ , & \backslash \\
 & & \quad g = (V-G)/(V-m) \ , & \backslash \\
 & & \quad b = (V-B)/(V-m) \ , & \backslash \\
 & & \quad h = if(R == V, b-g)+ & \backslash \\
 & & \qquad\quad if(G == V, 2+r-b)+ & \backslash \\
 & & \qquad\quad if(B == V, 4+g-r) \ , & \backslash \\
 & & if(h <= 0, h+6, h)*60)) &
\end{array}
$$

While saturation is normally defined in the range 0-1, it is multiplied here by 255 to retain more information after conversion to integer. Hue will be in the range 0-360, with 0 representing undefined hue (i.e., white, black, or grey).

The inverse transformation is:

$$
\begin{array}{rcll}
R &=& eval(& \backslash \\
 & & \quad s = S/255.0 \ , & \backslash \\
 & & \quad h = if(H >= 360, H-360, H)/60.0 \ , & \backslash \\
 & & \quad i = int(h) \ , & \backslash \\
 & & \quad f = h-i \ , & \backslash \\
 & & \quad p = V*(1-s) \ , & \backslash \\
 & & \quad q = V*(1-s*f) \ , & \backslash \\
 & & \quad t = V*(1-s*(1-f)) \ , & \backslash \\
 & & if(i == 0, V) + if(i == 1, q) + if(i == 2, p)+ & \backslash \\
 & & if(i == 3, p) + if(i == 4, t) + if(i == 5, V)) &
\end{array}
$$

$$
\begin{array}{rcll}
G &=& eval(& \backslash \\
 & & \quad s = S/255.0 \ , & \backslash \\
 & & \quad h = if(H >= 360, H-360, H)/60.0 \ , & \backslash \\
 & & \quad i = int(h) \ , & \backslash \\
 & & \quad f = h-i \ , & \backslash \\
 & & \quad p = V*(1-s) \ , & \backslash \\
 & & \quad q = V*(1-s*f) \ , & \backslash \\
 & & \quad t = V*(1-s*(1-f)) \ , & \backslash \\
 & & if(i == 0, t) + if(i == 1, V) + if(i == 2, V)+ & \backslash \\
 & & if(i == 3, q) + if(i == 4, p) + if(i == 5, p)) &
\end{array}
$$

$$
\begin{array}{rcll}
B & = & eval(& \backslash \\
 & & \quad \begin{array}{rcl} s & = & S/255.0\ , \end{array} & \backslash \\
 & & \quad \begin{array}{rcl} h & = & if(H >= 360, H-360, H)/60.0\ , \end{array} & \backslash \\
 & & \quad \begin{array}{rcl} i & = & int(h)\ , \end{array} & \backslash \\
 & & \quad \begin{array}{rcl} f & = & h-i\ , \end{array} & \backslash \\
 & & \quad \begin{array}{rcl} p & = & V*(1-s)\ , \end{array} & \backslash \\
 & & \quad \begin{array}{rcl} q & = & V*(1-s*f)\ , \end{array} & \backslash \\
 & & \quad \begin{array}{rcl} t & = & V*(1-s*(1-f))\ , \end{array} & \backslash \\
 & & \quad if(i == 0, p) + if(i == 1, p) + if(i == 2, t) + & \backslash \\
 & & \quad if(i == 3, V) + if(i == 4, V) + if(i == 5, q)) &
\end{array}
$$

5 Conclusion

The approach of providing a map algebra for manipulation of raster data results in a very flexible system. Given the appropriate raster data layers, the type and number of potential data manipulations are virtually limitless. For many applications, users are freed from software limits and bounded only by their ability to employ this algebra. The potential of *r.mapcalc* in fact supports three levels of usage: (1) it is a resource for users who need to perform specific algebraic functions that are not provided by other GRASS programs; (2) it is a foundational tool for advanced GIS users to develop a limitless set of image and map analysis functions; and (3) it is an important resource for programmers and developers to use in building new functions.

r.mapcalc is perhaps most important at the intermediate level, as a tool for advanced GIS users. A common implementation model for GIS in larger organizations is one or two sophisticated users supporting numerous casual users. In this context, *r.mapcalc* becomes a language that opens an array of possiblities for spatial analysis and provides a means to develop macros to support routine operations by less sophisticated users.

References

Burrough, P. A., 1986, *Principles of Geographical Information Systems for Land Resources Assessment*, Monograph on Soils and Resources Survey No. 12, (New York: Oxford University Press), 194 p.

Chavez, Pat S., Jr., 1989, Radiometric Calibration of Landsat Thematic Mapper Multispectral Images, *Photogrammetric Engineering and Remote Sensing*, Vol. 55, **9**, p. 1285-1294.

Dozier, J., and Strahler, A. H., 1983, Ground Investigations in Support of Remote Sensing, In *Manual of Remote Sensing*, Vol. 1, edited by R. N. Colwell (Falls Church, Virginia: American Society of Photogrammetry), 1232 p.

Eliason, Eric M., and McEwen, Alfred S., 1990, Adaptive Box Filters for Removal of Random Noise from Digital Images, *Photogrammetric Engineering and Remote Sensing*, Vol. 56, **4**, p. 453-458.

Foley, James D., and Van Dam, Andries, 1984, *Fundamentals of Interactive Computer Graphics*, (Reading, Massachusetts: Addison-Wesley), 664 p.

Frank, Thomas D., 1988, Mapping Dominant Vegetation Communities in the Colorado Rocky Mountain Front Range with Landsat Thematic Mapper and Digital Terrain Data, *Photogrammetric Engineering and Remote Sensing*, Vol. 54, **12**, p. 1727-1734.

Horn, B. K. P., 1981, Hill Shading and the Reflectance Map, *Proceedings of the I.E.E.E.*, **69**, 14.

Lillesand, Thomas M., and Kiefer, Ralph W., 1987, *Remote Sensing and Image Interpretation*, 2d ed. (New York: John Wiley & Sons), 721 p.

Price, John C., 1987, Calibration of Satellite Radiometers and the Comparison of Vegetation Indices, *Remote Sensing of Environment*, **21**, p. 15-27.

Richards, John A., 1986, *Remote Sensing Digital Image Analysis*, (New York: Springer-Verlag), 281 p.

Skidmore, Andrew K., 1989, A Comparison of Techniques for Calculating Gradient and Aspect from a Gridded Elevation Model, *International Journal of Geographical Information Systems*, Vol. 3, **4**, p. 323-334.

Spot User's Handbook, 1989, (Reston Virginia: USA Spot Image Corporation), Vol. 2, page 2-49.

Tomlin, C. Dana, 1990, *Geographic Information Systems and Cartographic Modeling*, (Englewood Cliffs, New Jersey: Prentice-Hall), 249 p.

Welch, R., and Ehlers, M., 1987, Merging Multiresolution SPOT HRV and Landsat TM Data, *Photogrammetric Engineering and Remote Sensing*, Vol. 53, **3**, p. 301-305.