A GRASS 3D-Vector Module for finding orthogonal Projections of Points on complex Surfaces

Alfonso VITTI, Paolo ZATELLI, Fabio Zottele

Dipartimento di Ingegneria Civile e Ambientale,
Università degli Studi di Trento,
Via Mesiano 77, 38100 Trento, Italy.
tel. +39-0461-88-2618, fax +39-0461-88-2672,
e-mail Paolo.Zatelli@ing.unitn.it

Abstract

A GRASS module for the evaluation of the distance subtended by a point in the space and its normal projection on the terrain surface has been developed. The knowledge of this distance allows to implement a physical model for the description of thermally driven slope winds. The model has been derived from an extension of a work by Prandtl, [1] - [2]. This study take advantage of the new GRASS 3D vector cababilites following a previous work based on a 3D raster approach, [3]. The new GRASS 3D vector implementation allows users to manage irregularly placed sets of points overcoming the rigid definition of a voxel (volume pixel) structured grid. This paper reports, from the programmer's point of view, a detailed description of the use and the connections between geometric primitives and their attribute data as implemented in the GRASS 3D vector architecture. The information required to solve the Prandtl's equation for the wind speed are stored as attributes of a 3D vector map and managed using the PostgreSQL DBMS and SQL queries. A detailed description of the GRASS module source code is also reported. A complete description of the Prandtl physiclal model and of its implementation within GRASS can be found in [3].

1. Introduction

A new GRASS module has been written to evaluate the normal distance between a point located in the 3D space and a complex surface. This module meets the demanding of the atmospheric models, such Prandtl's one, but it has been developed for all the applications in which a normal projection of points is needed. The new GRASS vector architecture, [4] - [5],although higly experimental during the module's developement, proved to fulfill the requirements for this model implementation: the need of managing points no more constrained to the a voxel grid data. This is useful when thicking the analysis in particular areas of the computational domain or, more simply, when a set of experimental measures are known in pre-determined points (i.e. glide's trajectories determined by GPS). The output of the *v.perp.seek*, mainly the three coordinates of the point to be projected, the coordinates of the point on the surface along normal projection and the normal distance, can be used as a source for further data analisys. An example is reported to show how to manage the information collected in the external vector database for expanding the attribute's map.

This paper mainly focus on the tecnical aspects of the model implemetatios, such the definition of a procedure for the evaluation of the normal distance, the code writing, the study, the analisys and the use of some of the GRASS 3D vector capabilites, the definition of the SQL statements and queries used to manage vector attributes and to evaluate the physical quantities. The description of the Prandtl physical model and how it has been implemented using GRASS has here been omitted; a detatiled description can be found in [3], while a more detailed introduction on the theory and on the application filed of the slope winds can be found in [6] - [7] and [8].

2. Evaluation of the normal to a surface

Given the three coordinates of a point in the space, its normal projection on the terrain surface and the measure of the normal segment have to be evaluated. The point to be projected is one of the two vertices of the normal segment.

Usually, the terrain surface is expressed by a raster DEM, slope and aspect maps can be easily evaluated from the DEM: the horizontal cell of the terrain model can be tilted using these two values, restoring for the courrent elemental the surface's original orientation.

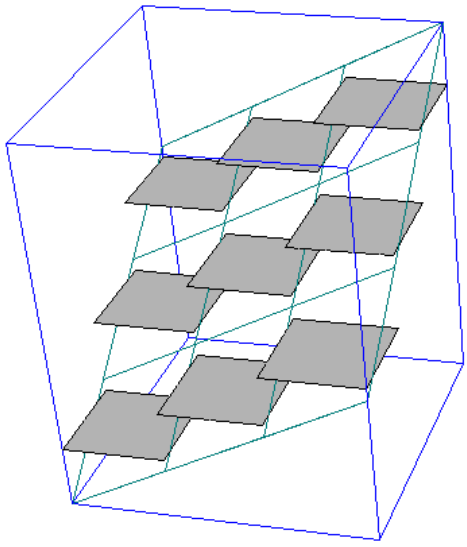

Figure 1: Conceptual scheme of a DEM (gray) and the plane to be intepolated (green)

The gradient components of the DEM can be used for the determination of the normal direction. Anyway, the wide availability of slope and aspect information makes these parameters preferable. Given a point P in the space the terrain surface orientation parameters can be used to find its actual projection point on the terrain surface. This procedure can be semplified considering, as an aproximate value of the projection point, the DEM cell that minimize the distance from the point P.

2.1 System geometry

A point in the 3D space can be espressed as:

$$P = (x, y, z)_P = (x_P, y_P, z_P) \tag{1}$$

and a general scalar field may be written as:

$$F(x, y, z) = Ax + By + Cz + D \tag{2}$$

A plane on the 3D space can be written as:

$$F(x, y, z) = 0 \tag{3}$$

Moreover it has to be verified that:

$$P \notin F(x, y, z)$$

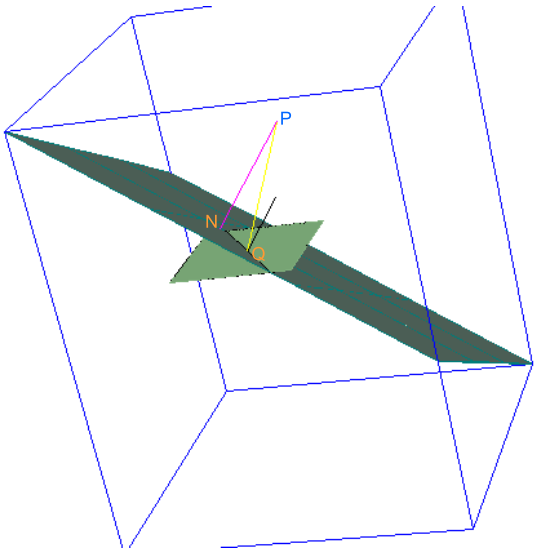

Figure 2: System geometry

From equation (2) it is possible to evaluate the components of the gradient vector wich may be applied to every point Q satisfying (3).

$$\nabla F(x, y, z) = \left(\frac{\partial F(x, y, z)}{\partial x}, \frac{\partial F(x, y, z)}{\partial y}, \frac{\partial F(x, y, z)}{\partial z} \right)$$

$$\nabla F(x, y, z) = (A, B, C)$$

The plane interpolating a portion of DEM is generally unknown, but, as further explained, it is possible to indirectly compute its gradient's components. As a first step, the term D of (2) is drawn out:

$$D = -Ax - By - Cz \tag{4}$$

Considering the point $Q = (x, y, z)_Q = (x_Q, y_Q, z_Q)$, as the center of the DEM's cell, it is possible to write:

$$Q \in \ F(x, y, z) = 0$$

Using the equation (4), the value of D is expressed as:

$$D = -Q \cdot \nabla F(x, y, z) = -Ax_Q - By_Q - Cz_Q \tag{5}$$

The distance between P and the plane is:

$$d = \frac{|Ax_P + By_P + Cz_P + D|}{\sqrt{A^2 + B^2 + C^2}} \tag{6}$$

Using (5) with (6) it is possible to write an expression for the distance depending only on the unknown vector $\nabla F(x, y, z)$.

$$d = \frac{|P \cdot \nabla + D|}{\sqrt{A^2 + B^2 + C^2}} = \frac{|P \cdot \nabla - Q \cdot \nabla|}{\|\nabla\|} \tag{7}$$

The previous expression, rewritten using the versor notation, leads to:

$$d = |(P - Q) \cdot \vec{\mathbf{v}_G}|$$

which depends on the direction identified by P and Q as:

$$\vec{\mathbf{v}_R} = \frac{P - Q}{|P - Q|}$$

The distance is then expressed as:

$$d = |\vec{\mathbf{v}_R} \cdot \vec{\mathbf{v}_G}| \|P - Q\| \tag{8}$$

Knowing the value of d, is simple to trace back the coordinates of the normal projection of the point P

$$N = P - \vec{\mathbf{v}_G} d \tag{9}$$

2.2 Gradient components

Althoug two of the three gradient components can be extract directly from the DTM, (using the GRASS module *r.slope.aspect*) slope and aspect maps are more often used to describe the orientation of surfaces. A numeric formulation to evaluate the gradient by the former knowledge of these two parameters has to be defined.

Knowing the two planimetric components of the gradient vector, it's quite easy to compute slope α and aspect θ angles using the following mathematical expression:

$$\alpha = tan^{-1}\sqrt{\nabla_x^2 + \nabla_y^2}\frac{180}{\pi} \tag{10}$$

$$\vartheta = atan2(\nabla_y, \nabla_x)\frac{180}{\pi} \tag{11}$$

In the equation (11) the matemathic function $atan2(x, y)$ is used. This function computes the principal value of the arc-tangent of x/y, using the signs of both arguments to determine the quadrant of the return value, [9]. An inverted form of $atan2(x, y)$ is necessray to estimate gradient's components. The reported definition is not useful: a much precise formulation is written below.

$$atan2(y, x) = \begin{cases} tan^{-1}\left(\dfrac{y}{x}\right) & x > 0 \\ sign(y)\left[\pi - tan^{-1}|\dfrac{y}{x}|\right] & x < 0 \\ 0 & x = y = 0 \\ sign(y) \cdot \dfrac{\pi}{2} & x = 0,\ y \end{cases} \tag{12}$$

where:

$$sign(x) = \begin{cases} 1 & x \geq 0 \\ -1 & < 0 \end{cases} \tag{13}$$

The 11 formula is only-locally invertible. The aspect value indicates the direction that slopes are facing. In this study the values of ∇_x and ∇_y are resuming accordingly to the magnitude of ϑ.

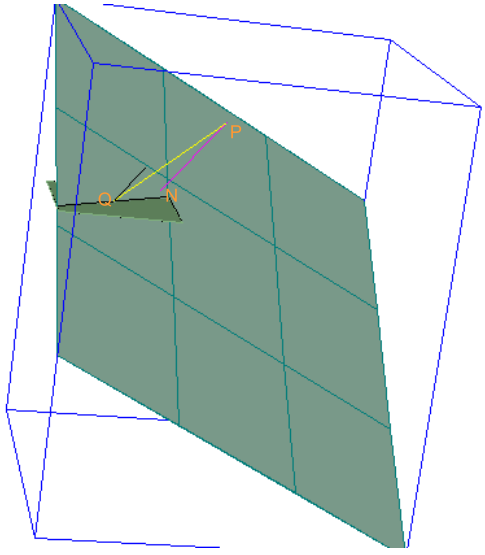

Figure 3: System geometry

For the third component of the gradient vector, ∇_z, a further analysis has to be performed. Matematically speaking, the equation (3) represents an implicit representation of the plane in the point of interest. A simpler way to describe a plane is:

$$z = ax + by + c \tag{14}$$

The parameters in equation (14) are computed using three non-allineated points belonging to the plane. Equation (3) needs one more point to be solved.
A new scalar field can be defined as:

$$f(x, y) = ax + by + c$$

The value of $\partial F/\partial z$ has to be obtained without an explicit knowledge of the (14) and of the coefficients of equation (3). A suitable use of the compound function's derivation rule, [9], repays the efforts of inferring $\partial f/\partial x$ and $\partial f/\partial y$ without explicitly knowing $f(x, y)$. It's here assumed that (14) defines:

$$F[x, y, f(x, y)] = 0 \tag{15}$$

then $F(x, y, z) = 0$ defines *implicitly* z as function of x and y, although it is not possible to give an explicit formula for $f(x, y)$. It can therefore be written as:

$$z = f(x, y)$$

A further auxiliary function can be defined as:

$$g(x, y) = F[x, y, f(x, y)].$$

Equation (15) is equivalent to $g(x, y) = 0$ and therefore $\partial g/\partial x = \partial g/\partial y = 0$. These partial derivative values can be determined using the compound functions derivative rule:

$$g(x, y) = F[u_1(x, y), u_2(x, y), u_3(x, y)]$$

where: $u_1(x, y) = x$, $u_2(x, y) = y$ e $u_3(x, y) = f(x, y)$. For semplicity, an alternative notation is used in the following. The **y** quantity is an unitary vector and the $f'(\mathbf{a}; \mathbf{y})$ is the directional derivative function of f in **a** referred to the **y** direction. In particular, if $\mathbf{y} = \mathbf{e}_k$ (unitary vector with the k-th component equal to 1) the directional derivative $f'(\mathbf{a}; \mathbf{e}_k)$ is called partial derivative function referred to $\mathbf{e}_k$ and is written as $D_k f(\mathbf{a})$:

$$D_k f(\mathbf{a}) = f'(\mathbf{a}; \mathbf{e}_k)$$

Then, the compound function's derivative rule gives:

$$\frac{\partial g}{\partial x} = D_1 F \frac{\partial u_1}{\partial x} + D_2 F \frac{\partial u_2}{\partial x} + D_3 F \frac{\partial u_3}{\partial x}$$

and

$$\frac{\partial g}{\partial y} = D_1 F \frac{\partial u_1}{\partial y} + D_2 F \frac{\partial u_2}{\partial y} + D_3 F \frac{\partial u_3}{\partial y}$$

where the partial derivative $\frac{\partial F}{\partial x}$ and $\frac{\partial F}{\partial x}$ have to be computed in the point $(x, y, f(x, y))$. Since

$$\frac{\partial u_1}{\partial x} = 1, \quad \frac{\partial u_2}{\partial x} = 0, \quad \frac{\partial u_3}{\partial x} = \frac{\partial f}{\partial x} \quad e \quad \frac{\partial g}{\partial x} = 0,$$

$\partial g/\partial x$ can be written as:

$$\frac{\partial F}{\partial x} = D_1 F + D_3 F \frac{\partial f}{\partial x} = 0.$$

which gives:

$$\frac{\partial f}{\partial x} = -\frac{D_1 F[x, y, f(x, y)]}{D_3 F[x, y, f(x, y)]}$$

referred to the points where $D_3 F[x, y, f(x, y)] \neq 0$. Analogously, the corresponding formulation for $\partial f/\partial y$ is:

$$\frac{\partial f}{\partial y} = -\frac{D_2 F[x, y, f(x, y)]}{D_3 F[x, y, f(x, y)]}$$

where $D_3 F[x, y, f(x, y)] \neq 0$. Briefly, this formulas are usually written as:

$$\frac{\partial f}{\partial x} = -\frac{\partial F/\partial x}{\partial F/\partial z}, \quad \frac{\partial f}{\partial y} = -\frac{\partial F/\partial y}{\partial F/\partial z} \tag{16}$$

Referring at the same time to (16), (2.2) and (15) it can be stated:

$$a = -\frac{A}{C}$$

$$b = -\frac{B}{C}$$

The two mentioned scalars fields have to represent the same interpolating surface, when the local ineversions of (11) and (10) are used with the purpose of evaluate ∇_x and ∇_y, the following expression is obtained:

$$\nabla_z = 1 \tag{17}$$

3. Module's implementation

The drop of the mathematic procedure, described above, into a new GRASS module based on the new 3D vector data architecture, [4] - [5], of GRASS is now discussed.

All the vector capabilities are recalled by libraries and code dependencies: the module, during compilation, is address by a *Makefile* to the locations where drawing the data structures off.
`GISLIB` the standard GRASS library, `VECTLIB` the vector data management library, and `DBMILIB` for the attribute management give the programmer a set of optimized functions to use in the developement phase. As suggested by *GRASS Developement Team*, a HTML manual comes with the module.

3.1 Targets

The *v.perp.ssek* module has been developed for a wider range project. The Prandtl theory is refered to a particular orthogonal frame with the s axis parallel to the slope and n axis normally defined. The geographic data refer to a Cartesian frame with the x direction along the horizontal plane and the z in the vertical direction.

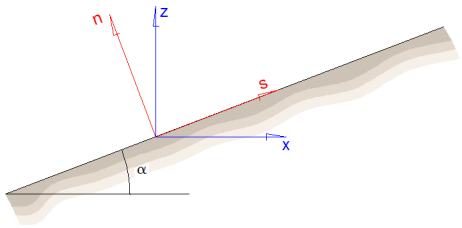

Figure 4: Reference systems on the tilted plan modeling the sloping ground.

The n coordinate is one of the parameters used by Prandtl for determining atmospheric conditions. The 3D module has to dart a normal segment from P searching the intersection with the terrain surface. Knowing the spacial position of the point to project and its normal projection, the distance n is evaluable as:

$$n = \sqrt{(x_P - x_N)^2 + (y_P - y_N)^2 + (z_P - z_N)^2}$$

However, the problem is much more complicated: the mathematic representation, through scalars field or equivalent forms, of the surfaces describing the terrain is unknown. As discussed the DEM discretization, accomplished by slope and aspect information, may suffice for an approximated value of normal segment's length.

The first aim of *v.perp.seek* is to generate a 3D vector map of lines connecting the points to be projected (read from the input 3D vector map) to the normal projection on DEM. In the linked table, as attributes, for each point various information are stored: the coordinates of the point to be projected, the distance between the two points, the coordinates of the normal projection, the indexes of the cell where the projection falls, the two planimetric coordinates of the cell in which the projection falls, the directions of the gradient versor and the components of the maximum slope direction.

Generally, terrain morfology is stored in DEMs: the surface is discontinuos, a mosaic of horizontal planes. As seen in §2.2, the cells of DEM can be tilted and oriented using the *slope* and *aspect* parameters. Once the gradient vector is known a richer description of land is available but the resulting surface is however discontinuos. For a complex surface more than one normal projections can be found for the same point. Regarding to the atmospheric model, wind and temperature conditions are influenced mostly by the physical conditions of the nearest projection, so only the smallest normal distance is considered. Where slope values rapidly change, the evaluation of the normal distance may be complex. The vector approach allows the management of geometrical configurations better than

the raster-voxel one (i.e. thickening the computing grid only near these zones). The definition of an algorithm to analitically manage geometrical singularities is still under development.

Although the module was intended as part of a wider tridimensional atmospheric model, an high grade of generality has been mantained, the module evaluates the normal projections of points either below or above the terrain surface. Such an operation would be non-sense in the atmospherical field while it could be interesting in others.

3.2 Normal distance's individuation pattern

For the points close to the surface the distance, evalutated on the surface, between the *normal* and the *vertical* projection can be assumed small, save for configuration with geometrical singularities. The determination of the vertical projection is trivial and this point, which minimize the distance from the projecting point in the space, can be used as a starting point for the search of the normal projection. The algorithm that evaluates the normal to a surface uses this proposition starting from a point close to the surface and moving towards the interesting point P in the space.

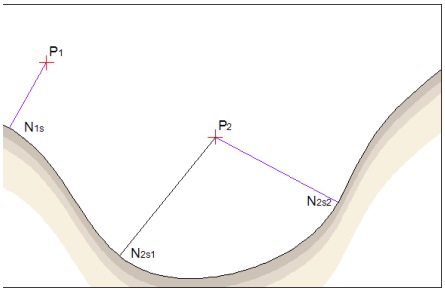

Figure 5: Normal projection of $P1$ and $P2$. For the second point only N_{2s2} has to be evaluated.

Let $P : (x_P, y_P, z_P)$ be the point to project. The first step is to compute the vertical projection $V : (x_V, y_V, z_V)$, the segment $\overline{PV}$ is divided into k parts, each with edge points $P_V(i)$ with coordinates given by:

$$P_V : (x_P + i\frac{\overline{PV}}{k}, y_P + i\frac{\overline{PV}}{k}, z_P + i\frac{\overline{PV}}{k}). \tag{18}$$

with i running from 0 to k, where obviously $P_V(i) \equiv V$for $i = 0$, $P_V(i) \equiv P$ for $i = k$ and lower values of k correspond to points closer to the surface. The algorithm starts with $k = 1$, for this point $P_V(1)$, which is close to the terrain surface, the normal and vertical direction are very close, unless the slope is very slanting or a singularity of the surface is involved. Therefore, a window of points under $P_V(1)$ is scanned and the distance between $P_V(1)$ and the centers of these cells are evaluated. The point $Q : (x_Q, y_Q, z_Q)$ corresponding to the shortest distance $d_{min} = \overline{PQ}$ can be easly found.

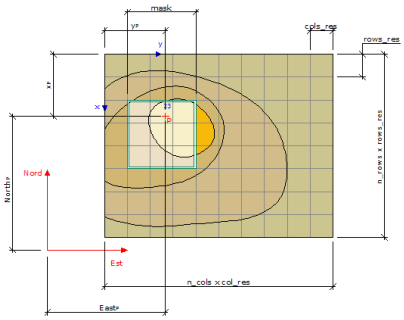

Figure 6: Scanning window over the DEM for the search of the minimum distance point.

The versor:

$$\overrightarrow{\mathbf{v}_r} = \frac{\overleftarrow{Q - P_V(1)}}{d_{min}}$$

identifies the direction of the segment from $P_V(1)$ to Q, which in general is not the normal direction since only the cell centers are tested. For the cell to which Q belongs, the normal direction to the terrain surface can be evaluated knowing the DEM gradient.

$$\overrightarrow{\mathbf{v}_g} = \frac{\overrightarrow{\nabla}}{\|\nabla\|} = \frac{(\nabla_x, \nabla_y, \nabla_z)}{\sqrt{{\nabla_x}^2 + {\nabla_y}^2 + {\nabla_z}^2}}$$

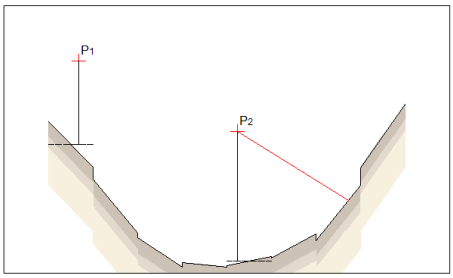

Figure 7: The terrain surface is modelled trough a DEM: every cell is tilted by the gradient vector. The resulting surface doesn't match the real one, thus influencing the normal projection evaluation.

If $\overrightarrow{\mathbf{v}_Q} \equiv \overrightarrow{\mathbf{v}_g}$ then $\overrightarrow{\mathbf{v}_Q}$ is the normal direction trough $P_V(1)$, otherwise, as usually happens, the distance from Q to the normal projection N can be evaluated through (8), N is found by moving, of the quantity d, along the direction $\overrightarrow{\mathbf{v}_g}$ as shown in (9).

Once the point N for $P_V(1)$ is found, the procedure is iterated for the other $k-1$ points. However, after the first iteration the point N of the previous iteration, rather than the vertical projection of $P_V(k)$, is used as starting point for the search.

3.3 The code

The procedure described above has been implemented as a new GRASS module *v.perp.seek*, taking advantage of the new GRASS vector architecture and the possibility to link the vector attributes to a database table, [4] - [5]. The *v.perp.seek* module creates a tridimensional vector map of the segments that connect each input point with its projection on a DEM. To the vector map a database table is associated where the following information are stored in each record: the coordinates of the point to be projected, the normal distance, the coordinates of the projection points, the index to the DEM cell containing the projection point, and the coordinates of the center of the cell. The input of the module consists in a vector map containing the points whose normal must be evaluated, the DEM, the slope and the aspect maps. The slope and aspect information are used to evaluate the local gradient of the surface. For a concave terrain surface more thane one normal direction can exist, the module select the direction corresponding to the shortest distance.

The program, written in C language, is structured in blocks. In the first part the external libraries are recalled. `Gis.h`, `Vect.h`, and `dbmi.h` are used for processing raster, vector and attribute data respectively. The POSIX library `unistd.h` is disposed for the correct initialization of the table management driver.

The data structure `Point_3D` is then initialized as a useful way for storing every tridimensional vector entity: points, directions, vectors, and versors.

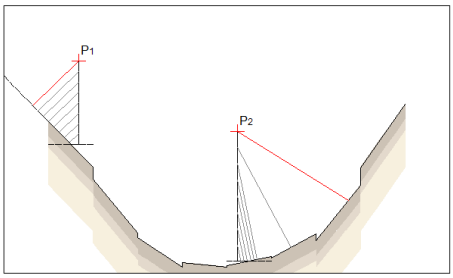

Figure 8: After evaluating the vertical projection the search algorithm is started.

The uptading of the tables linked to the vector map is done by the `write_line` function. For clarity's sake, this part of code is broken up from the main: during module's developement it has represented the hard-core of the information management. Moreover, an external procedure is much more flexible when changes in the output are needed.

The first part of the *main*, is dedicated to the initialization of the program and to input data harvesting. The last two parameters that must be chosen are *trimmering* and *masking*, the former k and m in the equation 3.2. Respectively, these two parameters represent the number of segments into which the vertical segment from the point to the surface is divided, and the dimension, as number of cells, of the square "neighbourhood tool" on the terrain. These parameters are set as a trade off between

the risk of selecting the wrong point on the surface and the computational cost. Infact, the choice of large values for k increases the number of iteration during the projection's seeking, while small values of k can lead to situations where the hypotesis that normal and vertical directions are close does not hold, resulting in the failure of the algorithm. The selection of large values of m causes the need of scanning a large number of cells when looking to the minimum distance (the number of scanned cells scales with m^2), however a small value of m can lead to the individuation of a local minimum for the cells' centers-point distances, which is not the absolute minimum, resulting again in a failure of the procedure.

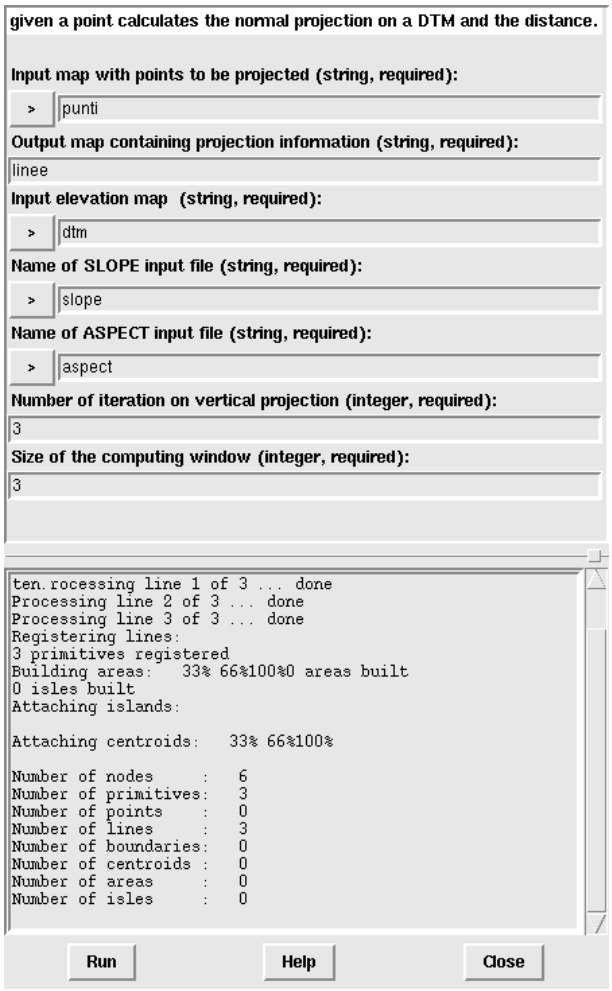

Figure 9: Module's input window.

Aftre parsing, the variable `type = GV_POINT` is set. This allows module to process every vector primitive input, reading and processing only the points which make them up.
From line 195 to 199, the vector structures needed for reading data inputs and write data outputs are

recalled: at line 219 the header of the table linked to vector entities is set:

Cat : category index (*integer*);

xP, yP, zP : point to project (*double precision*);

distance : normal distance (*double precision*);

xN, yN, zN : projection point (*double precision*);

cell : cell index where projection falls (*integer*);

x_cell, y_cell : index cell's baricenter (*double precision*);

vg_x, vg_y, vg_z : gradient versor (*double precision*);

vn_x, vn_y, vn_z : max slope versor (*double precision*).

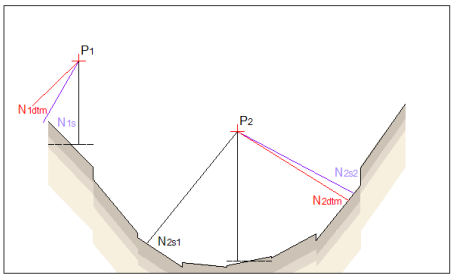

Figure 10: Theoric and effective projection' comparison.

From line 202 to 210 the vector input map is open at 2nd level, i.e. all topology is imported. The output map is then created and the header and history files are written. Linked tables's creation is preceded by creation of the *field* and of the link to the table into the field. This may seem such a complication, but it is the only way to manage more table in a field and to link the field (with all tables into) to a vector map. A series of functions do this job:

```
fi = Vect_default_field_info(&vector_map_new, 1, NULL, GV_1TABLE)
```

initialized a new database with a table into it and links it to the vector output map.
Then

```
Vect_map_add_dblink(&vector_map_new, 1, NULL,
fi->table, "cat", fi->database,fi->driver)
```

sets the database driver. The default driver is the GRASS native one, dbf, and it opens the database for writing with:

```
driver = db_start_driver_open_database(fi->driver, fi->database)
```

The vector map can now be written and the tables in the field can be populated. Every time a primitive is added the table in the field will be updated.

From line 228 to line 262 the raster data imput map are open for further reading. With
`G_get_cellhd(dtm_name, dtm_mapset,&header)`
the limit of the maps are rekoned and information about resolution of maps are harvested.

From line 228 to line 451 the core of the discussed algorithm for finding the projection is implemented: every vector point is processed until the projection's coordinates and all the related data are ready to be written to the output vector map.

The two planimetric coordinates of the point to project are enough to identify the memory address of the raster cell on wich the vertical projection falls (from line 283 to 291).

The storage of raster data can be achieved in two ways: with the transcription to memory of the whole map or copying small noticeable portion of the map with dynamic allocating memory routines and freeing the RAM space when data are no longer needed. With the second approach machine's resources are optimized. So, the reading is made pointing the right cell on the map accordingly with the `jump_x` and `jump_y` values. As far as raster informations are collected in a "blind" way a check on the typology of data is always made (from line 286 to 292 et sim.). Then the algorithm implementation starts, restrained from line 306 to 309, the `do` loop goes up or downward the vertical (accordingly to `g_wt`'s value) finding intermediate projections. The control loop continues until the while loop
`while (pov.z != ptp.z)`
is satisfied.

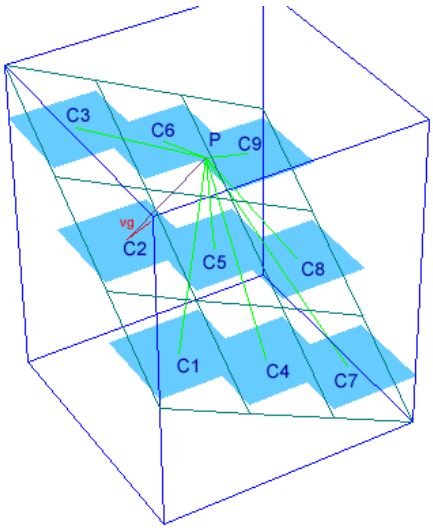

Figure 11: Distance analysis in the computing window. $(\overline{PC})$ is compared with gradient's versor.

The DEM information collected in the window-matrix are processed to detect the minimal distance

(lines 354-364); the cell is indexed and the center's coordinates evaluated, the direction $\overleftarrow{Q\ P_V}$ with the $\overrightarrow{\mathbf{v}_r}$ is computed. Starting with Q's coordinates, slope and aspect values are computed (lines 380-401) and the ∇_x e ∇_y are evaluated (lines 402-433). Consequently, knowing $\overrightarrow{\mathbf{v}_g}$, the equations (8) and (9) are used to find the normal distance and the projection's coordinates. Moreover, the maximum slope versor $\overrightarrow{\mathbf{m}}$ is evaluated as:

$$\overrightarrow{\mathbf{m}} \cdot \nabla = 0$$

$$m_x = -\nabla_x$$

$$m_y = -\nabla_y$$

$$m_z = \frac{\nabla_x^2 + \nabla_y^2}{\nabla_z}$$

When $i = k$, the projection of P is found and all the information are stored in the map and to the linked table by `write_line` procedure. The debug level is set to the third level with: `G_debug(3,"write_line ()")` so that every line in the vector file is checked for errors. The vector map is then prepared for being written with `Vect_reset_line()` and `Vect_reset_cats()`. The `Vect_append_point()` stores the coordinate accomplaining the GRASS 3D vector data structure and finally with `Vect_write_line()` a primitive `GV_LINE` (a line) is written into a new line of the `Out` map. The table is then written and debugged with:
`db_execute_immediate(driver, &sql)!=DB_OK`
When all the points in the input maps have been processed, the procedure for closing the files and for de-allocating memory are called (lines 460-271).

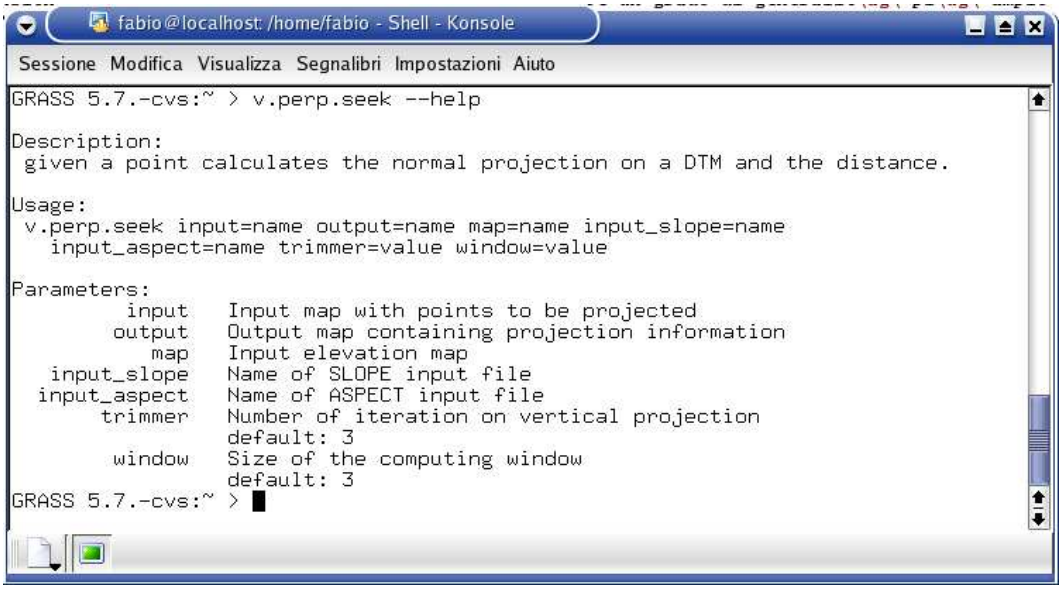

Figure 12: Video help of *v.perp.seek*

4. Discussion

4.1 Simple geometry

During module's tests and debug a series of plane geometry were investigated. An example is here reported.

On a plane defined as:

$$z = 75 - \frac{3}{4}x - \frac{1}{2}y$$

a set of points was projected. The module's results were compared with theoric values obtained from:

$$d = \frac{|Ax_P + By_P + Cz_P + D|}{\sqrt{A^2 + B^2 + C^2}} \tag{19}$$

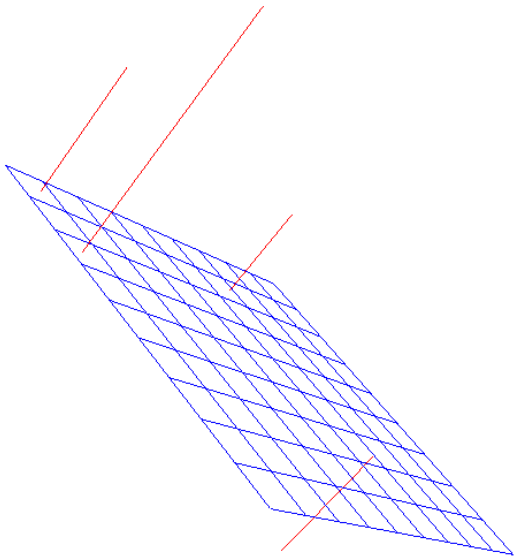

Figure 13: `nviz` rendering of the module output. Both above and under the plane were considered.

Tables (1) and (2) report the results of the comparison, the small errors that appear are due to computation and troncation errors. When a vector map is visualized, it is possible to inquire, with mouse-clicking, the primitive of interest obtaining the corresponding record of the database table linked to the vector map, as shown in figure (14).

Table 1: Comparison between model's and theoric results.

point	x_P	y_P	z_P	d	x_N	y_N	z_N
1	37	73	50	29.154164	20.758617	62.172399	28.344844
2	32	25	90	38.253242	10.689646	10.793081	61.586222
3	70	34	100	70.192838	30.896537	7.930995	47.862099
4	50	50	-40	38.996023	71.724147	64.482781	-11.034498
point	x_P	y_P	z_P	d_{teor}	x_{Nteor}	y_{Nteor}	z_{Nteor}
1	37	73	50	29.15416809	20.75862069	62.17241379	28.34482758610
2	32	25	90	38.25323966	10.68965517	10.79310345	61.58620689641
3	70	34	100	70.19283783	30.89655172	7.931034483	47.86206896525
4	50	50	-40	38.99602102	71.72413793	64.48275862	-11.03448275847

Table 2: Absolute errors

point	err_{dist}	err_{xN}	err_{yN}	err_{zN}
1	-4.09E-06	-3.69E-06	-1.48E-05	1.64E-05
2	2.34E-06	-9.17E-06	-2.24E-05	1.51E-05
3	1.69E-07	-1.47E-05	-3.95E-05	3.00E-05
4	1.98E-06	9.07E-06	2.24E-05	-1.52E-05

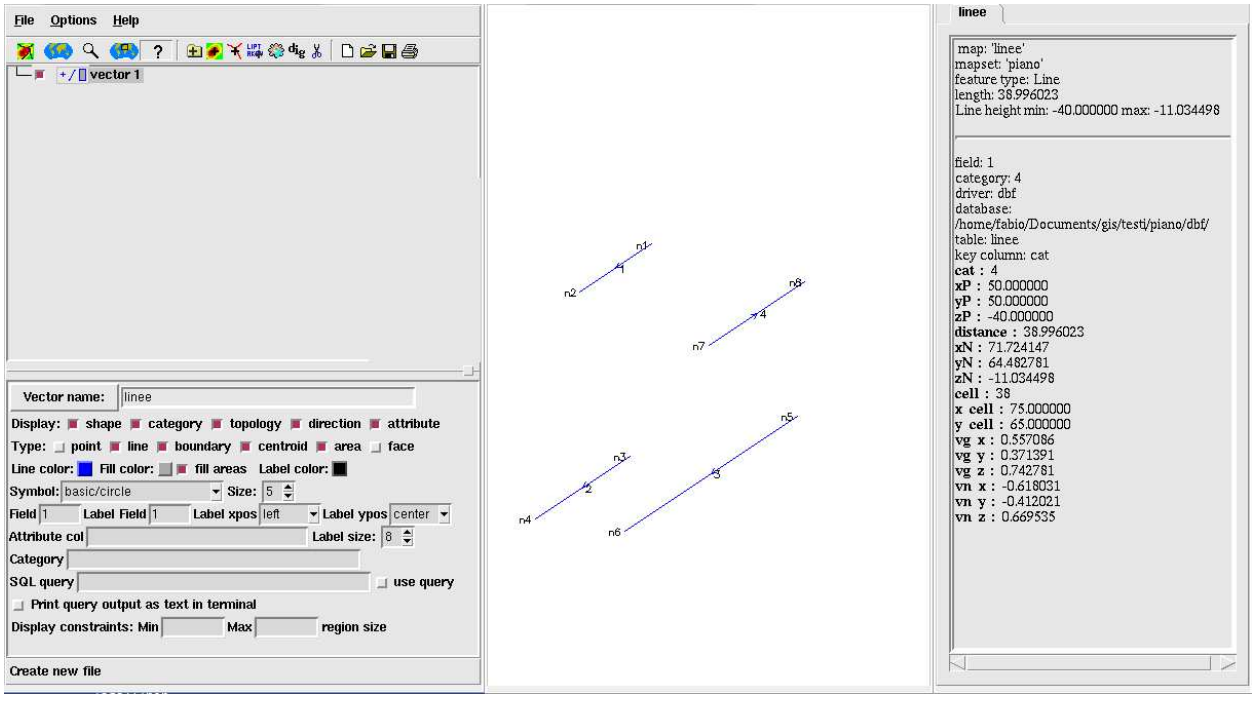

Figure 14: Map inquiring in real-time

4.2 Complex geometry

As for the raster-voxel apprach, [3], also the vector model was tested on an synthetic terrain surface and the results were compared using a vector grid reproducing the GRASS 3D-raster one. A structured grid, like voxel's one, can be easily reproduced in the vector approach, (some programs to produce grids of points in GRASS-like format have been developed by the author). The vector model supplies an higher grade of freedom is available: unstructured grid and irregular structured grid can be managed to fulfill user's needs.

Although the approach, both geometrically and computationally, is quite different, the distance evaluated with the *r3.isosurf* module were identical with those outputted by *v.perp.seek* and discussed here.

Figure 17 reports a side-effect of using the Horn's method to evaluate α e ϑ with the *r.slope.aspect* module. On the horizontal plane, some normals, which should be vertical, are actually inclinated. During the derivative's calculation, the 2nd-order differntial operator really debars the cell in wich the values have to be evaluate, and considers some cell wich are not horizontal. This lead to an "apparent slope" effect blunting the DEM and inclining the normal segment.

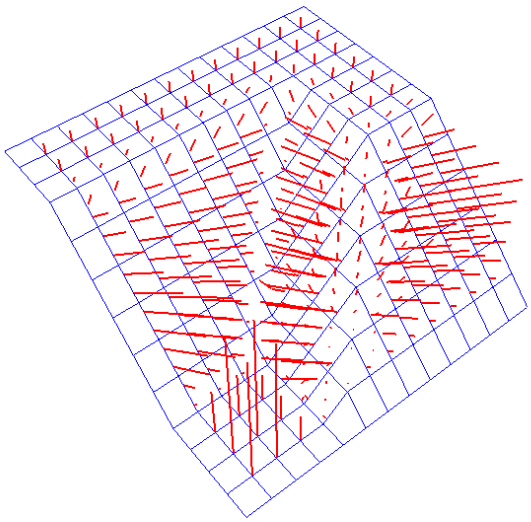

Figure 15: DEM used and lines normal to it's surface. WS point of view.

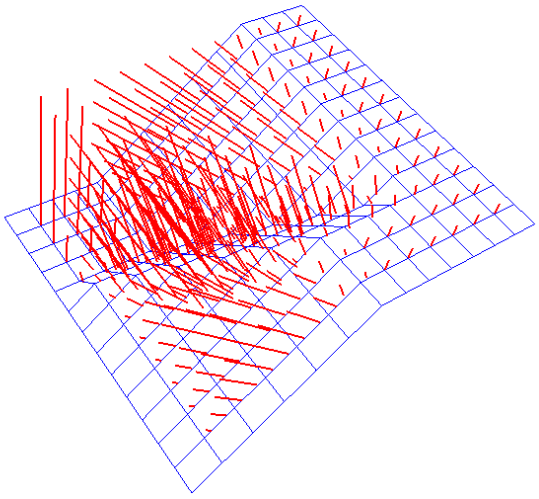

Figure 16: DEM used and lines normal to it’s surface. NE point of view.

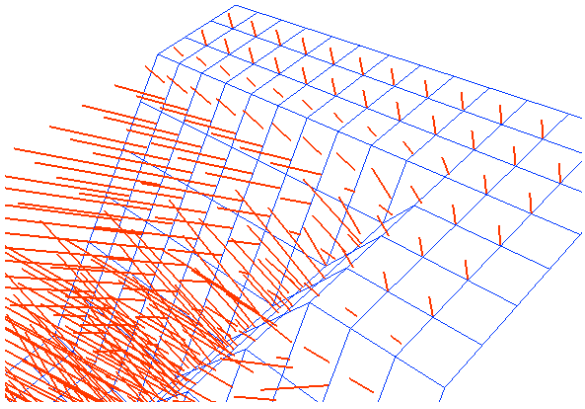

Figure 17: Blunting effect of the derivative Horn’s method.

5. Managing tables with PostgreSQL

In the implementation of the thermally driven wind's physical model, the normal distance is only one of the parameters used by Prandtl to describe the air fluxes. No details are given in this paper regarding the complete formulation of the Prandtl theory, [1] - [2]. It has to be noted that the terrain features (i.g. coverage, temperature) strongly influence the development of slope winds. In the Prandtl formulation the terrain influence is modeled through the C parameter that has to be evaluated on the entire terrain surface. This information is used to compute the wind velocity. In this work the Prandtl equation for the wind velocity is implemented using SQL capabilities, since all the parameters involved in the model are considered as vector attribute and managed using a DBMS. The native GRASS tables are built with an *xdB* driver. Some of the SQL functionalities required to the model implementation were not availble in GRASS during the work development and the PostgreSQL DBMS has therefore been used. At the time of writing the GRASS SQL functionalities are more complete. Conceptually the procedure developed and reported below and based on PostgreSQL can also be followed using the internal GRASS database driver and the SQL funcionalities implemented within GRASS.

A back-step is here necessary: the raster data are not directly read by DBMI. They can be made disposable transforming the raster map in a 3D-vector map using the *r.to.vect* module. For each center of the cell the value of the raster is set as an attribute. So, for each point to be projected, the C value where the projection falls is needed. In the database linked to the vector C map is searched only the record that satisfy:

$$cell_n = cat_C$$

Knowing the C values the Prandtl's equations can be easilly solved. When all the points will be processed the table is ready for being imported in GRASS again.

A simple step-by-step scheme describing how the Prandtl model has been implemented coupling GRASS and PostgreSQL capabilities is reported.

- Export GRASS database to another DBMI without changing the links to the vector map;
- use the values stored in the table to evaluate wind and temperature;
- add the results to the existing table;
- import into GRASS the new table and linking it correctly with the vector map.

From the point of view of Prantl's model the scheme followed is:

- from the point to project, the normal direction is darted, (`v.perp.seek`);
- where projection falls the value of C is read (PostgreSQL);
- Prandtl equations are solved.

First the PostgreSQL database has to be created and its virtual server initializated for DBMI's data management. Then, in GRASS, the table obtained using *v.perp.seek* is copyied into PostgreSQL's database transforming the *xdB* file in a *pg* one.

```
db.copy from_driver=dbf from_database=/GRASS/dbf
from_table=projections to_driver=pg to_database=Prandtl to_table=n
```

In the same way, the C raster is first transformed in a 3D vector map and then imported in PostgreSQL:

```
r.to.vect input=C_rast output=C_vect feature=point

db.copy from_driver=dbf from_database=/GRASS/dbf from_table=C_vect
to_driver=pg to_database=Prandtl to_table=C
```

At this point, the data are used in the Prandtl's equations. So the PostgreSQL's query is built:

```
CREATE TABLE wind
AS SELECT n.cat, (C.value*exp(-n.distance/elle)*cos(n.distance/elle))
AS delta_theta, (C.value*sqrt(g*beta*nu_h/nu_k/tau)*
*exp(-n.distance/elle)*sin(n.distance/elle))
AS ws, (C.value*sqrt(g*beta*nu_h/nu_k/tau)*
*exp(-n.distance/elle)*sin(n.distance/elle)*n.wind_x)
AS ws_x, (C.value*sqrt(g*beta*nu_h/nu_k/tau)*
*exp(-n.distance/elle)*sin(n.distance/elle)*n.wind_y)
AS ws_y, (C.value*sqrt(g*beta*nu_h/nu_k/tau)*
*exp(-n.distance/elle)*sin(n.distance/elle)*n.wind_z)
AS ws_z FROM n, C WHERE C.cat=n.cell;
```

The created table is then imported to GRASS:

```
db.copy from_driver=pg from_database=Prandtl from_table=wind
to_driver=dbf to_database=/GRASS/dbf to_table=wind
```

The table named *wind* is now in the GRASS database and it has to be linked to the vector map. First the old table is disconnected.

```
v.db.connect map=projections driver=dbf
database=/GRASS/dbf table=projections key=cat field=1 -d
```

and the vector map is added:

```
v.category input=projections output=wind option=add field=2

v.db.connect map=projections driver=dbf
database=/GRASS/dbf table=projections key=cat field=1

v.db.connect map=projections driver=dbf database=/GRASS/dbf
table=wind key=cat field=2
```

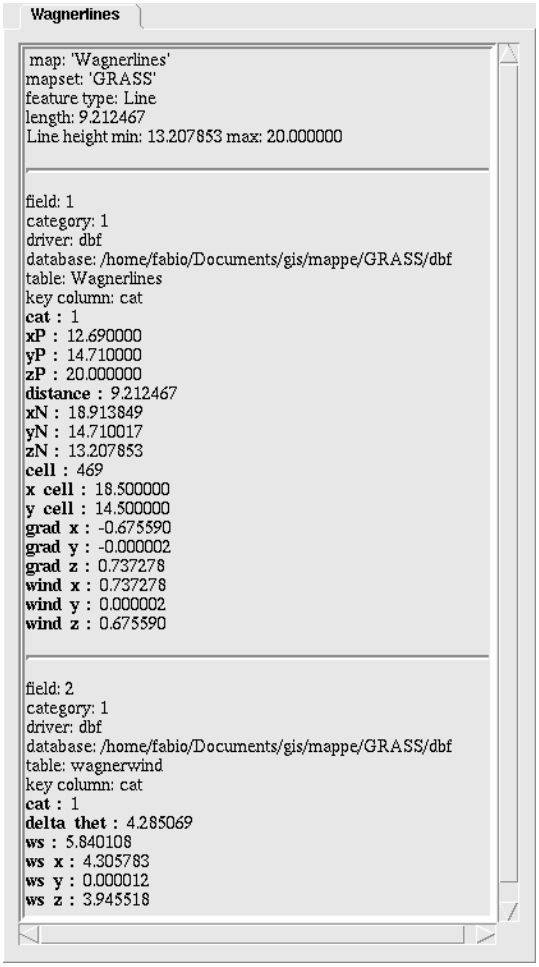

Figure 18: Visualization of the new data developement.

The result is shown in figure 18. Clicking on the map a table containig projection, wind and temperature data opens. The figure (19) reports a 3D visualization of the wind stream traces where the colors are given by temperature magnitude.

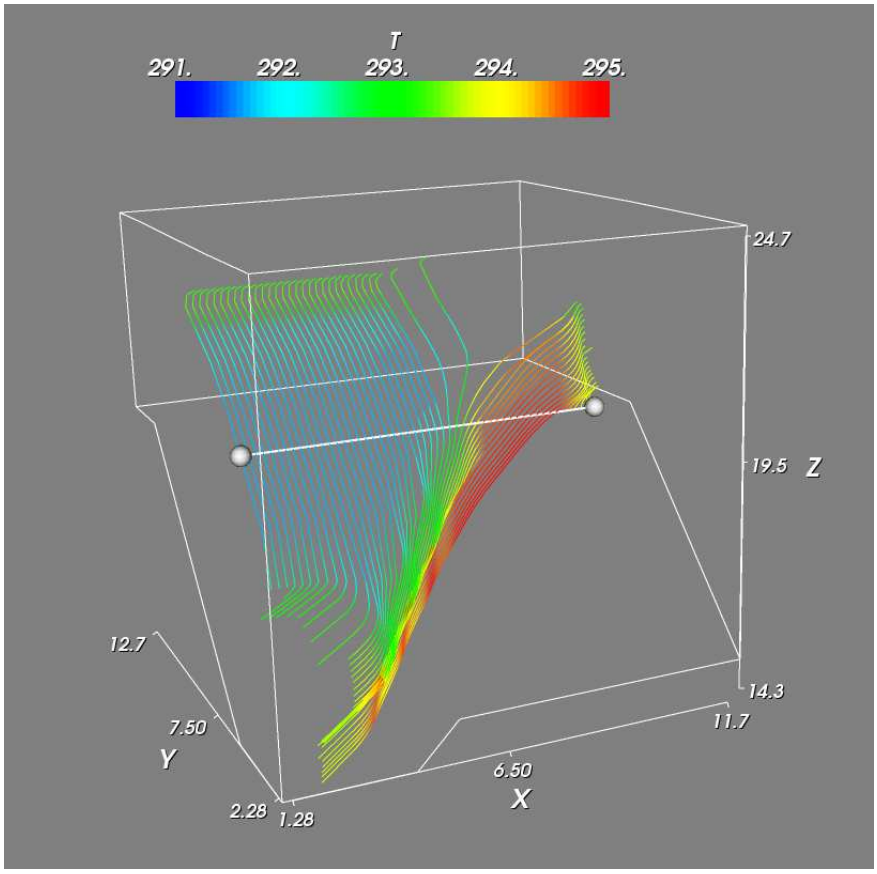

Figure 19: Steam traces of wind. Color is given by temperature.

6. Conclusions

A physical model for the description of thermally driven slope winds has been successfully implemented using the GRASS GIS. A GIS supplies a proper environment for the development of local atmospheric models allowing the management of referenced information and the integration of models' results into wider enviromental analyses.

The physical model is based on the Prandtl formulation, [1] - [2], which allows to evaluate the wind velocity and the air temperature in the first meters (approximately 100 m) of the atmosphere over the terrain surface. Some information related to the terrain surface, as coverage and solar radiation, and the distance of the air particle from the surface, measured along the direction normal to the terrain surface, are the main quantities the Prandtl equations depend on. This paper reports a detailed description of the algorihtm defined to evaluate the normal distance of a point in the space and how this algorithm has been implemented in a new GRASS module `v.perp.seek`. The module takes advantage of the new GRASS vector model which supplies a powerful environmet for the implementation and the study

of three-dimentional physical models. The GRASS GIS capabilities to managed three-dimentional vector entities, to link vector attributes to database tables and to perform SQL statements and queries constitute a very powerful and flexible tool in performing enviromental analyses.

In [3] the Prandtl physical model is reported in detail along with the description on how the model had been implemented following a raster approch based on the GRASS 3D raster functionalities. The comparison, in terms of computational weight and usage flexibility, between the 3D raster approach and the 3D vector approach, reported in this paper, can be found in [10]. Briefly, depending on the model and on the geometry of the involved domain, it is possible to choose between the raster and the vector approach. For regular domains the raster approach shows better computational performance while the vector approach is a more flexible solution in managing irregular grids of points.

References

[1] Prandtl L. *Strömungslehre*. Verlag Vieweg & Sohn, 1942.

[2] Defant F. Zur theorie der Hangwinde, nebst Bemerkungen zur Theorie der Berg- und Talwinde (A theory of slope winds, along with remarks on the theory of mountain winds and valley winds.). In *Archiv für Meteorologie Geophysik und Bioklimatologie*, volume 1 of *Ser. A*, pages 421–450. 1949.

[3] Ciolli M., de Franceschi M., Rea R., Vitti A., Zardi D., and Zatelli P. Development and application of 2d and 3d grass modules for simulation of thermally driven slope winds. *Transactions in GIS*, 8(2):191–209, 2004.

[4] Blazek R., Neteler M., and Micarelli R. The new GRASS 5.1 vector architecture. In B. Benciolini, M. Ciolli, and P. Zatelli, editors, *Open source GIS – GRASS users conference 2002, Trento, Italy, 11-13 September 2002*. University of Trento, Italy, 2002.

[5] GRASS Developement Team. *GRASS 5.7 Reference Manual-Vector Architecture*, June 2003. Draft Version.

[6] Henne S., Furger M, Nyeki S., Steinbacher M.and Neininger B, de Wekker S.F.J., Dommen J., Spichtinger N, Stohl A., and Prévôt A.S.H. Quantification of topographic venting of boundary layer air to the free troposphere. *Atmospheric Chemistry and Phisics*, 1(4):497–509, 2004. SRef-ID: 1680-7324/acp/2004-4-497.

[7] Horn B.K.P. Hill shading and the reflectance map. In IEEE, editor, *Procs. of the IEEE*, volume 1, pages 14–47, 1981.

[8] Rampanelli G. and Zardi D. Analysis of airborne data and identification of thermal structures with eostatical techniques. In *14th Symosium on Boundary Layer and Turbolence, Aspen (CO)*, pages 239–241, 7-11 August 2000.

[9] Apostol T.M. *Analisi 2*, volume 3 of *Calcolo*. Bollati Boringhieri s.r.l., 8th edition, 1992. ISBN 88-339-5071-9.

[10] Vitti A., Zatelli P., and Zottele F. 3d vector approach to local thermally driven slope winds modeling. In Raghavan V. and Santitamnont P., editors, *Proceedings of the Free/Libre and Open Source Software (FOSS) for Geoinformatics: GIS - GRASS Users Conference*, Bangkok (TH), 12-14 settembre 2004.

[11] Vitti A. Modellazione tridimensionale mediante *GIS-GRASS* di venti di pendio forzati termicamente. Master thesis, Università degli Studi di Trento, Facoltà di Ingegneria, Trento, 2002.

A Source code

```
 1 /***************************************************************************
 2                          vectorinvectorout.c  -  description
 3                             -------------------
 4     date                 : sep 04 2004
 5     copyright            : (C) 2004 by fabio
 6     email                : fabio@localhost


 7     MODULE               : v.perp.seek -- Given a set of points, finds
 8                            their the normal projection on a DTM and the
 9                            distance colleting informations in a linked
10                            table(GRASS 5.7)


11     PURPOSE              : A general module to find normal projections
12                            using GRASS 5.7 3D_vect capabilities


13  ***************************************************************************/


14 /***************************************************************************
15  *
16  *   This program is free software; you can redistribute it and/or modify
17  *   it under the terms of the GNU General Public License as published by
18  *   the Free Software Foundation; either version 2 of the License, or
19  *   (at your option) any later version. Read the file COPYING that comes
20  *   with GRASS for details.
21  *
22  ***************************************************************************/


23 #include <stdio.h>
24 #include <stdlib.h>
25 #include <string.h>
26 #include <ctype.h>
27 #include <math.h>
28 #include <unistd.h>/*necessary to db_start_driver()*/
29 #include "gis.h"/*library for accessing the database*/
30 #include "Vect.h" /*ANSI prototypes for the lib/vector/Vlib functions*/
31 #include "dbmi.h"


32 #define withz 1
33 #define PI 3.14159265359
34 #define MY_NULL -999.


35 struct Point_3D
36 {
37     double east;
38     double north;
39     double z;
40 };
41 static struct line_cats *Cats;
42 static struct line_pnts *Points;
43 static int point_cat;
44 static dbString sql;
45 static dbDriver *driver;
```

```
46 static struct field_info *fi;


47 void write_line(struct Map_info *Out, double x1, double y1, double z1,double x2,
48 double y2, double z2, int point_cat, double distance, int cell, double x3, double y3,
49 double grad_x, double grad_y, double grad_z, double w_x, double w_y, double w_z)
50 {
51 char buf[2048];


52 G_debug(3,"write_line ()");


53 Vect_reset_line(Points);
54 Vect_reset_cats(Cats);
55 Vect_append_point(Points, x1, y1, z1);
56 Vect_append_point(Points, x2, y2, z2);
57 Vect_cat_set(Cats, 1, point_cat);
58 Vect_write_line(Out,GV_LINE,Points,Cats);
59 db_zero_string(&sql);
60 sprintf(buf,"insert into %s values ( %d, %lf, %lf, %lf, %lf, %lf, %lf, %lf, %d, %lf, %lf, %lf, %lf, %lf, %
61 fi->table, point_cat, x1, y1, z1, distance, x2, y2, z2, cell, x3, y3, grad_x, grad_y, grad_z, w_x, w_y, w_
62 db_append_string(&sql, buf);
63 if(db_execute_immediate(driver, &sql) != DB_OK)
64 {
65 G_warning("Cannot inser new row: %s",db_get_string(&sql));
66 }
67 point_cat++;
68 }


69 int main(int argc,char **argv)
70 {
71 struct GModule *module;
72 struct Option *outvect, *DTMin, *slopein, *aspectin, *trimmerin, *maskin, *invect;
73 double trimmer;
74 int mask;
75 char *dtm_name, *slope_name, *aspect_name;
76 char *dtm_mapset, *slope_mapset, *aspect_mapset;
77 RASTER_MAP_TYPE dtm_type, slope_type, aspect_type;
78 int i, n_el, j;
79 char buf[1024];
80 int dtm_fd, slope_fd, aspect_fd;
81 struct Cell_head header;
82 int n_rows, n_cols;
83 float rows_res, cols_res;
84 float north, south, east, west;
85 void *dtm_raster, *slope_raster, *aspect_raster, *pointer;
86 struct Point_3D ptp, pv, pov, pq, PN;/*points*/
87 int jump_x, jump_y, jump_x_g, jump_y_g, pointer_row, pointer_col;
88 double tracer;
89 struct Point_3D **pod;/*matrix of points*/
90 double **distance;
91 double dist_min, norma, normal_distance, norma_m_s;
92 struct Point_3D direction, versor_d, versor_r, gradient, versor_g, m_s, versor_n;/*vectors*/
93 double g_wt;
94 double slope, aspect;
95 char *vector_mapset;
96 struct Map_info vector_map_old, vector_map_new;
97 int field;
98 int line,nlines;
99 struct line_cats *LCats;
```

```
100 struct line_pnts *LPoints;
101 int type;
102 int cell_on_dtm;
103 double x_cell, y_cell;


104 G_gisinit(argv[0]); /*GRASS initialization*/


105 module = G_define_module();
106 module->description=
107 "given a point calculates the normal projection "
108 "on a DTM and the distance.";


109 invect = G_define_standard_option(G_OPT_V_INPUT);
110 invect->description = "Input map with points to be projected";


111 outvect = G_define_standard_option(G_OPT_V_OUTPUT);
112 outvect->description = "Output map containing projection information";


113 DTMin = G_define_standard_option(G_OPT_R_MAP);
114 DTMin->description = "Input elevation map ";


115 slopein = G_define_option();
116 slopein->key           = "input_slope";
117 slopein->description   = "Name of SLOPE input file";
118 slopein->type          = TYPE_STRING;
119 slopein->required      = YES;
120 slopein->multiple      = NO;
121 slopein->gisprompt     = "old,cell,raster";


122 aspectin = G_define_option();
123 aspectin->key          = "input_aspect";
124 aspectin->description  = "Name of ASPECT input file";
125 aspectin->type         = TYPE_STRING;
126 aspectin->required     = YES;
127 aspectin->multiple     = NO;
128 aspectin->gisprompt    = "old,cell,raster";


129 trimmerin = G_define_option();
130 trimmerin->key         = "trimmer";
131 trimmerin->description = "Number of iteration on vertical projection";
132 trimmerin->type        = TYPE_INTEGER;
133 trimmerin->required    = YES;
134 trimmerin->multiple    = NO;
135 trimmerin->answer      = "3";


136 maskin = G_define_option();
137 maskin->key            = "window";
138 maskin->description    = "Size of the computing window";
139 maskin->type           = TYPE_INTEGER;
140 maskin->required       = YES;
141 maskin->multiple       = NO;
142 maskin->answer         = "3";
```

```
143 if(G_parser (argc, argv))
144 exit(EXIT_FAILURE);
145 sscanf(trimmerin->answer, "%lf", &trimmer);
146 sscanf(maskin->answer, "%d", &mask);


147 type = GV_POINT;//set the primitives to be read as points


148 if(trimmer <= 1)
149 G_fatal_error("The number of iteration has to be >1\n");
150 if(mask <= 1)
151 G_fatal_error("The dimension of the neighborhood tool has to be >1\n");


152 /*vector geometry-table read-write initialization*/
153 Cats = Vect_new_cats_struct();
154 Points = Vect_new_line_struct();
155 LCats = Vect_new_cats_struct();
156 LPoints = Vect_new_line_struct();
157 db_init_string(&sql);


158 /*vector processing*/
159 vector_mapset = G_find_vector2(invect->answer, NULL);
160 if(vector_mapset == NULL)
161 G_fatal_error("Vector file [%s] not available\n", invect->answer);
162 Vect_set_open_level(2);
163 Vect_open_old(&vector_map_old, invect->answer, vector_mapset);


164 Vect_open_new(&vector_map_new, outvect->answer, withz);
165 Vect__init_head(&vector_map_new);
166 Vect_hist_command(&vector_map_new);
167 /*writes command info on history file*/
168 fi = Vect_default_field_info(&vector_map_new, 1, NULL, GV_1TABLE);
169 /*get default information about link to database for new db link*/
170 Vect_map_add_dblink(&vector_map_new, 1, NULL, fi->table, "cat", fi->database, fi->driver);
171 /*add new db connection to map info structure*/
172 driver = db_start_driver_open_database(fi->driver, fi->database);
173 if(driver == NULL)
174 G_fatal_error("Cannot open database %s by driver %s", fi->database, fi->driver);
175 sprintf (buf, "create table %s ( cat int, xP double precision, yP double precision, zP double precision, c
176 db_append_string(&sql, buf);
177 if(db_execute_immediate(driver, &sql)!= DB_OK)
178 {
179 db_close_database_shutdown_driver(driver);
180 G_fatal_error("Cannot create table: %s", db_get_string(&sql));
181 }


182 /*raster processing*/
183 dtm_name = DTMin->answer;
184 dtm_mapset = G_find_cell2(dtm_name, NULL);
185 if(dtm_mapset == NULL)
186 G_fatal_error("DTM_file [%s] not available\n", dtm_name);
187 if((dtm_fd = G_open_cell_old(dtm_name, dtm_mapset)) < 0)
188 G_fatal_error("Not able to open DTM_raster <%s@%s>\n", dtm_name, dtm_mapset);
189 G_get_cellhd(dtm_name, dtm_mapset, &header);
190 n_rows = header.rows;/* number of rows in the data */
191 n_cols = header.cols;/* number of columns in the data */
```

```
192 rows_res = header.ns_res;/* North to South cell size */
193 cols_res = header.ew_res;/* East to West cell size */
194 north = header.north;/* coordinates of layer */
195 south = header.south;
196 east = header.east;
197 west = header.west;
198 dtm_type = G_raster_map_type(dtm_name, dtm_mapset);
199 dtm_raster = G_allocate_raster_buf(dtm_type);


200 slope_name =  slopein->answer;
201 slope_mapset = G_find_cell2(slope_name, NULL);
202 if(slope_mapset == NULL)
203 G_fatal_error("slope_file [%s] not available\n", slope_name);
204 if((slope_fd = G_open_cell_old(slope_name, slope_mapset)) < 0)
205 G_fatal_error("Not able to open slope_raster <%s@%s>\n", slope_name, slope_mapset);
206 slope_type = G_raster_map_type(slope_name, slope_mapset);
207 slope_raster = G_allocate_raster_buf(slope_type);


208 aspect_name = aspectin->answer;
209 aspect_mapset =  G_find_cell2(aspect_name, NULL);
210 if(aspect_mapset == NULL)
211 G_fatal_error("aspect_file [%s] not available\n", aspect_name);
212 if ((aspect_fd = G_open_cell_old(aspect_name, aspect_mapset)) < 0)
213 G_fatal_error("Not able to open aspect_raster <%s@%s>\n", aspect_name, aspect_mapset);
214 aspect_type = G_raster_map_type(aspect_name, aspect_mapset);
215 aspect_raster = G_allocate_raster_buf(aspect_type);


216 /*point acquiring and developement of data*/
217 point_cat = 1;
218 nlines = Vect_get_num_lines(&vector_map_old);
219 for(line = 1; line <= nlines; line++)
220 {
221 int ltype, line_cat;
222 G_debug(5, "line = %d", line);
223 printf("Processing line %d of %d ...",line,nlines);
224 ltype = Vect_read_line(&vector_map_old, LPoints, LCats, line);
225 if(!(ltype & type)) continue;
226 Vect_cat_get(LCats, field, &line_cat);
227 ptp.east = LPoints->x[0];
228 ptp.north = LPoints->y[0];
229 ptp.z = LPoints->z[0];
230 jump_x= floor((ptp.east-west)/cols_res);
231 jump_y= n_rows-ceil((ptp.north-south)/rows_res);
232 /*building the vertical projection*/
233 pv.east = ptp.east;
234 pv.north = ptp.north;
235 G_get_raster_row(dtm_fd, dtm_raster, jump_y, dtm_type);
236 pointer = dtm_raster;
237 pointer = G_incr_void_ptr(pointer, jump_x*G_raster_size(dtm_type));
238 switch(dtm_type)
239 {
240 case CELL_TYPE: pv.z = (double)*((CELL*) pointer);break;
241 case FCELL_TYPE: pv.z = (double)*((FCELL*) pointer);break;
242 case DCELL_TYPE: pv.z = (double)*((DCELL*) pointer);break;
243 default: pv.z = MY_NULL;
244 }
245 if ( pv.z >= ptp.z)
246 {
247 g_wt = -1.;
```

```
248 }
249 else
250 {
251 g_wt = 1.;
252 }
253 tracer =(ptp.z-pv.z)/trimmer;
254 n_el = 1;
255 direction.east = 0.;
256 direction.north = 0.;
257 PN.east = pv.east;
258 PN.north = pv.north;
259 /*going up- or downward the vertical until matching PTP*/
260 do
261 {
262 pov.east = pv.east;
263 pov.north = pv.north;
264 pov.z = pv.z+n_el*tracer;
265 /*placing every neighbourhood tool directly in the last normal projection*/
266 jump_x= floor((PN.east-west)/cols_res);
267 jump_y= n_rows-ceil((PN.north-south)/rows_res);
268 if((mask%2) == 0)
269 mask++;
270 pod = (struct Point_3D**)G_malloc((signed)(mask*sizeof(struct Point_3D*)));
271 for (i = 0; i <= (mask-1); i++)
272 {
273 pod[i] = (struct Point_3D*)G_malloc((signed)(mask*sizeof(struct Point_3D)));
274 for (j = 0; j <= (mask-1); j++)
275 {
276 pod[i][j].east =(jump_x-0.5*mask+j+1)*rows_res;
277 pod[i][j].north =(n_rows-jump_y+0.5*mask-i-1)*cols_res;
278 if((pod[i][j].east < east && pod[i][j].east > west) &&(pod[i][j].north < north && pod[i][j].north > south)
279 {
280 G_get_raster_row(dtm_fd, dtm_raster, jump_y+i-(int)floor(mask/2.), dtm_type);
281 pointer = dtm_raster;
282 pointer = G_incr_void_ptr(pointer, (jump_x+j-(int)floor(mask/2.))*G_raster_size(dtm_type));
283 switch(dtm_type)
284 {
285 case CELL_TYPE: pod[i][j].z = (double)*((CELL*)pointer);break;
286 case FCELL_TYPE: pod[i][j].z = (double)*((FCELL*)pointer);break;
287 case DCELL_TYPE: pod[i][j].z = (double)*((DCELL*)pointer);break;
288 default: pod[i][j].z =  MY_NULL;
289 }
290 }
291 else
292 pod[i][j].z = MY_NULL;
293 }
294 }
295 distance =(double**)G_malloc((signed)(mask*sizeof(double*)));
296 for(i = 0; i <= (mask-1); i++)
297 {
298 distance[i] = (double *)G_malloc((signed)(mask*sizeof(double)));
299 for(j = 0; j <= (mask-1); j++)
300 distance[i][j] = sqrt(pow((pod[i][j].east-pov.east),2.)+pow((pod[i][j].north-pov.north),2.)+pow((pod[i][j
301 }
302 /*finding the minimum distance*/
303 pointer_row = (int)(floor(mask/2.));
304 pointer_col = pointer_row;
305 dist_min = distance[pointer_row][pointer_col];
306 for(i = 0; i <= (mask-1); i++)
307 {
308 for(j = 0; j <= (mask-1); j++)
309 if(distance[i][j] <= dist_min)
```

```
{
dist_min = distance[i][j];
pointer_row = i;
pointer_col = j;
}
}
pq.east = west + (jump_x-mask/2+pointer_col+0.5)*cols_res;
pq.north = south+(n_rows-jump_y+mask/2-pointer_row-0.5)*rows_res;
pq.z = pod[pointer_row][pointer_col].z;
G_free(pod);
G_free(distance);
n_el++;
direction.east = pq.east-pov.east;
direction.north = pq.north-pov.north;
direction.z = pq.z-pov.z;
versor_d.east = direction.east/dist_min;
versor_d.north = direction.north/dist_min;
versor_d.z = direction.z/dist_min;
versor_r.east = -versor_d.east;
versor_r.north = -versor_d.north;
versor_r.z = -versor_d.z;
/*finding values of slope and aspect*/
jump_x_g = floor((pq.east-west)/cols_res);
jump_y_g = n_rows-ceil((pq.north-south)/rows_res);
G_get_raster_row (slope_fd, slope_raster, jump_y_g, slope_type);
pointer = slope_raster;
pointer = G_incr_void_ptr(pointer, jump_x_g*G_raster_size(slope_type));
switch(slope_type)
{
case CELL_TYPE: slope =  (double)*((CELL *)pointer);break;
case FCELL_TYPE: slope =  (double)*((FCELL *)pointer);break;
case DCELL_TYPE: slope =  (double)*((DCELL *)pointer);break;
default: slope = MY_NULL;
}
G_get_raster_row(aspect_fd, aspect_raster, jump_y_g, aspect_type);
pointer = aspect_raster;
pointer = G_incr_void_ptr(pointer, jump_x_g * G_raster_size(aspect_type));
switch(aspect_type)
{
case CELL_TYPE: aspect = (double)*((CELL *)pointer);break;
case FCELL_TYPE: aspect =  (double)*((FCELL *)pointer);break;
case DCELL_TYPE: aspect =  (double)*((DCELL *)pointer);break;
default: aspect = MY_NULL;
}
/*converting slope and aspect in greadient components*/
if(slope == 0. && aspect == 0.)
{
gradient.north = 0.;
gradient.east = 0.;
}
if((aspect >= 0. && aspect < 90.) || (aspect > 270. && aspect <= 360.))
{
gradient.east = tan(PI*slope/180)/sqrt(1+pow(tan(PI*aspect/180),2.));
gradient.north = tan(PI*aspect/ 180)*gradient.east;
}
if(aspect == 90.)
{
gradient.east = 0.;
gradient.north = tan(PI*slope/180);
}
if(aspect > 90. && aspect <= 180.)
{
```

```
gradient.east = -tan(PI*slope/180)/sqrt(1+pow(tan(PI*(1-aspect/180)),2.));
gradient.north = tan(PI*(1-aspect/180))*fabs(gradient.east);
}
if(aspect > 180. && aspect < 270.)
{
gradient.east = -tan(PI*slope/180)/sqrt(1+pow(tan(PI*(1+aspect/180)),2.));
gradient.north = -tan(PI*(1+aspect/180))*fabs(gradient.east);
}
if(aspect == 270.)
{
gradient.east = 0.;
gradient.north = -tan(PI*slope/180);
}
gradient.z = 1.;
norma = sqrt(pow(gradient.east,2.)+pow(gradient.north,2.)+pow(gradient.z,2.));
versor_g.east = gradient.east / norma;
versor_g.north = gradient.north / norma;
versor_g.z = gradient.z / norma;
/*finding the normal projection*/
normal_distance = dist_min*fabs((versor_r.east*versor_g.east+versor_r.north*versor_g.north+versor_r.z*vers
PN.east = pov.east-normal_distance*g_wt*versor_g.east;
PN.north = pov.north-normal_distance*g_wt*versor_g.north;
PN.z = pov.z-normal_distance*g_wt*versor_g.z;;
/*computing the maximum_slope direction (normal to gradient scoping the maximum slope)*/
if(versor_g.z == 1.)
{
m_s.east = 1.;
m_s.north=0.;
m_s.z= 0.;
}
else
{
m_s.east = - gradient.east;
m_s.north = - gradient.north;
m_s.z = - (gradient.east*m_s.east+gradient.north*m_s.north)/gradient.z;
}
norma_m_s = sqrt(pow(m_s.east,2.)+pow(m_s.north,2.)+pow(m_s.z,2.));
versor_n.east = m_s.east/norma_m_s;
versor_n.north = m_s.north/norma_m_s;
versor_n.z = m_s.z/norma_m_s;
/*matching the cell on DTM where the projection falls*/
cell_on_dtm = (n_rows-ceil((PN.north-south)/rows_res))*n_cols+ceil((PN.east-west)/cols_res);
x_cell =(floor(PN.east/cols_res)+ 0.5)*cols_res;
y_cell =(floor(PN.north/rows_res)+ 0.5)*rows_res;
}
while(pov.z != ptp.z);
write_line(&vector_map_new, ptp.east, ptp.north, ptp.z, PN.east, PN.north, PN.z, line, normal_distance, ce
printf(" done\n");
}
db_close_database_shutdown_driver(driver);
Vect_build(&vector_map_new, stdout);
Vect_close(&vector_map_old);
Vect_close(&vector_map_new);
G_free(dtm_raster);
G_free(slope_raster);
G_free(aspect_raster);
G_free(pointer);
G_close_cell(dtm_fd);
G_close_cell(slope_fd);
G_close_cell(aspect_fd);
return EXIT_SUCCESS;
}
```