Volume

1

DIVESTCO INC

Programmers Notes

PICScript Users Guide

DIVESTCO INC

PICScript Users Guide

Divestco

520 3 Ave SW • Suite 400
Phone (587) 952-8000 • Fax (587) 952-8370

Date: May 9, 2012
Version 5.9
By: Chris Gibson

Table of Contents

Chapter 1

Overview

Using a simple programming language to extend the capabilities of Horizon / Grid math.

In order to run advanced math operations on horizons & grids, a programming language called "**PICScript**" has been developed. Based loosely on C, this language includes most of the original functions, updated versions of others, and some brand new ones.

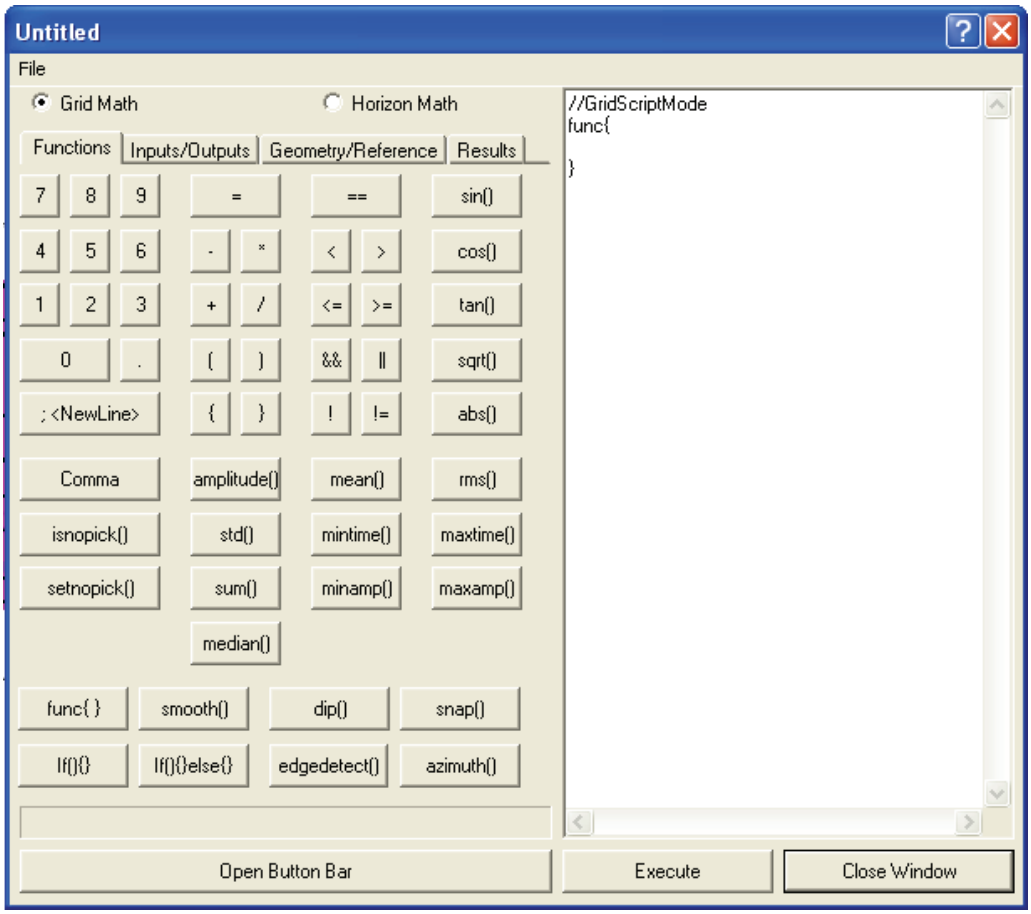
Not all of the functions were carried over; the **clip** & **rep** were not copied due to the fact that the **if/then/else** can be used to produce the same results.

Opening the dialog

The PICScript dialog can be accessed from the tools menu of both the Map & Data windows of the program.

Dialog usage

Once you open the dialog you will see the following:



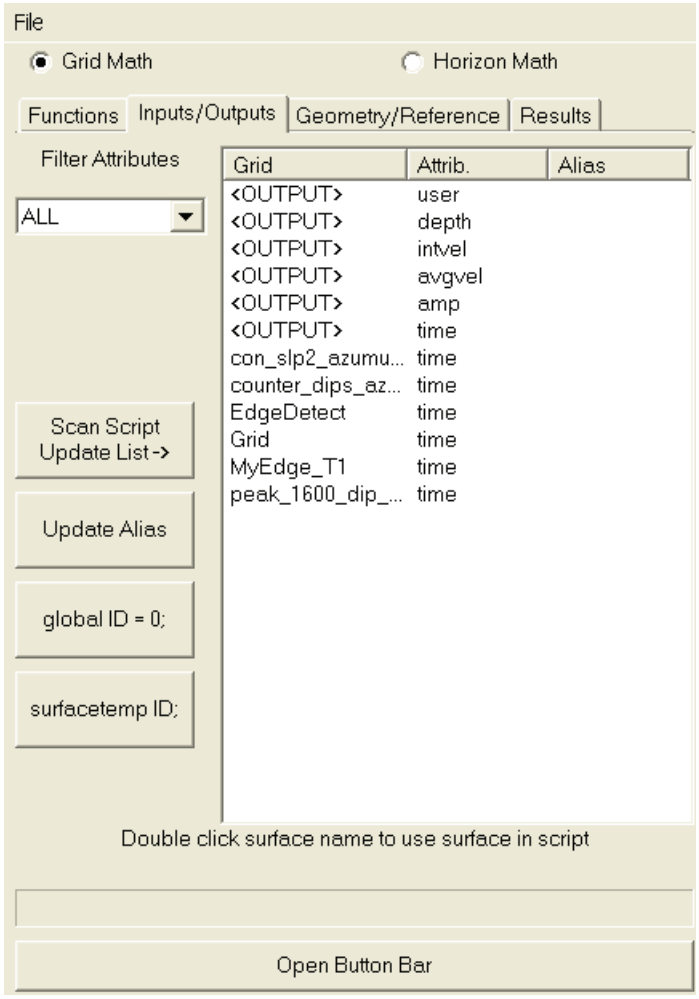
Screen shot of the startup screen for the PICScript dialog.

The script editor pane is where you can type the script that is to be run. **The Grid Math / Horizon Math** radio buttons set the mode for math execution. The tabs allow you to access different pages for the building of the script; the currently highlighted page gives the user a series of quick buttons to build the script. Take note of the **Open Button Bar** button on the bottom, this lets you open the quick buttons in another dialog window. This window is initially docked to the main window; you can drag this new window away from the main window if you wish.

Under the file menu you will find a series of options which will allow you to create **New** script**, Open** an existing script**, Save** the current script (requesting a name if the

current script does not have one)**, Save As** (to a new file name) and **Exit** (and requesting to save any changes)

INPUTS / OUTPUTS



Screen shot of the Inputs / Outputs tab

This gives you a list of all the currently available horizons / grids that are in the database. As you can see, the horizon mode is selected. If you double click on one of the surfaces, you can cause that surface to be used in the script. If that surface has no alias in the list, then a declaration statement will also be entered into the script.

- **<OUTPUT>** enters an output declaration, uses the attribute as the alias, grid mode also has intvel, avgvel & depth attributes as well.

- **<SEISMIC>** (not available in grid mode) double click on one of these and you will be asked for an alias to use. This gives you access to seismic data in the script. See the declaration section for more information on the usage.

- **Horizon / Grid names** If no alias is present, it will also build the declaration as well.

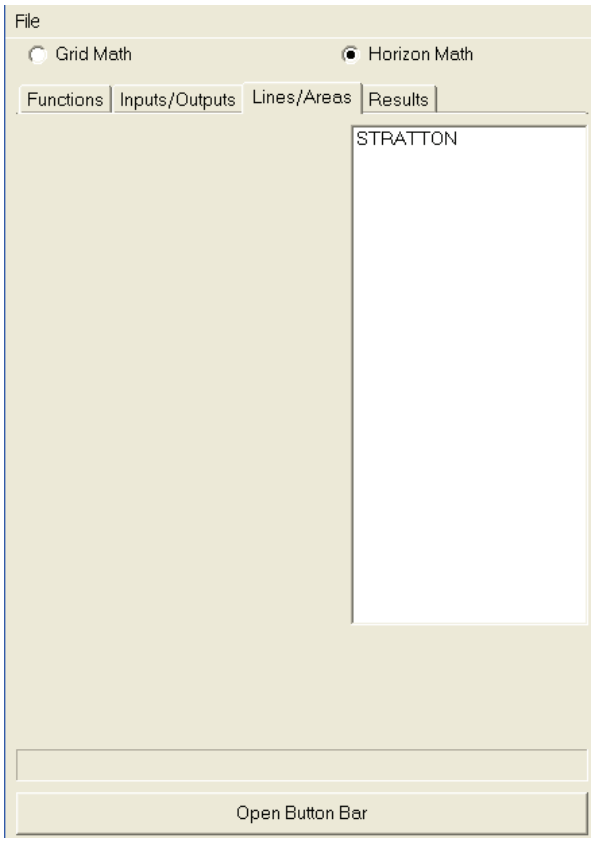
The button marked **Scan Script Update List** will read the script that is currently loaded, and display the alias information in the list. If a surface in the script is not present in the list, it will stop with a warning, and highlight the offending surface.

The **Update Alias** button moves through each surfacein declaration, selects it, and allows you to change the input surface by double clicking it in the list.

The buttons marked **global id = 0** and **surfacetemp** will insert the declarations for these variables. Both will ask for an alias to use.

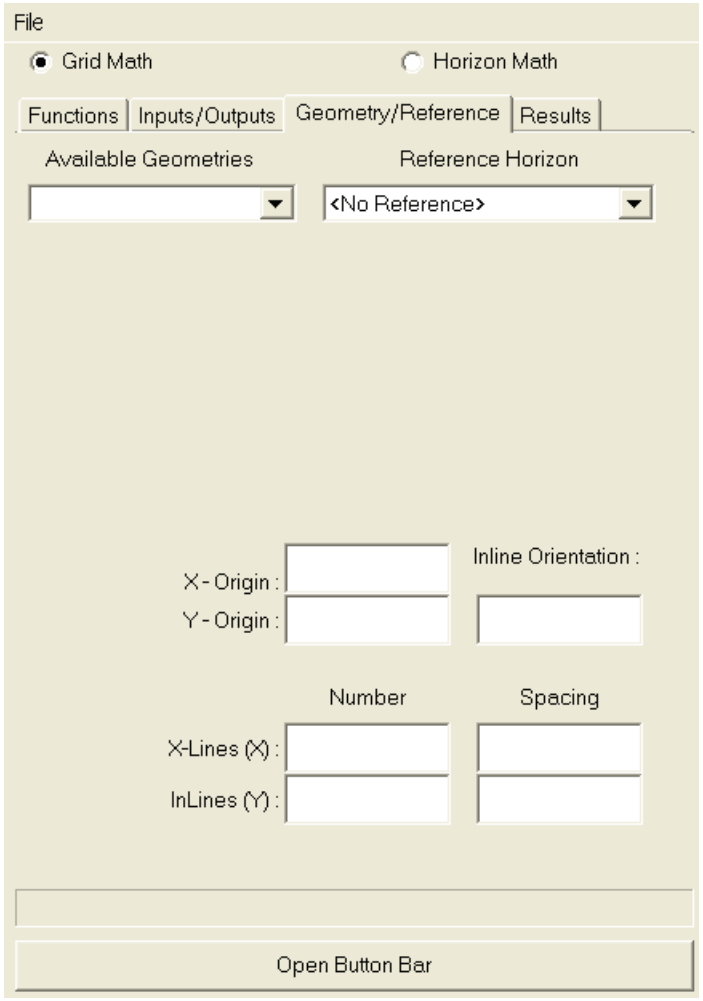
(Horizon mode only)

Just highlight the lines that you wish the script to be run on.



Lines / Areas tab

(Grid mode only)



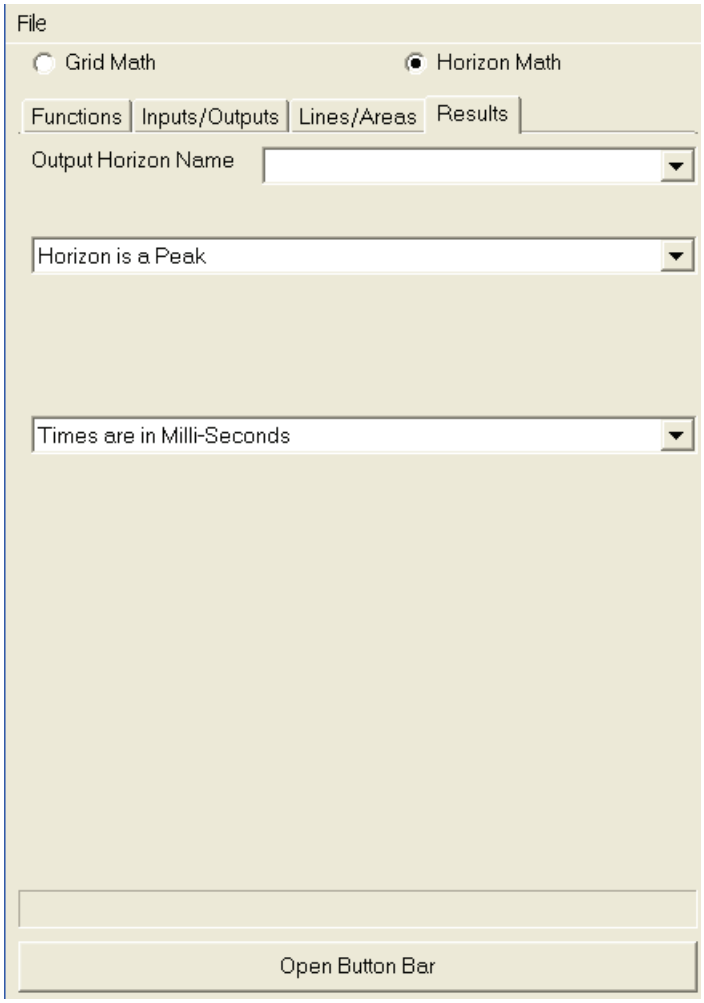
Geometry / Reference Tab

The drop down box lets you select from existing grid geometries to base the output grid on, and/or you can enter your own values in the boxes provided.

The reference horizon lets you attach the resultant grid to an existing horizon in the database.

RESULTS

This tab lets you set parameters that affect the results of the script.



Example of the Results tab

The **Output surface name** is the name to give the output horizon or grid depending on the mode. This is a dropdown list; you can select an existing name, or enter a new name. The name may only contain letters and numbers or the underscore or dash. If you enter a name that is not valid, you will receive an error message when you attempt to run the script.

The drop down that currently says *Horizon is a Peak* lets you specify the picking mode for the created horizon, and this will only be visible in horizon mode.

When working with times in the script, it is important to know what times units you are using. For example take the statement

```
Time = horz + 10;
```

What does the 10 refer to, 10 seconds, 10 microseconds? This lets you specify what the units are, in effect, this converts the horizon times to this unit for use in the script, and converts it back from this unit prior to storage into the database.

Chapter 2

Script Commands

An overview of the format and commands in a script.

Script Basics

The general format for a script is as follows:

- Declarations
- Functions

Comments

You can also have comments in the script. Any line that starts with `//` is ignored as a comment.

Example

```
// this is a comment
```

Special Comments

In the script you may notice one of two special comment lines, they are:

```
//GridScriptMode
```

```
//HorizonScriptMode
```

These comments are used during the loading of a script to set the script editor into the correct mode, if you load scripts that do not have these comments present, then the Grid/Horizon mode of the editor will need to be set manually.

Declarations

The declarations allow you to set-up the required inputs and outputs for the whole script. The commands that can be found in this section are as follows:

SURFACEIN

To use an existing surface (grid or horizon) you need to declare it. The format for the command is

surfacein *DBName* **from** *Attribute* **as** *Alias*;

- *DBName* The name for the surface as it appears in the database. If the name has a dash in it, example **B-46,** then wrap it in quotes, **"B-46"**
- *Attribute* Which attribute of the surface do you wish to use, since currently a surface has both a Time & Amp component, this lets you indicate which part of the surface you wish to use. Choices are:
    - *time*
    - *amp*
    - *intvel* The rest of these actually read from the Amp part of the surface.
    - *avgvel*
    - *depth*
- *Alias* This is a name to be used in the rest of the script to refer to this surface. This can not be a reserved word, and this symbol can not be used as an output variable.

Example

```
surfacein C_38 from time as time1;
surfacein DepthGrid from depth as theDepth;
surfacein "B-46" from amp as my_amp;

...

time = time1 + 20;
```

Neat Feature!

When you use this surface in your script, you would use its *alias*, if you use the alias by itself, then as you process the script on a cell by cell basis, the cell that is currently being processed will be read from. You can also use the *absolute* form to indicate that the value is to be read from an absolute cell, the format for that is:

```
Alias[inline,xline]
```

This would read the value from the cell specified. The values used for inline and xline are offsets from the first inline and first xline, thus the first inline in the surface is 0, the second is 1 and so on.

OUTPUT

This lets you declare what attribute you are going to output. Format is:

output *Attribute;*

- *Attribute* Is the output attribute. Currently you are only allowed to have one time, and one amp output. The amp like outputs of intvel, avgvel & depth are included in the amp attribute. Using this symbol in your script will be the value that will ultimately be in the output surface. Attribute is also the name that you will use within the rest of the program, and it can be used as both an input & output variable. Choices are:
    - *time*
    - *amp*
    - *avgvel*
    - *intvel*
    - *depth*

Example

```
output depth;

...

depth = time1 * theVelocity;
```

Like the surfacein declaration above, you can use this in both the *relative* & *absolute* formats, for example:

```
time[10,20] = 1200.0;
```

GLOBAL

If you want to declare a value that is to be used to accumulate information, and needs to be initialized to a set value, use a global value.

global *Alias* **=** *InitialValue;*

- *Alias* Is the name to be used in the rest of the script, once again it can't be a reserved word.
- *InitialValue* The expression to initialize this variable to, the most likely choice is 0.

Example

```
global sum = 0;

...
```

```
sum = sum + time1;
```

SURFACETEMP

If you need to create a "whole surface" working object, use a surfacetemp. The main difference from a global is that a global is a single value, where as, a surfacetemp is the whole surface. Currently, you may not find much use for this as any variable will work, but when we create "whole surface" functions, you will need to prepare your surface to a surfacetemp variable, then use that temp surface in your surface function. The format is:

surfacetemp *Alias;*

- *Alias* The name to use within the rest of the script.

Example

```
surfacetemp smoothGrid;

...

time = smooth(smoothGrid);
```

Like the surfacein declaration above, you can use this in both the *relative* & *absolute* formats, for example:

```
smoothGrid[10,20] = 1200.0;
```

Smooth is an example of a whole grid function that we could implement at a later date.

SEISMIC

In the horizon mode, you can also declare and use the seismic data. The biggest feature in this script tool is the ability to use multiple seismic versions, as well as the seismic attributes of those versions in the script. The format is as follows:

seismic *Version* **attribute** *Attrib* **as** *Alias;*

- *Version* The version of the seismic to use, choices are as follows:
  - *default* Use the latest version of the seismic.
  - *v0, v1 etc* Use the specified version.
  - *yd1, fy2, ia0, ip0* Use the version with the extension specified. Special note, if you are using the processed attributes, use iseis when specifying the attribute.
- *Attrib* Which of the seismic attributes to use, choices are:
  - *iseis* Use the seismic amplitude (raw data). The returned values will be the seismic amplitude range.

- *iphase* Use the instantaneous phase of the data. The returned values will be in the range of -180 to +180 degrees.
- *iamp* Use the instantaneous amplitude of the data.
- *ifreq* Use the instantaneous frequency of the seismic data. The range of returned values are between 0 & the Limit. If you are using this, there are additional parameters that need to specified, the format for this declaration is:
  - **seismic** *Version* **attribute** *Attrib* **as** *Alias* **params:***Band,Limit,Smooth;*
    - *Band* Is the frequency band for the calculation, usually 1/3 the nyquist.
    - *Limit* Is the maximum frequency, usually the nyquist.
    - *Smooth* Is the frequency smoothing in hertz to apply, usually 0.
- *ipol* This is the apparent polarity, the returned values are –1, 0 or 1.
- *icos* Cosine on the instantaneous phase. Returned values are between –1 & 1.

- *Alias* Is the name that will be used in the rest of the script.

Examples

```
seismic default attribute iseis as data;

seismic v2 attribute iphase as phasedata;

seismic fy3 attribute ifreq as frequency
params:187,250,0;
```

PILOTIN

For the cross-correlation functions, a single pilot seismic trace is required. This declaration specifies the input file to use, as well as its alias for the rest of the script.

pilotin *segyfile* **as** *alias;*

- *segyfile* name of the file without the *.SGY* extension. This file will be found in the ***pilot*** subdirectory of the project directory. If the file does not exist, an error will be produced.

- *Alias* name for this pilot to be used in the rest of the script.

Special Information

The SEGY file that contains the pilot trace is created using a utility within the WinPICS program. This utility outputs the selected trace, and encodes in the trace header for the selected trace, the center of the gate, as well as the gate each side of this center. The gate center is stored in bytes 200 – 203 in microseconds in IEEE floating point format, the gate each side can be found in bytes 204-207 also in microseconds and IEEE float.

This declaration loads in the first trace from the SEGY file, and reads in the window information from the specified header words.

Example

```
pilotin ln2pilot as myPilot;
```

This will open the file **`ln2pilot.sgy`** in the directory project\pilot, and assign it the alias of **`myPilot`** for use in the rest of the script.

Special Keywords

There are special keywords that are of particular interest in the script. They hold special meaning as defined

INLINE

XLINE

Using these two words in your script, and more importantly within the *func* block will let you access the current inline and xline offset numbers that are currently being processed. You could use this to find out which cell had the largest value, and thus access that cell from another surface.

Example

As an example we could find the largest amplitude for a horizon, then to use the time of that cell as a reference for an isochron.

```
output time;
surfacein HorzA from time as MyTime;
surfacein HorzA from amp as MyAmp;
surfacein HorzB from time as MyRef;

global myInLine = -1;
global myXline = -1;
global myLargest = 0;

// Find the largest amp
func
{
      if(MyAmp > myLargest)
      {
            myLargest = MyAmp;
```

```
            myInLine = inline;
            myXline = xline;
      }
}
// Use this information
func
{
      time = myRef - MyTime[myInLine,myXline];
}
```

Surface Functions

After the declarations, you code the actual script. The execution flow is based on the concept of the "**Surface as a Whole**". The main loops operate on the whole surface before moving on to the next function. You must have at least one in the script, but the main feature for this is you can have as many of them as you need.

FUNC{}

All of the statements inside the braces will be run on each cell in the surface. You can include more than 1 func loop in your script, the first can be used to collect information about all of the cells in the surface (a sum perhaps) then in the second loop, use that sum as part of your calculation. The format for this is:

```
func{
      Statement;
      Statement;

      …
}
```

Within the func{} loop you can use a variety of statements, these statements operate on a single cell at a time.

SMOOTH()

Applies a smoother to the input data using 1 of 3 methods. This stand alone function must exist outside a func{} block and uses the following syntax:

ID = smooth(*input,smoother,mode***);**

- **ID** Output surface, if this symbol has not been defined, it will create a new surface variable. It cannot be an input surface.

- *Input* Input surface, this must be a defined surface variable. It can be a surfacein variable, or a surface declared and created in an above function. It can also be an output surface like time or amp created in an earlier function.

- *Smoother* An expression used to provide the size of the smoother to apply.

    - If the mode is a **mean** or **median** smoother, then this is the Length of the smoother to apply to the surface. It must always be an odd number, if you enter an even number; it will be rounded up to the next odd value. The output will be the average or the median of the square of the smoother around the output cell.

    - If the mode is a **gaussian** smooth, then this is the sigma value. Larger values give more smoothing.

- *Mode*

    - **mean** Take the average of all of the input cells.

    - **median** Calculate the median value for all the input cells.

    - **gaussian** Apply a Gaussian smooth to the surface.

DIP()

Computes the local dip for a given cell in a surface. Dip is given by the following function:

$\mathrm{Sqrt}((dt/dy)^2 + (dt/dx)^2)$

In order to obtain a more stable dip, the values for the dips are averaged over a specified number of cells. Because dip values a quite small, dip should be stored in an amp type attribute.

Script syntax is:

ID = dip(*input, window***);**

- **ID** is the symbol for the output surface. This must be a surface variable, and if it is not currently defined, one will be created.

- *Input* The surface to compute the dip map for. This must be a surface variable that has already been defined. If you need to pre-process this surface, use a ***func{}*** loop prior to this function.

- *Window* Smoother to apply to the grid prior to dip calculation. This is a smooth with the mean option.

The output is in time units (ms / microseconds / seconds) per project unit (meter / foot).

AZIMUTH()

This is the direction of the local dip. The direction is looking towards higher values. In the case of depth & time, this will be down dip. Like the ***dip*** calculation above, a smooth function is applied prior to the calculation of the azimuth. Final result is given in degrees with north being 0. Azimuth values are in degrees, so they could be stored in the time field, but we recommend the amp field.

Azimuth is calculated using the following function:

```
atan((dt/dy)/(dt/dx))
```

Script syntax is:

ID = azimuth(*input,window***);**

- **ID** is the symbol for the output surface. This must be a surface variable, and if it is not currently defined, one will be created.

- *Input* The surface to compute the azimuth map for. This must be a surface variable that has already been defined. If you need to pre-process this surface, use a ***func{}*** loop prior to this function.

- *Window* Smoother to apply to the grid prior to azimuth calculation. This is a smooth with the mean option.

Output is in degrees from north.

EDGEDETECT()

Search for, and highlight the edges in a surface. The output from this function will be a surface that has the value 1 where it has detected an edge, and a 255 where no edge exists.

The basic process is as follows:

1. Apply a **gaussian** smooth to the input surface, using the sigma value to describe the smoothing factor.

2. Compute the Delta_X & Delta_Y components of the smoothed surface.

3. Compute the dip from the delta's.

4. Suppress the Non-Maxima.

5. Apply Hysteresis to highlight the edges, using the lower & upper thresholds.

Syntax is:

ID = edgedetect(*input, sigma, tlow, thigh***);**

- **ID** is the symbol for the output surface. This must be a surface variable, and if it is not currently defined, one will be created.

- *input* Surface variable that contains the surface to process. This must be a surface variable that has already been defined. If you need to pre-process this surface, use a ***func{}*** loop prior to this function

- *sigma* The smoothing factor used for the **gaussian** smooth done prior to the process. The higher the value, the more blurred the image.

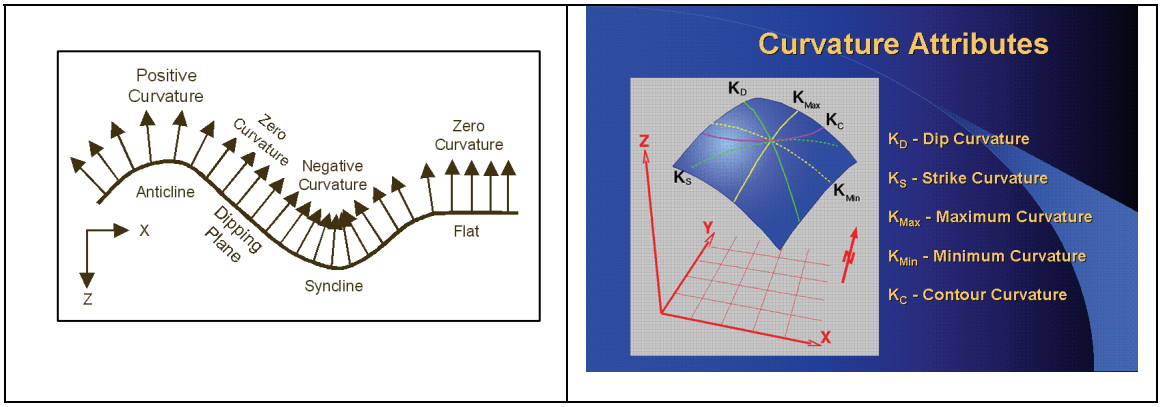
- *tlow* Lower threshold to apply to the Non-Maxima to determine if this is an edge. The value should be between 0 & 1, and less than the ***thigh***.

- *thigh* Upper threshold to apply to the Non-Maxima to determine if this is an edge. The value should be between 0 & 1, and be greater than ***tlow.***

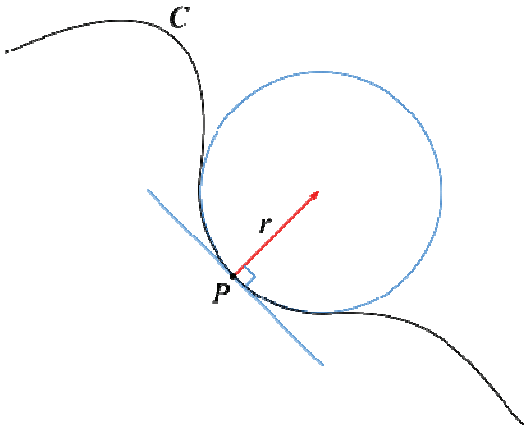
If you wish, you can apply an ***if-then-else*** to the output, and any values that are not an edge, set it to a No-Pick (***setnopick***), by doing this, you can overlay the edge grid on top of the original grid so you can see the relationship, there is an example script in the examples section.

CURVATURE()

Computes the curvature attributes for the input surface. Curvature is the amount a surface is changing dip in a particular direction. For the 2D case curvature is calculated along the line. In the 3D case, curvature may be calculated any direction along the surface. Curvature is defined by fitting an osculating circle to a point P of the surface. The inverse of the radius of this circle is the curvature.



Curvature



Script syntax is:

```
ID = curvature(input, aperture, scale, type);
```

- **ID** is the symbol for the output surface. This must be a surface variable, and if it is not currently defined, one will be created.
- *Input* The surface to compute the curvature map for. This must be a surface variable that has already been defined. If you need to pre-process this surface, use a ***func{}*** loop prior to this function.
- *Aperture* – distance in bins from the centre bin to use for the calculations.
- *Scale* – Multiply the result by this number, think of this as an amplifier.
- *Type*
    - **mean** – average of the minimum and maximum curvature.

- **gaussian** – the product of the minimum and maximum curvatures.
- **maximum** – the maximum curvature in all directions.
- **minimum** – the minimum curvature in all directions.
- **positive** – defines the curvature that has the most positive value.
- **Negative** – defines the curvature that has the most negative value.
- **shape_index** – indicates the local shape of a surface
- **dip** – the curvature in the direction of the dip.
- **strik** – the curvature along the strike plane.
- **contour** – curvature of the contour itself.
- **curvedness** – an overall measure of curvature
- **profile** – positive values indicate convex upward surfaces.
- **plan** – positive values indicate convex outward surfaces.

Mathematical Functions

These are the functions and operators that result in numbers, most are the same as was in the old math routines. All of these functions are to be placed inside a func{} block, you will get a syntax error if this is not the case.

MATH OPERATORS

These are the basic math operators they are:

- Addition, **+**
- Subtraction, **–**
- Multiplication, *****
- Division, **/**
  - Special note about division is if the equation results in a division by 0, the result will be a NO_PICK.
- Assignment **=**

- Allows you to assign the result on the right side of the equation, to the **ID** on the left side of the equation.
- If the **ID** is a single value, then the result for the equation will be stored.
- If the **ID** is a surface, then the result will be stored in the cell that is currently being processed. You can use the absolute format to store a value at a particular cell **ID[inline,xline]**
- You can't assign a value to **surfacein** or **seismic** variables, an error will be returned if you attempt this.

Examples

```
func{
     junk = surface + 100 * surf2;
     mysurface[45,23] = 100.0 * surf2;
}
```

SIN()

Computes the **sin** of the expression, the parameter is in radians, format:

sin(*expression***)**

Example

```
time = sin(27+surface);
```

COS()

Computes the cosine of the expression, the parameter is in radians, format:

cos(*expression***)**

Example

```
time = cos(surface - 20);
```

TAN()

Computes the tangent of the expression, the parameter is in radians, format:

tan(*expression***)**

Example

```
time = tan(surface);
```

SQRT()

Computes the square root of the expression. If the expression is negative, a NO_PICK will be returned. Format for the command is:

sqrt(*expression***)**

Example

```
amp = sqrt(x + y);
```

ABS()

Return the absolute value of the expression. The result will always be a positive number.

abs(*expression***)**

Example

```
time = abs(upper – lower);
```

Script Flow Statements

A new feature is the ability to have different script statements executed depending on a condition being met. These flow statements use the logical constructs define below to determine which set of statements will be executed.

IF(){}

A simple test, if the test results in a true, then the statements inside the braces will be run, otherwise, they will be skipped. The format is:

```
if(logical expression){
      Statements if true;
}
```

Example

```
if(time < 100){
      time = 100;
}
```

IF(){}ELSE{}

An extension of the above test, in this case if the test is true the statements after the if are executed. If the result is false, then the statements following the else will be executed.

```
if(Logical Expression){
      true statements;
}else{
      false statements;
}
```

Example

```
If(time < 100){
      time = 100;
}else{
      time = time + 100;
}
```

Logical Operations

A new feature in PICScript is the addition of logical operations (true/false), this combined with the script flow statements allow for alternate script paths dependant on a condition being met.

LOGICAL COMPARISONS

These are the basic logical operators, the test as you will. All these operators take expressions, and return **true** or **false** (a logical), they are:

- Equal to, **==**
- Greater than, **>**
- Less that, **<**
- Greater than or Equal to, **>=**
- Less than of Equal to, **<=**
- Not equal to, **!=**

Examples

```
Time > 100
x == 20
y <= x
b != 20
```

LOGICAL OPERATORS

These operators let you string logical comparisons together; they all take logical expressions as the parameter, and return a logical expression. They are:

- And, **&&**
    - *Logical* **&&** *Logical*
        - Returns true if **both** logical expressions are true.
- Or, **||** (this is the pipe symbol, generally found above the \)
    - *Logical* **||** *Logical*
        - Returns true if **either** logical expression is true.
- Not, **!**
    - **!***Logical*
        - Inverts the result of the logical, returns true if the logical expression is false, and false if the expression is true.

Examples

- `if(!(27>30))`
    - This will return true
- `if((time < 20) || (time > 50))`

- True if time is not between 20 & 50

- `if (time > 20) && (time < 50))`
  - Reverse of the above, true if time is between 20 & 50

No Pick Functions

Since a cell can be a no pick, these function allow you to test, or set the NO_PICK flag for the cell.

ISNOPICK()

This is a logical test, if the expression has the NO_PICK flag set, this will return a logical true. The format for the command is:

isnopick(*expression***)**

Example

```
if(isnopick(time)){
      time = time + 100;
}
```

SETNOPICK()

There may be times that you want to set or change the flag setnopick gives you that ability. The usage for the command is:

setnopick(*expression, flag***)**

- *Expression,* the expression to set the flag to.
- *Flag,* What to do with the flag, choices are:
  - **on** turns on the flag (makes it a NO_PICK)
  - **off** turns off (clears) the flag, makes the value live.
  - **flip** changes the state of the flag, if on, turn it off, and visa-versa.

Example

```
time = setnopick(time,on);
```

Seismic Functions

These functions give you access to the seismic amplitudes for the seismic data that is declared in the declaration section.

AMPLITUDE()

This function returns the amplitude from the seismic data at the sample nearest the given time. The usage is:

amplitude(*Seismic Alias,Time***)**

- *Seismic Alias,* is the symbol declared in the declaration section, and must point to a seismic variable.

- *Time* An expression that will result in a time, for the amplitude to be returned. The time is in user units. If time is a NO_PICK, the function will return a NO_PICK.

Example

```
amp = amplitude(data,C_38 + 100);
```

Important Note!

The Amplitude function returns the amplitude of the sample **nearest** to the time specified. If you need subsample interpolation use the **snap** function in manual mode.

MINAMP()

Returns the minimum amplitude found over a window. Also optionally limits the test to amplitudes in a range. This function works to the nearest sample of the input data.

minamp(*Seismic Alias,Start,End,[Lower],[Upper]***)**

- *Seismic Alias,* is the symbol declared in the declaration section, and must point to a seismic variable.

- *Start,* An expression that results in a time in user units for the **start** of the window.

- *End,* An expression that results in a time in user units for the **end** of the window.

- *Lower,* An optional expression, if present, the result will put a lower limit on the values to be tested. Both lower & upper must be specified.

- *Upper,* An optional expression, if present, the result will put an upper limit on the values to be tested. Both lower & upper must be specified.

If any of *Start, End, Lower,* or *Upper* results in a NO_PICK, then the function will return a NO_PICK.

MAXAMP()

Returns the maximum amplitude found over a window. Also optionally limits the test to amplitudes in a range. This function works to the nearest sample of the input data.

maxamp(*Seismic Alias,Start,End,[Lower],[Upper]***)**

- *Seismic Alias,* is the symbol declared in the declaration section, and must point to a seismic variable.

- *Start,* An expression that results in a time in user units for the **start** of the window.

- *End,* An expression that results in a time in user units for the **end** of the window.

- *Lower,* An optional expression, if present, the result will put a lower limit on the values to be tested. Both lower & upper must be specified.

- *Upper,* An optional expression, if present, the result will put an upper limit on the values to be tested. Both lower & upper must be specified.

If any of *Start, End, Lower,* or *Upper* results in a NO_PICK, then the function will return a NO_PICK.

MINTIME()

Searches for the minimum amplitude over a window, when found, it will return the **time** of the minimum. Also optionally limits the test to amplitudes in a range. This function works to the nearest sample of the input data.

mintime(*Seismic Alias,Start,End,[Lower],[Upper]***)**

- *Seismic Alias,* is the symbol declared in the declaration section, and must point to a seismic variable.

- *Start,* An expression that results in a time in user units for the **start** of the window.

- *End,* An expression that results in a time in user units for the **end** of the window.

- *Lower,* An optional expression, if present, the result will put a lower limit on the values to be tested. Both lower & upper must be specified.

- *Upper,* An optional expression, if present, the result will put an upper limit on the values to be tested. Both lower & upper must be specified.

If any of *Start, End, Lower,* or *Upper* results in a NO_PICK, then the function will return a NO_PICK.

MAXTIME()

Searches for the maximum amplitude over a window, when found, it will return the **time** of the maximum. Also optionally limits the test to amplitudes in a range. This function works to the nearest sample of the input data.

maxtime(*Seismic Alias,Start,End,[Lower],[Upper]***)**

- *Seismic Alias,* is the symbol declared in the declaration section, and must point to a seismic variable.

- *Start,* An expression that results in a time in user units for the **start** of the window.

- *End,* An expression that results in a time in user units for the **end** of the window.

- *Lower,* An optional expression, if present, the result will put a lower limit on the values to be tested. Both lower & upper must be specified.

- *Upper,* An optional expression, if present, the result will put an upper limit on the values to be tested. Both lower & upper must be specified.

If any of *Start, End, Lower,* or *Upper* results in a NO_PICK, then the function will return a NO_PICK.

GETTIME()

Searches for the value indicated, and returns the time that value was found within the window specified, if the value is not in the window, a NO_PICK is returned. If there are multiple occurrences of the value, the one closest the center of the gate will be used. The time is returned to the nearest $1/8^{th}$ of a sample.

gettime(*Seismic Alias,value,Start,End***)**

- *Seismic Alias,* is the symbol declared in the declaration section, and must point to a seismic variable.

- *Value,* An expression that results in the value to look for.

- *Start,* An expression that results in a time in user units for the **start** of the window.

- *End,* An expression that results in a time in user units for the **end** of the window.

If any of *Value, Start,* or *End,* results in a NO_PICK, then the function will return a NO_PICK.

MEAN()

ABSMEAN()

Returns the average (mean) of all the samples over a window, using either the original or absolute values. Also optionally limits the values included to amplitudes in a range. Mean is defined as the sum of all the samples, divided by the count of the samples.

mean(*Seismic Alias,Start,End,[Lower],[Upper]***)**

absmean(*Seismic Alias,Start,End,[Lower],[Upper]***)**

- *Seismic Alias,* is the symbol declared in the declaration section, and must point to a seismic variable.

- *Start,* An expression that results in a time in user units for the **start** of the window.

- *End,* An expression that results in a time in user units for the **end** of the window.

- *Lower,* An optional expression, if present, the result will put a lower limit on the values to be tested. Both lower & upper must be specified.

- *Upper,* An optional expression, if present, the result will put an upper limit on the values to be tested. Both lower & upper must be specified.

If any of *Start, End, Lower,* or *Upper* results in a NO_PICK, then the function will return a NO_PICK.

MEDIAN()

Returns the median of all the samples over a window. Also optionally limits the values included to amplitudes in a range. Median is defined as taking all the values in a window, sorting the items in ascending order, then picking the middle item in the sorted list. The result is similar to mean (average) however it is not as affected by extreme values.

median(*Seismic Alias,Start,End,[Lower],[Upper]***)**

- *Seismic Alias,* is the symbol declared in the declaration section, and must point to a seismic variable.

- *Start,* An expression that results in a time in user units for the **start** of the window.

- *End,* An expression that results in a time in user units for the **end** of the window.

- *Lower,* An optional expression, if present, the result will put a lower limit on the values to be tested. Both lower & upper must be specified.

- *Upper,* An optional expression, if present, the result will put an upper limit on the values to be tested. Both lower & upper must be specified.

If any of *Start, End, Lower,* or *Upper* results in a NO_PICK, then the function will return a NO_PICK

RMS()

Returns the RMS (root mean square) of all the samples over a window. Also optionally limits the values included to amplitudes in a range. RMS is defined as the square root of the sum of the samples squared.

rms(*Seismic Alias,Start,End,[Lower],[Upper]***)**

- *Seismic Alias,* is the symbol declared in the declaration section, and must point to a seismic variable.
- *Start,* An expression that results in a time in user units for the **start** of the window.
- *End,* An expression that results in a time in user units for the **end** of the window.
- *Lower,* An optional expression, if present, the result will put a lower limit on the values to be tested. Both lower & upper must be specified.
- *Upper,* An optional expression, if present, the result will put an upper limit on the values to be tested. Both lower & upper must be specified.

If any of *Start, End, Lower,* or *Upper* results in a NO_PICK, then the function will return a NO_PICK.

SUM()

Returns the SUM of all the samples over a window. Also optionally limits the values included to amplitudes in a range.

sum(*Seismic Alias,Start,End,[Lower],[Upper]***)**

- *Seismic Alias,* is the symbol declared in the declaration section, and must point to a seismic variable.
- *Start,* An expression that results in a time in user units for the **start** of the window.
- *End,* An expression that results in a time in user units for the **end** of the window.
- *Lower,* An optional expression, if present, the result will put a lower limit on the values to be tested. Both lower & upper must be specified.
- *Upper,* An optional expression, if present, the result will put an upper limit on the values to be tested. Both lower & upper must be specified.

If any of *Start, End, Lower,* or *Upper* results in a NO_PICK, then the function will return a NO_PICK.

COUNT()

Counts the number of peaks or troughs within a window, & optionally will only count the peak/trough provided it is within a range of values.

count(*Seismic Alias,Start,End,Mode,[Lower],[Upper]***)**

- *Seismic Alias,* is the symbol declared in the declaration section, and must point to a seismic variable.
- *Start,* An expression that results in a time in user units for the **start** of the window.
- *End,* An expression that results in a time in user units for the **end** of the window.
- *Mode* What are we looking for?
    - **peak** The function will count the peaks.
    - **trough** The function will count the troughs.
- *Lower,* An optional expression, if present, the result will put a lower limit on the values to be tested. Both lower & upper must be specified.
- *Upper,* An optional expression, if present, the result will put an upper limit on the values to be tested. Both lower & upper must be specified.

If any of *Start, End, Lower,* or *Upper* results in a NO_PICK, then the function will return a NO_PICK.

STD()

Returns the standard deviation of all the samples over a window. Also optionally limits the values included to amplitudes in a range.

std(*Seismic Alias,Start,End,[Lower],[Upper]***)**

- *Seismic Alias,* is the symbol declared in the declaration section, and must point to a seismic variable.
- *Start,* An expression that results in a time in user units for the **start** of the window.
- *End,* An expression that results in a time in user units for the **end** of the window.
- *Lower,* An optional expression, if present, the result will put a lower limit on the values to be tested. Both lower & upper must be specified.
- *Upper,* An optional expression, if present, the result will put an upper limit on the values to be tested. Both lower & upper must be specified.

If any of *Start, End, Lower,* or *Upper* results in a NO_PICK, then the function will return a NO_PICK.

SNAP()

This function will apply the WinPICS horizon picking algorithms to find the requested event about a given seed time. Unlike the Min/Max functions above, this applies a quadratic function to the input samples, and will provide the time or amplitude to the nearest 1/8 sample.

snap(*Seismic Alias*,*seed*,*event*,*TimeOrAmp***)**

- *Seismic Alias* is the symbol declared in the declaration section, and must point to a seismic variable.
- *Seed* Time to be used as a seed for the picker, this is the same as the "rubber band" that is used during horizon picking within WinPICS.
- *Event* A mnemonic describing what is to be picked, the valid values are:
  - **peak** A peak.
  - **trough** A trough.
  - **zcmp** A zero-crossing going from a trough to a peak.
  - **zcpm** A zero-crossing going from a peak to a trough.
  - **manual** Fixed Time, used for subsample amplitude extraction.
- *TimeOrAmp* What is to be returned from the function, the valid values are:
  - **time** Return the time to the nearest 1/8 of a sample.
  - **amp** Return the amplitude for the above pick.

Example

```
MyNewHorz = snap(mySeismic,myHorz+10,peak,time);
```

This example will take the seismic attached to the alias **mySeismic** and pick a peak that is 10 milliseconds above **myHorz** and return the time for that pick.

Notes

There is very little use for the time output of the snap function in **manual** mode, it will just return the input seed time. The real power is when **manual** is used to return an amplitude, in this case the quadratic fit is applied at the time specified; this will result in an accurate representation of the amplitude at the time (even sub sample intervals).

```
MyNewHorz = snap(mySeismic,myHorz+5.25,manual,amp);
```

AREA()

Using a window, this will compute the area under the peaks or troughs from an optional baseline

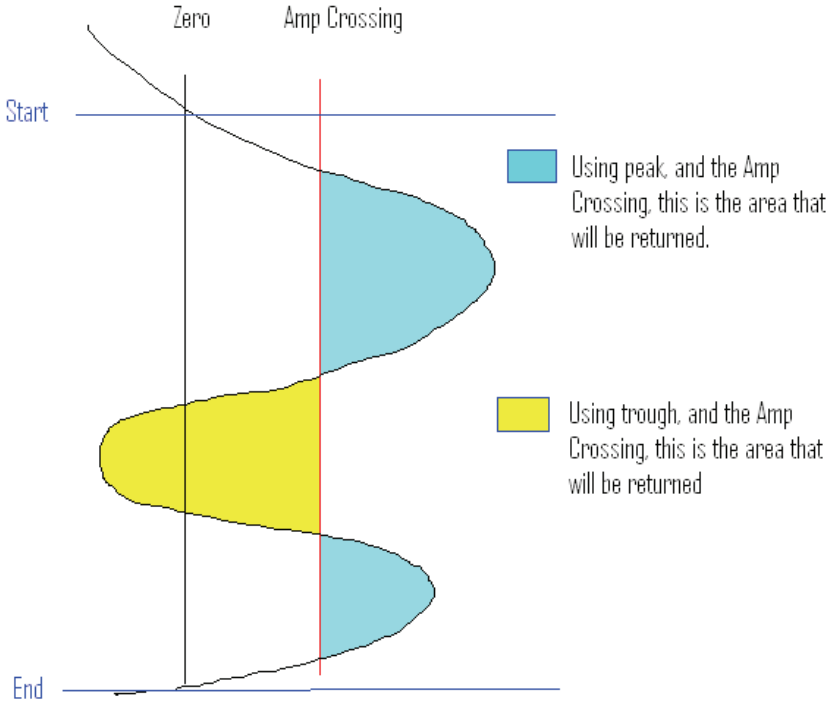
The value that is returned is in Amp by User Time Units. So if the user has selected Milliseconds as the time units, this is in Amp Milliseconds.

area(*Seismic, Start, End, Mode, [Amp Crossing]*)

- *Seismic* is the symbol declared in the declaration section, and must point to a seismic variable.
- *Start,* An expression that results in a time in user units for the **start** of the window.
- *End,* An expression that results in a time in user units for the **end** of the window.
- *Mode* defines the type of wavelet we are looking for
    - **peak** The function will calculate the area of the peak (+), if there are no amplitudes above the crossing value, then a no-pick is returned.
    - **trough** The function will calculate the area under the trough (-) if there are no amplitudes below the crossing value, then a no-pick is returned.
- *Amp Crossing* optional, allows the user to supply an amp value to use as the boundary, area is calculated between this and the rest of the wavelet. If no value is supplied, then the default is 0.

notes

The following diagrams should help to define the areas that will be returned.



CORGATE()

Cross-Correlate the specified pilot with the seismic trace centred about a specified time, and including a window above and below this zero-time, as well as a search window. Return the best cross-correlation coefficient, or the time for the best correlation over the specified search window.

corgate(*pilot,seismic,centre,gate,search,return***)**

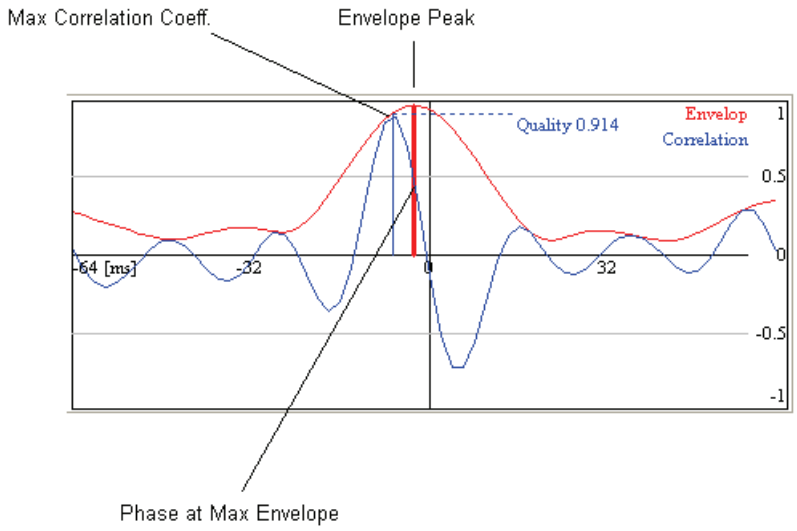
- *Pilot* is the symbol declared in the declaration section, and must point to a pilot variable.
- *Seismic* is the symbol declared in the declaration section, and must point to a seismic variable.
- *Centre* An expression that resolves to the centre time for the correlation, this is also the time zero as well.
- *Gate* An expression that resolves to the size of the gate to be included each side of the centre. So the size of the seismic that will be correlated with the pilot is centre – gate to centre + gate.
- *Search* An expression that resolves to the amount of time to search for the value requested, a value of 0 means no search.

- *return* What is to be returned from the function, the valid values are:
    - **xcortime** Return the time that the maximum cross correlation value is found. This is the time measured from time zero for the seismic data set.
    - **xcor** Return the max cross-correlation coefficient.
    - **envtime** Return the time for the maximum of the cross-correlation envelope. Simply put, this is the time at which you would get the best tie if the seismic trace were phase adjusted to the pilot. This is the time measured from time zero for the seismic data set.
    - **envamp** Return the maximum value for the cross-correlation coefficient envelope. This is a measure of how good the tie could be if the seismic trace were phase rotated to match the pilot.
    - **phase** Return the phase of the cross-correlation at the maximum of the envelope. This gives you a measure of how much the phase is different from the pilot. If you were to phase adjust the data by this value, you would in effect line up the cross-correlation maximum and the envelope maximum.
    - **rawxcor** The maximum of the cross-correlation. This is similar to **xcor** but the value is not normalized to 1 (the coefficient).
    - **rawenvamp** The maximum of the envelope, same as **envamp** but the value is not normalized to 1.
    - **xcorabs** Returns the absolute maximum of the cross-correlation coefficient.
    - **xcorabstime** Returns the time at which the absolute maximum occurs.
    - **rawxcorabs** Similar to **xcorabs** but the values are not normalized to 1.

Notes

This function will pad the windows of the pilot or seismic with zeros so that they are the same size.

In order to better understand what is returned from this (and the corwnd function) take a look at the following diagram taken from WinPICS tie diagnostic:



WinPICS tie diagnostic.

Using the **xcortime** parameter, you will get the time of the maximum cross-correlation coefficient. This will be the actual time of this, so if time zero for the input is 1100 ms, and the peak is –10 ms from time zero, then the time will be 1090 ms.

If you are requesting **xcor**, you will get the quality value. For **envamp**, you will get the value for the peak of the envelope.

For **envtime** you will get the time of the envelope peak, and it will be applied in the same manner as the **xcortime** calculation above.

For the **phase** calculation, you will get the instantaneous phase of the cross-correlation at the time of the envelope peak.

It is important to note that the search will limit how far from zero these functions will search for the max values. So if the max is outside this range, you may not get what you expect.

Example

```
amp=corgate(myPilot,mySeis,myHorz,40,10,correlate);
```

This example will cross-correlate **myPilot** with **mySeis** taking 80 ms centred about **myHorz**, and looking +/- 10 ms for the best cross-correlation coefficient, and returning that value to be stored in the amp field.

CORWND()

Cross-Correlate the specified pilot with the seismic trace between the start and end times, as well as a search window. Return the best cross-correlation coefficient, or the time for the best correlation over the specified search window.

corwnd(*pilot,seismic,start,end,search,return***)**

- *Pilot* is the symbol declared in the declaration section, and must point to a pilot variable.

- *Seismic* is the symbol declared in the declaration section, and must point to a seismic variable.

- *start* An expression that resolves to the start time for the correlation window.

- *end* An expression that resolves to the end of the seismic window for the correlation.

- *Search* An expression that resolves to the amount of time to search for a maximum correlation, a value of 0 means no search.

- *return* What is to be returned from the function, the valid values are:

    - **xcortime** Return the time that the maximum cross correlation value is found. This is the time measured from time zero for the seismic data set.

    - **xcor** Return the max cross-correlation coefficient.

    - **envtime** Return the time for the maximum of the cross-correlation envelope. Simply put, this is the time at which you would get the best tie if the seismic trace were phase adjusted to the pilot. This is the time measured from time zero for the seismic data set.

    - **envamp** Return the maximum value for the cross-correlation coefficient envelope. This is a measure of how good the tie could be if the seismic trace were phase rotated to match the pilot.

    - **phase** Return the phase of the cross-correlation at the maximum of the envelope. This gives you a measure of how much the phase is different from the pilot. If you were to phase adjust the data by this value, you would in effect line up the cross-correlation maximum and the envelope maximum.

    - **rawxcor** The maximum of the cross-correlation. This is similar to **xcor** but the value is not normalized to 1 (the coefficient).

    - **rawenvamp** The maximum of the envelope, same as **envamp** but the value is not normalized to 1.

- **xcorabs** Returns the absolute maximum of the cross-correlation coefficient.
- **xcorabstime** Returns the time at which the absolute maximum occurs.
- **rawxcorabs** Similar to **xcorabs** but the values are not normalized to 1.

Notes

This function will pad the windows of the pilot or seismic with zeros so that they are the same size. Also, the time zero is taken as the middle of the window.

See **corgate** for an in-depth description of the return values.

Example

```
amp = corwnd(myPilot,mySeis,horza,horzb,10,amp);
```

This example will cross-correlate **myPilot** with **mySeis** taking the window from **horza** to **horzb**, and looking +/- 10 ms for the best cross-correlation coefficient, and returning that value to be stored in the amp field.

MANHATTANGATE ( )

Manhattan Distance is another measure of similarity of a seismic trace to some model trace. Manhattan Distance uses 2 equal length wavelets, with N time samples, it will sum the absolute differences for each sample along the gate specified, and therefore the formula is:

$$M = \sum_{i=1}^{N} | P_i - T_i |$$

Where:

- M is the Manhattan Distance
- P is the pilot (model) wavelet
- T is the target wavelet
- N is the number of time samples

One can see that the result of two identical traces is 0, and wavelets that are not identical are positive values. There is also a normalized version using the formula:

$$M = \frac{\sum_{i=1}^{N} |P_i - T_i|}{\sum_{i=1}^{N} |P_i| + |T_i|}$$

This will result in identical data being 0, and the most discontinuous data being a 1.

manhattangate(*Pilot,Seismic,Centre,Gate,Search,Ret***)**

- *Pilot* is the symbol declared in the declaration section, and must point to a pilot variable. The number of samples pulled from the pilot will always match the number of samples from the gate.
- *Seismic* is the symbol declared in the declaration section, and must point to a seismic variable.
- *Centre* An expression that resolves to the centre time for the window, this is also the time zero.
- *Gate* An expression that resolves to the size of the gate to be included each side of the centre. So the size of the seismic that will be processed with the pilot is centre – gate to centre + gate.
- *Search* An expression that resolves to the time to search up and down to find the best match, 0 will not search. Please note that increasing the time will require more time to process.
- *Ret* What is to be returned from the function, the valid values are:
    - ***raw*** Just return the raw Manhattan Distance.
    - ***normalized*** Return the Normalized Distance.

MANHATTANWND ()

This is the same as the above **manhattangate** function with the exception that the times define the start and end of the gate to use. In this case, the centre of the gate is used as a match for the centre of the pilot trace.

manhattanwnd(*Pilot,Seismic,Start,End,Search,Ret***)**

- *Pilot* is the symbol declared in the declaration section, and must point to a pilot variable. The pilot will always have the window to match the window from the seismic.

- *Seismic* is the symbol declared in the declaration section, and must point to a seismic variable.
- *Start* An expression that resolves to the start time for the window.
- *End* An expression that resolves to the end time for the window.
- *Search* An expression that resolves to the time to search up and down to find the best match, 0 will not search a value. Please note that increasing the time will require more time to process.
- *Ret* What is to be returned from the function, the valid values are:
    - ***raw*** Just return the raw Manhattan Distance.
    - ***normalized*** Return the Normalized Distance.

Chapter 3

QuickScript

PicScript is just a bit too much, how about a simple interface to reuse stock scripts?

What is QuickScript

QuickScript is a tool that will allow the user to open existing PicScript files, and select input grid/horizon information as well as parameters for the functions in the script.

Chapter 4

Sample Scripts

The best way to learn to use scripts is to look at some examples.

Compute an isochron

For Starters

This first example shows a simple use to compute an isochron between 2 horizons. This shows the minimum to create a 1 line math function.

```
// Basic isochron script.

// Declare input horizons
surfacein B_46 from time as b46time;
surfacein test from time as testtime;

// And what to output
output time;

// And do it.
func{
   time = b46time - testtime;
}
```

Changing the geometry of a grid

```
// A simple Grid Resample Script

// Changes the geometry of the input grid test_T1
// to the geometry as specified on the
// Geometry/Reference tab.

// Declare the input grid.
surfacein test_T1 from time as test;

// Only outputting time
output time;

// For every cell in the output geometry, get the time
// of the input grid. If the cell of the output does
// not fall over a cell of the input, it will be set to
// a NO_PICK.
func{
```

```
   time = test;
}
```

Using If statements

This example shows a simple if then else use to clip a horizon to a min / max range.

```
// Limit the range of a horizon

// Declare the input
surfacein C_38 from time as HorizonTime;

// Declare the output
output time;

// For each cell
func{

   // Start with the time
   time = HorizonTime;

   if( HorizonTime > 1360 ){
      // Set the time to the maximum
      time = 1360;
   }

   if( HorizonTime < 1340 ){
      // Set the time to the minimum
      time = 1340;
   }
}
```

Clip a grid to an average range

No Limits!

This example shows the use of **if(){true}else{false}** constructs. As well it shows testing and setting the NO-PICK flag.

```
// Blank out picks that are more than 50 meters
// away from the average depth of the surface.

// Define our input horizon
surfacein C_38_DW1 from depth as GridDepth;

// Create a couple of globals to compute the
// average depth with, initialize to 0
global count = 0;
global total = 0;

// Define the output attribute
output depth;

// First pass to collect information,
// and compute average from live data.
```

```
func{
   if(!isnopick(GridDepth)){
      total = total + GridDepth;
      count = count + 1;
      average = total / count;
   }
}

// Second pass to use that information
func{
   if((GridDepth < (average - 50)) || ((average + 50) <
GridDepth)){
      // The depth is outside 50 meters of the average
      // Set it to a no_pick
      depth = setnopick(GridDepth,on);
   }else{
      // Depth is how far the depth is from the average
      depth = GridDepth - average;
   }
}
```

The min / max functions

These functions are now able to return the min / max amplitude, as well as return the time at which that min / max occurs. The most obvious use for these would be to re-pick the data after phase rotations, or to pick a peak or trough near an existing pick. In this example, we are looking for the trough above the C_38 event. That trough is within 20 ms of the event, so we can limit our search.

```
// Re-Pick the trough within 20 ms above the C_38
// horizon. As well as get the amplitude of that trough

// Define our reference input horizon
surfacein C_38 from time as C_38;

// Which seismic data to use, in this
// case, use the default (latest)
seismic default attribute iseis as Data;

// And what to output (both time & amp)
output amp;
output time;

func{
   // Find the minimum amplitude above the C_38
   // Give me the time of that minimum
   time = mintime(Data , C_38 - 20, C_38);

   // Set the amplitude component to the
   // amplitude at this time. Note that you can
   // use time as an input at this point.
   amp = amplitude(Data, time);
}
```

Surface Smoothing with smooth()

If the horizon or grid is a little rough, you can apply a mean smooth to the data set. This gives an example of a using the repick function above, then appling a 3 by 3 cell smoother to the resultant surface, just to clean it up a bit. Note how a func{} block is used to pre-process the surface, then the smoother is applied at the end!

```
// Re-Pick the trough within 20 ms above the C_38
// horizon. As well as get the amplitude of that trough

// Define our reference input horizon
surfacein C_38 from time as C_38;

// Which seismic data to use, in this
// case, use the default (latest)
seismic default attribute iseis as Data;

// And what to output (both time & amp)
output amp;
output time;

func{
   // Find the minimum amplitude above the C_38
   // Give me the time of that minimum
   time = mintime(Data , C_38 - 20, C_38);

   // Set the amplitude component to the
   // amplitude at this time. Note that you can
   // use time as an input at this point.
   amp = amplitude(Data, time);
}

// Now apply a 3X3 smoother to the time data to
// clean up the jitter

time = smooth(time,3,mean);
```

If the dataset contains spikes, and applying a mean smoother to it would result in an undesirable solution, you could use the median smoother which would toss out the wild values. Replace the smooth function with the following:

```
time = smooth(time,3,median);
```

Dip / Azimuth Calculations

These 2 whole surface functions give you a picture of the lay of the surface so to speak. Like the smoother example above, you can prepare a surface in a func{} block, then calculate the dip & azimuth on the result. As an example, we want to calculate an isochron, and then calculate the dip of this isochron.

```
// Declare the 2 input surfaces
surfacein C_38 from time as top;
surfacein B_46 from time as bottom;

// Declare a temporary surface to hold
// the isochron
surfacetemp isochron;

// Declare the output
output amp;

// Create the isochron
func{
   isochron = top - bottom;
}

// Compute the dip
amp = dip(isochron,2);
```

In the next example, we can output the isochron in the time field, smooth it, and compute the azimuth of the isochron:

```
// Declare the 2 input surfaces
surfacein C_38 from time as top;
surfacein B_46 from time as bottom;

// Declare the output surfaces
output time;
output amp;

// Create the isochron
func{
   time = top - bottom;
}

// smooth the isochron
time = smooth(time,5,mean);

// Compute the azimuth
amp = azimuth(time,2);
```

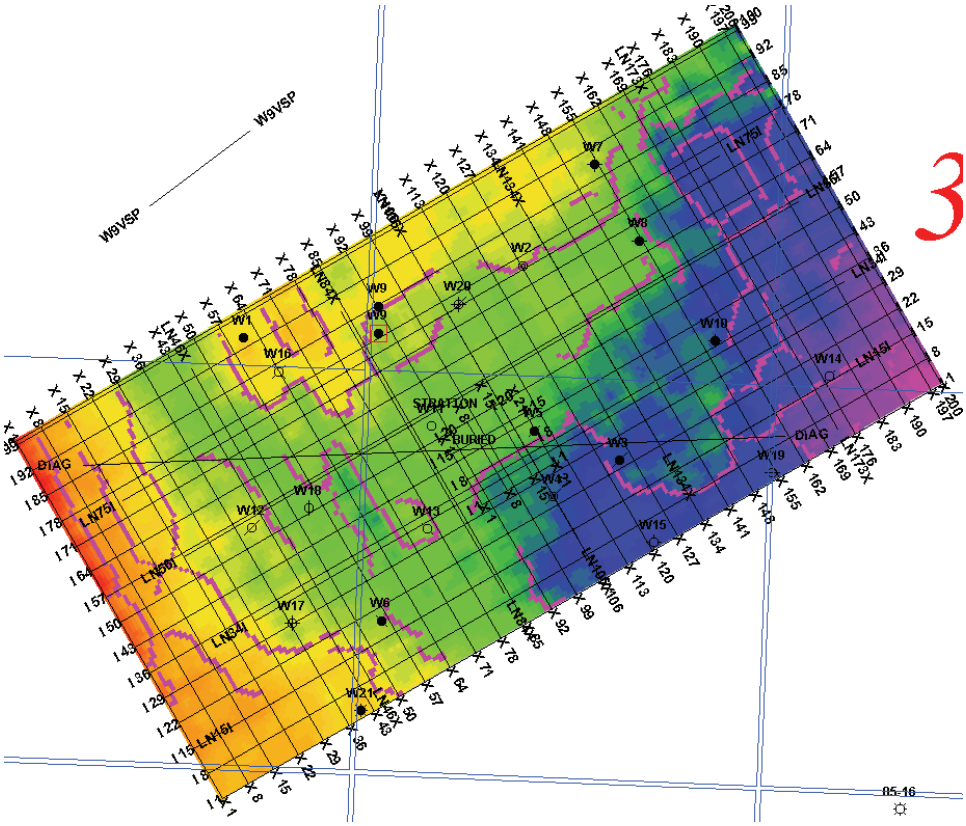
Edge Detection

Well, now that you have a surface, why not try to highlight the edges. Edge detection looks for areas of high dip, and marks the change with a value of 1, the rest of the surface will have a 255. In this example, we will output a new surface which will be a copy of the input surface, with the edges set to the maximum of the surface. By setting the edges to the maximum, when you display this new grid, you should see the original grid with the edges as a contrasting color.

```
//GridScriptMode

// Declare the Input, change for your case
```

```
surfacein test_T1 from time as InputGrid;


// And what you are outputting
output time;

// And a temporary surface to store the edge grid
surfacetemp edges ;

// And a variable to hold the maximum
global MaxGrid = 0;

// Step 1, find the maximum time
func{
   if(InputGrid > MaxGrid){
      MaxGrid = InputGrid;
   }
}

// Find the edges
edges = edgedetect(InputGrid, 2, .5 , .7);

// Superimpose the edges on the original surface
// Output in the time grid
func{
   if(edges == 1){
      // 1 Means this is an edge, set to Max
      time = MaxGrid;
   }else{
      // Set the value to that of the
      // original input surface!
      time = InputGrid;
   }
}
```

Below is an example of an output from this script.



Example of output from Edge Detection