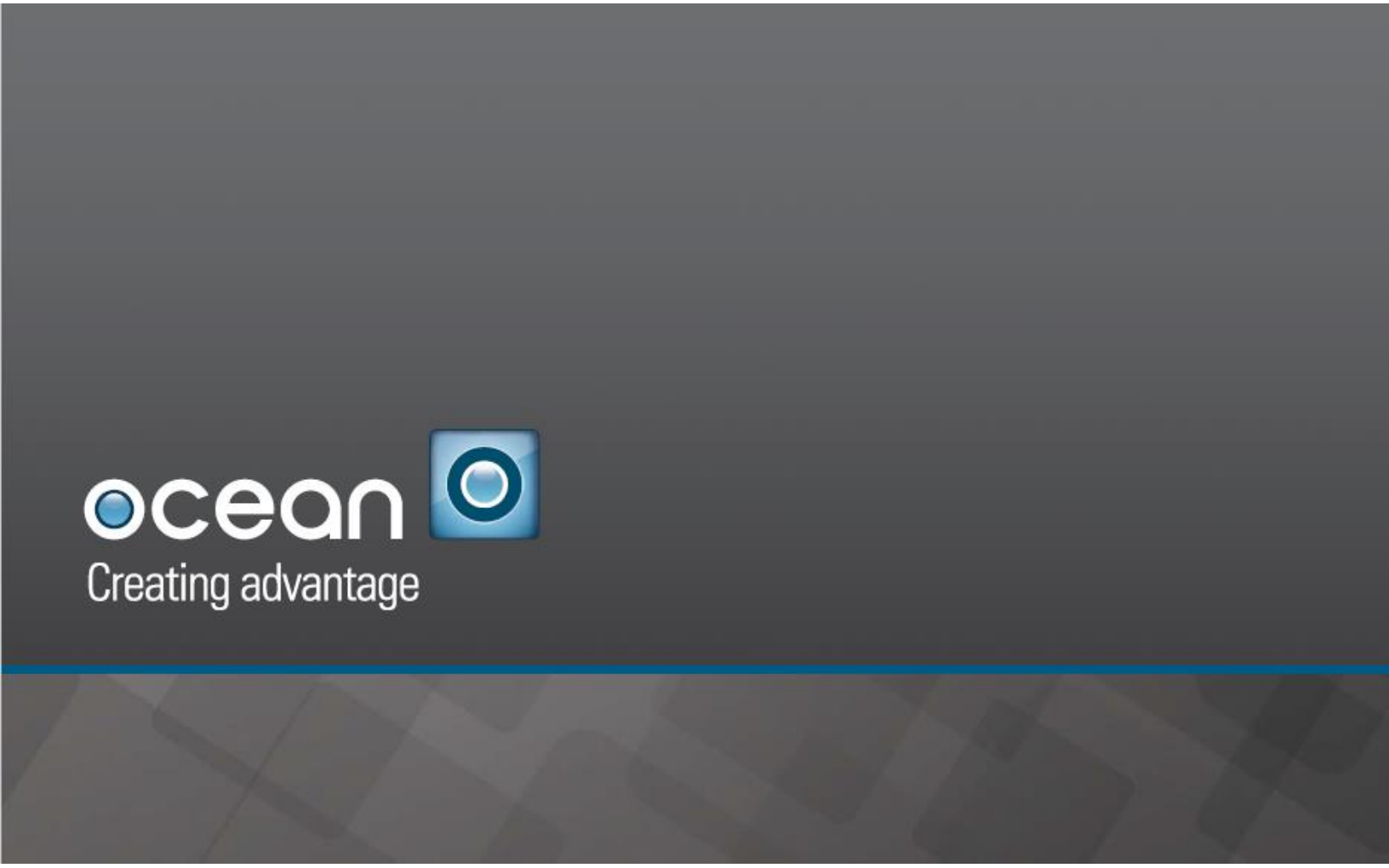
Geology
Volume 4



Ocean Software Development Framework for Techlog
Version 2023

Contents

1 Geology plots

In This Chapter

Introduction

In Techlog all the common plots are presented in the **Plot** menu tab. The domain-oriented plots are in their respective domain menu tabs. For example, the plots related to geology are in the **Geology** tab. The **Geology** tab is divided into several groups and each group presents the most common plots on the left side. More geology plot options are available in the drop-down menu for each geology plot group.

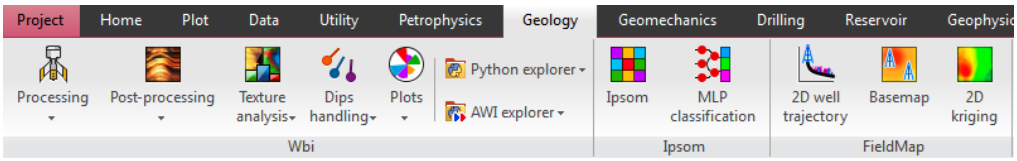


Figure 1-1 Geology plots availables in "Geology" tab of Techlog

Geology plots exposed with Ocean are:

- 2D well trajectory plot
- Stereonet plot

2D well trajectory plot

The 2D well trajectory allows you to view the well trajectory in a cross section or a top view, with curves or dips along the well. To display the well trajectory in 2D, Techlog needs an **Index** dataset with all the data to build the spatial geometry of the well. Create the Index dataset for the well using the **Create Index set** tool available in **Data** > **Techdata** > **Reference and coordinates** > **Reference dataset** > **Create Index set**.

The reference index dataset tool creates mappings between all the references of a well. The Index dataset must contain the following references to display the well trajectory in 2D well trajectory plot:

- Measured Depth
- True Vertical Depth
- X and Y offset

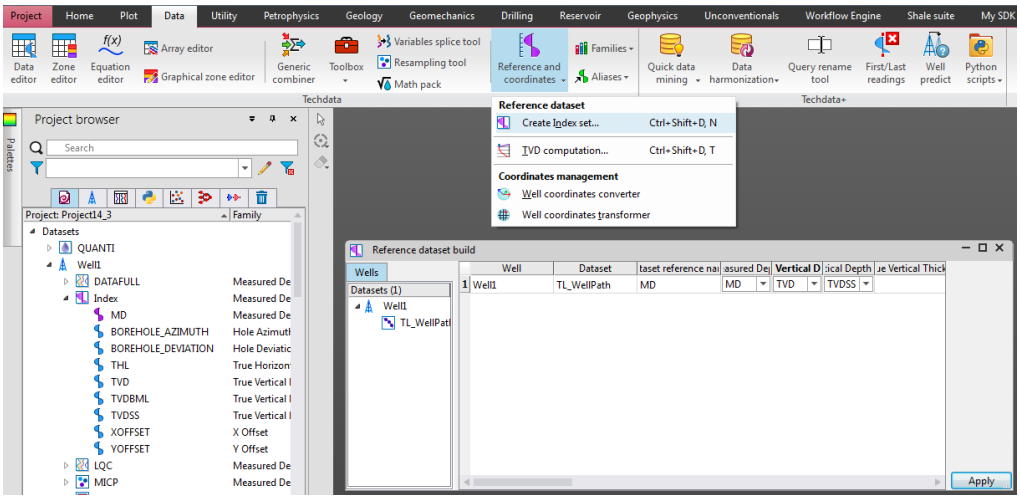


Figure 1-2 Create an index dataset

The 2D well trajectory plot is found in **Geology** > **FieldMap** > **2D well trajectory**.

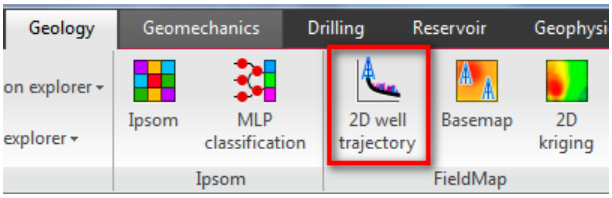


Figure 1-3 2D well trajectory plot in Techlog

The **2D well trajectory** window is composed of three primary areas:

1. Well side box: to drag the variables to be displayed.
2. Line of section toolbar: manages the azimuth of the projection axis for the cross-section view.
3. Display area: where the well trajectory is displayed. You can display only one well at a given time, and you can display up to one dip variable and two other variables with continuous or point data types.

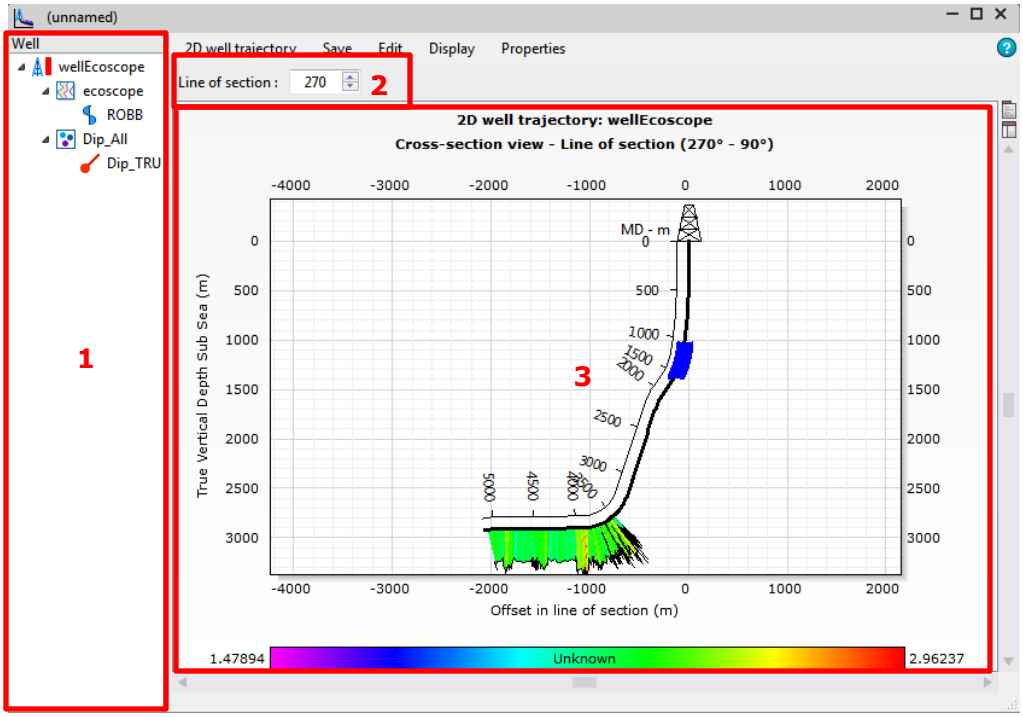


Figure 1-4 2D well trajectory window

All those areas are accessible programmatically through Ocean.

Ocean also allows you to represent properties (lithology) along the borehole trajectory through base polygons and to add geological boundaries inside these base polygons through geometry objects.

To implement a 2D well trajectory plot with Ocean:

1. Create the 2D well trajectory plot and add logs.
2. Add base polygons.
3. Set polygon properties.
4. Add boundaries to the geometry.
    - Update polygons properties result

Well2DTrajectoryPlot domain object

Step 1 - Create the 2D well trajectory plot and add logs

The **`Well2DTrajectoryPlot`** domain object represents the 2D well trajectory plot. This domain object is used to create a 2D well trajectory plot and add logs to it.

The **`Well2DTrajectoryPlot`** class inherits from the **`Plot`** class and does not support a name. A **`Well2DTrajectoryPlot`** object is instantiated using the static **`create`** method, passing the parent **`Workspace`** object as an argument. The uniqueness of a **`Well2DTrajectoryPlot`** object is managed by the system. The only way to retrieve a **`Well2DTrajectoryPlot`** object from the workspace is by its droid, typically stored as a private member variable of the plug-in.

You can also add a **`Well2DTrajectoryPlot`** to a container plot (matrix-plot) using the dedicated **`create`** static methods.

See the "ContainerPlot domain object" section in the *Ocean for Techlog Developer Guide - Plots* for more information on how to create a 2D well trajectory plot in a container plot.

```
class Well2DTrajectoryPlot : public Plot
{
public:
  static Well2DTrajectoryPlot create(Workspace workspace);
  static Well2DTrajectoryPlot create(ContainerPlot
    containerPlot, unsigned int row, unsigned int column);
  static Well2DTrajectoryPlot create(ContainerPlot
    containerPlot, unsigned int row, unsigned int column,
    unsigned int rowSpan, unsigned int columnSpan);

  void setDipVariable(const Variable &dipVariable);
  void unsetDipVariable();
  const Variable findDipVariable() const;
  void setVariable1(const Variable &variable1);
  void unsetVariable1();
  const Variable findVariable1() const;
  void setVariable2(const Variable &variable2);
  void unsetVariable2();
  const Variable findVariable2() const;

  Well2DTrajectoryDisplayType displayType() const ;
  void setDisplayType(const Well2DTrajectoryDisplayType
  displayType);
  ...
};
```

Set the well logs on the plot using the following public functions:

- **`setDipVariable`**, **`unsetDipVariable`** and **`findDipVariable`** allow you to respectively add, remove and retrieve a **`VariableTypeDip`** variable that has a variable unit compatible with degree angle
- **`setVariable1`**, **`unsetVariable1`** and **`findVariable1`** allow you to respectively add, remove and retrieve a **`VariableTypePointData`** or **`VariableTypeContinuous`** variable and the well log curve is added to the "variable1" position of the **`Well2DTrajectoryPlot`**.
- **`setVariable2`**, **`unsetVariable2`** and **`findVariable2`** allow you to respectively to add, remove and retrieve a **`VariableTypePointData`** or **`VariableTypeContinuous`** variable and the well log curve is added to the "variable2" position of the **`Well2DTrajectoryPlot`**.

Note: The well trajectory is built when the variable is added to the 2D well trajectory plot. If the variable belongs to another well, a new trajectory is built.

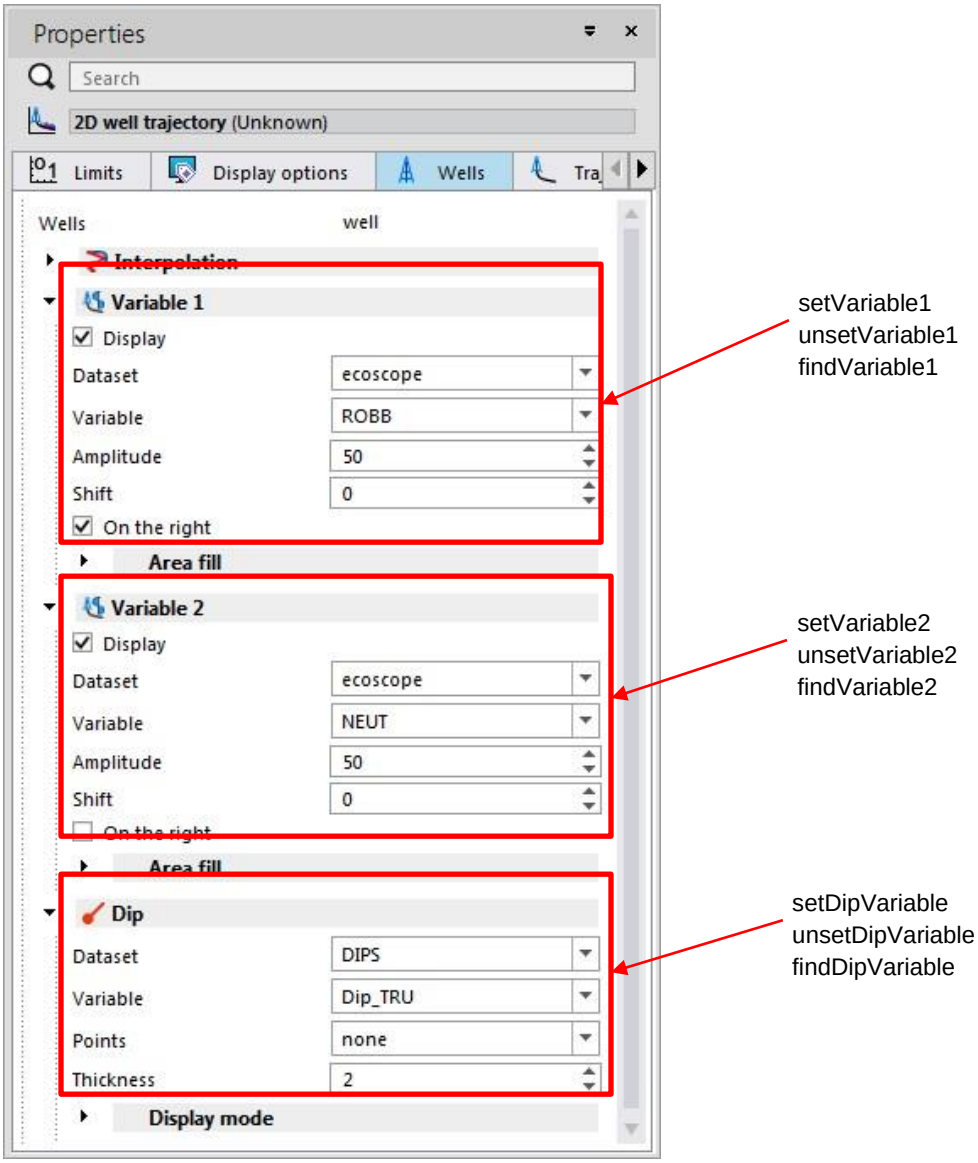


Figure 1-5 2D well trajectory properties showing variable assignements

Use the **`displayType`** function to get the current view of the well trajectory in the plot. The function returns an enum value of **`Well2DTrajectoryDisplayType`** enum class. To change the view, use **`setDiplayType`** function.

```
enum Well2DTrajectoryDisplayType
{
  Well2DTrajectoryDisplayType_TopView,
  Well2DTrajectoryDisplayType_CrossSectionView,
  Well2DTrajectoryDisplayType_CurtainSectionView
};
```

The available views are:

- Cross-section view: the trajectory corresponds to the well trajectory offset from the well head, projected on the line of section in red.

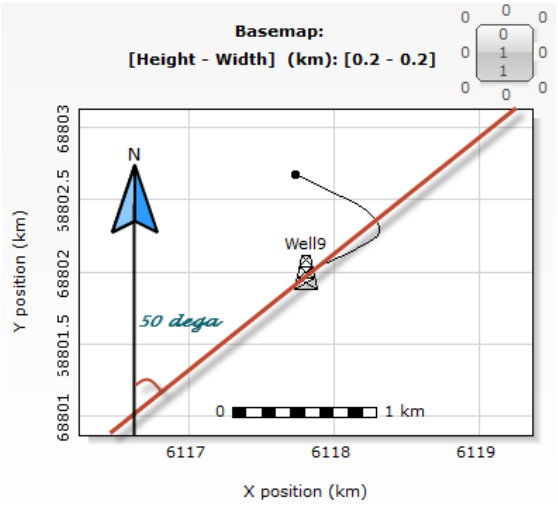


Figure 1-6 Cross-section view

- Top view: the top view in the 2D well trajectory corresponds to a map view of the well path. The trajectory displayed corresponds to the X and Y offset of the well path.

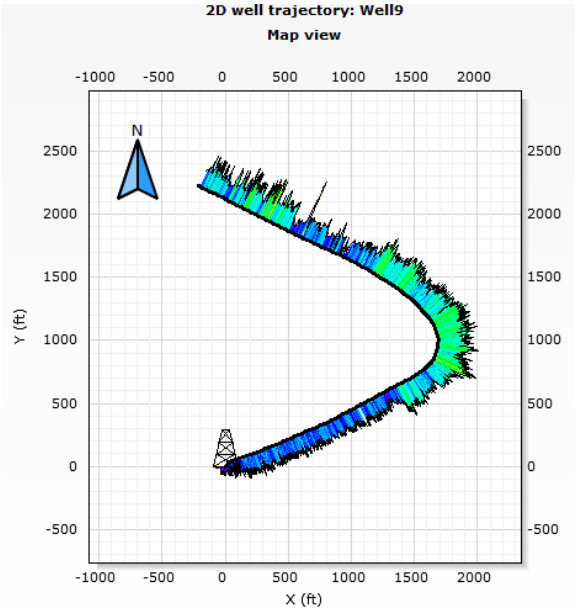


Figure 1-7 Top view

- Curtain section view: the trajectory corresponds to the True Vertical Depth vs. True Horizontal length of the well path.

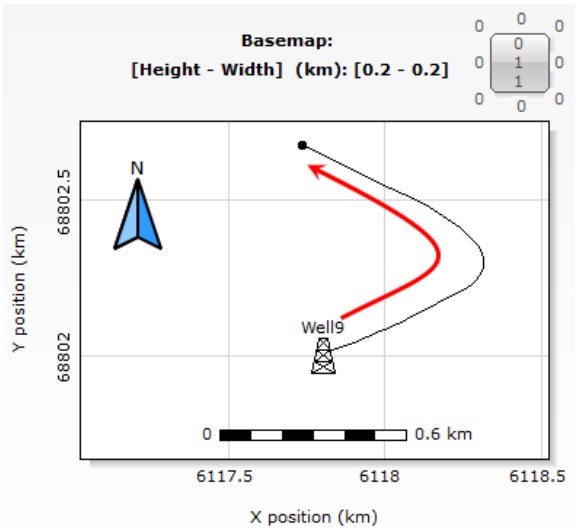


Figure 1-8 Curtain-section view

This example creates a 2D well trajectory plot and add logs in the curtain section view:

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

Workspace workspace = Session::current().currentWorkspace();
Project project = Session::current().mainProject();

Well well = project.getWell("well");

Dataset dipDataset = well.getDataset("Dip_All");
Variable dip = dipDataset.getVariable("Dip_TRU");

Dataset dataset = well.getDataset("ecoscope");
Variable density = dataset.getVariable("ROBB");

Well2DTrajectoryPlot well2DTrajectoryPlot =
Well2DTrajectoryPlot::create(workspace);

well2DTrajectoryPlot.setDipVariable(dip);
well2DTrajectoryPlot.setVariable1(density);

well2DTrajectoryPlot.setDisplayType(
Well2DTrajectoryDisplayType_CurtainSectionView);

lock.release();
```

This screenshot shows the **`Well2DTrajectoryPlot`** in curtain section view:

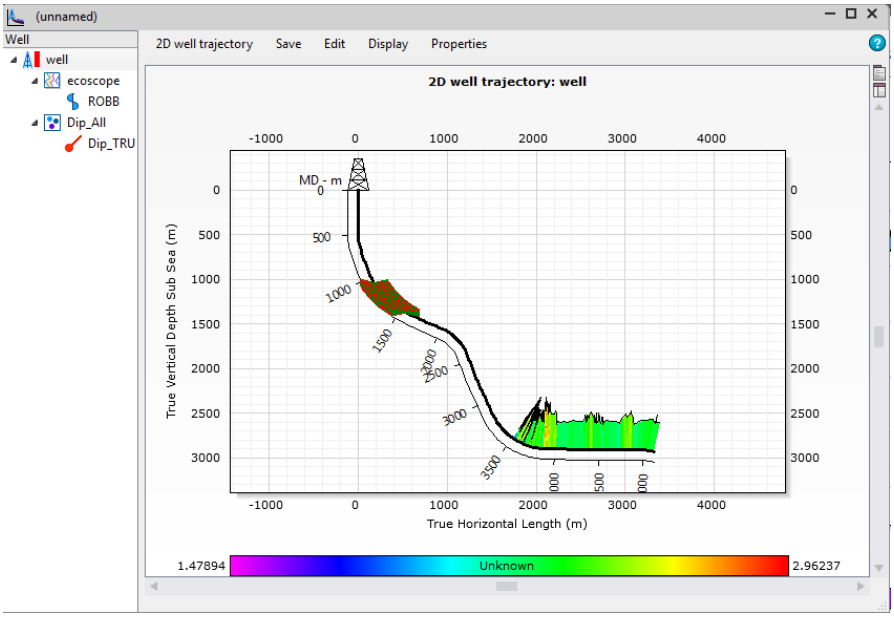


Figure 1-9 2D well trajectory plot in curtain-section view

Well2DGeometry domain object

Step 2 - Add base polygons

The **`Well2DGeometry`** domain object allows you to draw base polygons in the **`Well2DTrajectoryPlot`**. Using base polygons, represent lithology values (pattern or color) along the borehole trajectory with some geological boundaries.

The **`Well2DGeometry`** object cannot be created and the only way to get a **`Well2DGeometry`** instance is from the **`Well2DTrajectoryPlot`** object using the **`findGeometry`** method.

```
class Well2DTrajectoryPlot : public Plot
{
public:
  ...
  Well2DGeometry findGeometry() const;
};
```

Note: The geometry instance is internally created by the 2D well trajectory plot and cannot be created and erased.

```
class Well2DGeometry : public DomainObject
{
public:
  ...
  ReturnValue<bool> trySetBasePolygon(const QPolygonF
    &basePolygon);
  ReturnValue<bool> trySetBasePolygons(const QList<QPolygonF>
    &basePolygons);
```

```
  QList<QPolygonF> polygons() const;
  void clear();
};
```

Add polygons to the plot with two public methods from the **`Well2DGeometry`** object:

- **`trySetBasePolygon`** – initializes the geometry with one base polygon
- **`trySetBasePolygons`** – initializes the geometry with a list of base polygons

The functions return false if the base polygons cannot be initialized. One reason for the failure may be that a base polygon is overlapping another one.

Create polygons using a **`QPolygonF`** Qt object where the values passed to the object are expressed in the 2D well trajectory plot unit.

This example sets some base polygons.

```
Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock,
well2DTrajectoryPlot);

// Create polygons
QPolygonF polygon1, polygon2, polygon3;

polygon1 << QPointF(-500., 0.) << QPointF(-500.,1400.) <<
QPointF(1000.,1400.) << QPointF(1000.,0.);

polygon2 << QPointF(0.,1500.) << QPointF(0.,2500.) <<
QPointF(500.,3000.) << QPointF(1400.,3000.) <<
QPointF(1900.,2500.) << QPointF(1900.,1500.);

polygon3 << QPointF(2000.,2500.)<< QPointF(1500.,3000.) <<
QPointF(2000.,4000.) << QPointF(3200.,4000.) <<
QPointF(3200.,2500.);

Well2DGeometry geometry = well2DTrajectoryPlot.findGeometry();

if (geometry.isNull())
{
  lock.release();
  return;
}

geometry.clear();

// Try to add base polygons
if (!geometry.trySetBasePolygons(QList<QPolygonF>() <<
polygon1 << polygon2 << polygon3))
{
  lock.release();
  return;
```

```
}

lock.release();
```

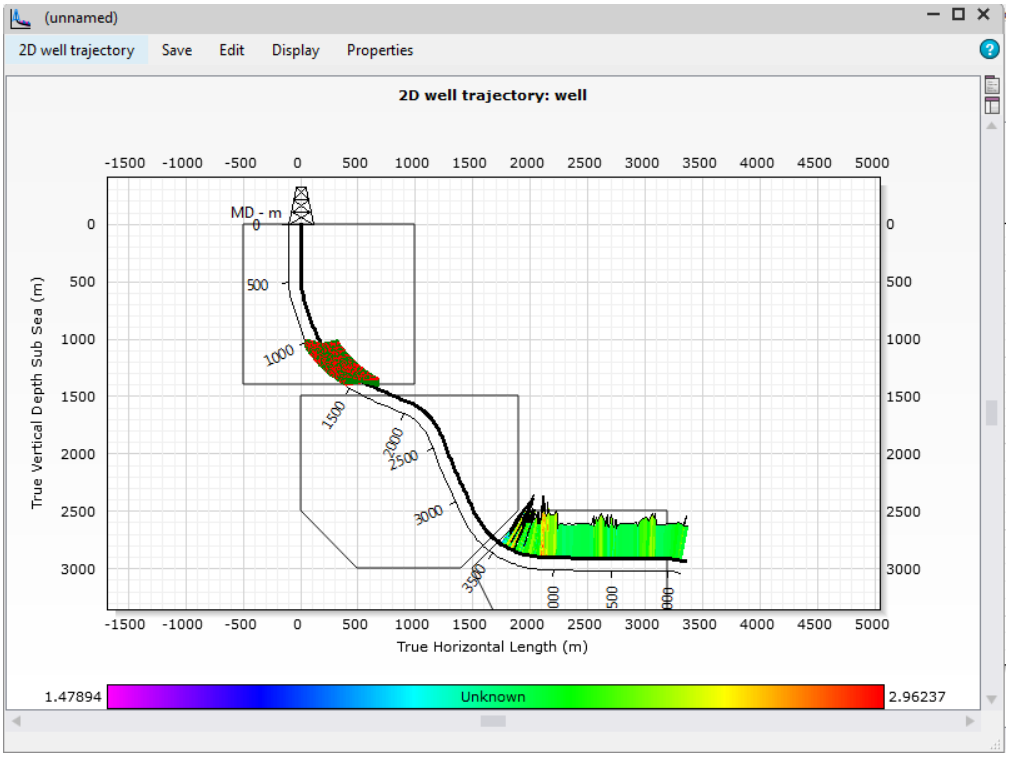


Figure 1-10 Base polygons added to the 2D well trajectory plot

Step 3 - Set polygon properties

The next step is to fill the base polygons with values: you must populate a list of properties associated with a palette. **`Well2DGeometry`** class holds methods to manipulate this list of properties.

```
class Well2DGeometry : public DomainObject
{
public:
  ...
  QStringList propertyFamilyNames() const;
  void addPropertyFamily(const Family &familyName, const Unit
    &propertyUnit, const QString &paletteName);
  void removePropertyFamily(const Family &familyName);
  void selectPropertyFamily(const Family &familyName);
  ReturnValue<bool> isInsideAnyPolygon(const QPointF &point)
    const;
  double polygonPropertyValue(const QPointF &point, const
    Family &familyName) const;
  void setPolygonPropertyValue(const QPointF &point, const
    Family &familyName, double value);
```

```
};
```

The **addPropertyFamily** adds a new property associated with a Techlog palette. This property is unique with respect to the family name in the list of properties; the family name is the first argument of the function.

Note: The palette is retrieved across all the storage levels of the current session: the user level, the project level, the company level and the Techlog level (in this order). If the palette is not found, the geometry uses the default palette with a name of "Unknown" and its color is a gradient of brown (from white to dark brown).

The next step is to select the property to display in the plot using the **selectPropertyFamily** function. This property displays some values (either patterns or colors) from the associated palette within the base polygons.

Use the **setPolygonPropertyValue** function to set a property value for a base polygon; the first argument is the point contained by the polyon, the second argument is the family name (property name) and the third argument is a value in the range of values of the palette associated with the selected property.

Example:

```
Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock,
well2DTrajectoryPlot);

Well2DGeometry geometry = well2DTrajectoryPlot.findGeometry();

if (geometry.isNull())
{
  lock.release();
  return;
}

// add a new property named "lithology"
geometry.addPropertyFamily("lithology", "", "LITH_10");

lock.release();

lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock, geometry);

// select the property to display in the plot
geometry.selectPropertyFamily("lithology");

//set some polygons "lithology" property values
geometry.setPolygonPropertyValue(QPointF(0.,500.),
"lithology", 1);
geometry.setPolygonPropertyValue(QPointF(1000., 2000.),
"lithology", 2);
geometry.setPolygonPropertyValue(QPointF(2500., 3500.),
"lithology", 3);
```

```
lock.release();
```

This screenshot shows three first patterns of "LITH_10" palette applied to the base polygons of **`Well2DGeometry`** object:

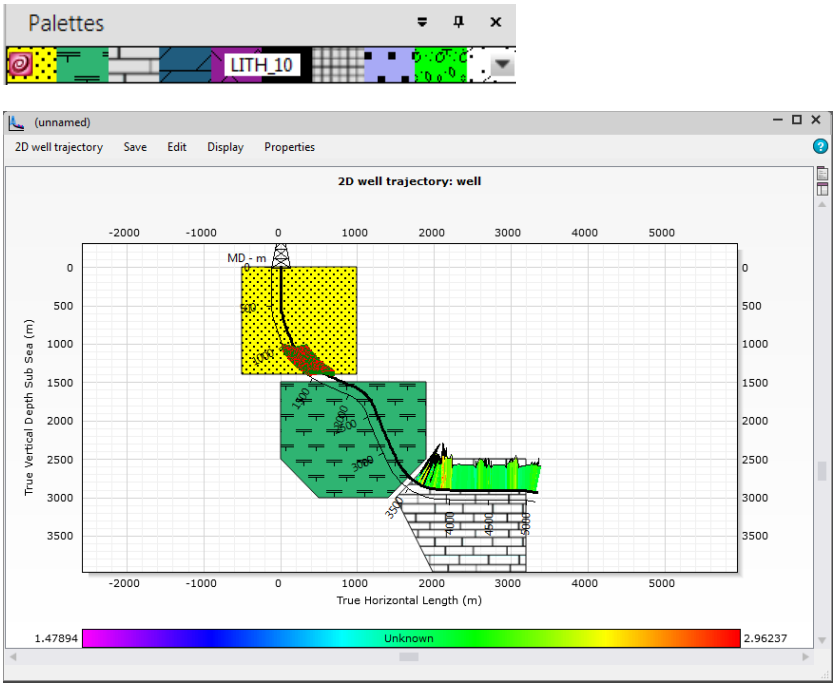


Figure 1-11 Set polygon properties

Note: Only one property can be active at a time. You cannot apply values of different properties to **`Well2DGemoetry`** object. Only values of the palette associated with the selected property are displayed in 2D well trajectory plot.

Step 4 - Add boundaries to the geometry

The last step of 2D well trajectory plot implementation is to add some boundaries within the base polygons; do this using the **`Well2DGeometry`** domain object.

```
class Well2DGeometry : public DomainObject
{
public:
  ...
  ReturnValue<bool> tryAddBoundary(const QList<QPointF>
    &points, QPolygonF &polygon1, QPolygonF &polygon2);
};
```

Add a new boundary to the base polygon using the **`tryAddBoundary`** function. The boundary is drawn from the list of points (polyline) which must be within the base polygon. If one of the points is outside the base polygon the function returns false and the boundary is not created.

If boundary creation is successful, the function returns true and the new boundary is extrapolated to the limits of the base polygon. A boundary splits a polygon into two

polygons and the original one is removed. This method returns the two polygons created. If adding a boundary fails, the two returned polygons are empty.

This example shows where two boundaries are added to a base polygon representing geological structures with their facies along the borehole.

```
Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock,
geometry);
// add boundaries to base polygon
QPolygonF polygonArg1, polygonArg2;
geometry.selectPropertyFamily("lithology");

if (geometry.tryAddBoundary(QList<QPointF>() << QPointF(-
250.,600.) << QPointF(0.,400.) << QPointF(250.,300.) <<
QPointF(500.,400.) << QPointF(750.,600.), polygonArg1,
polygonArg2))
{
  geometry.setPolygonPropertyValue(QPointF(0.,100.),
"lithology", 4);
}

if (geometry.tryAddBoundary(QList<QPointF>() << QPointF(-
250.,1000.) << QPointF(0.,800.) << QPointF(250.,700.) <<
QPointF(500.,800.) << QPointF(750.,1000.), polygonArg1,
polygonArg2))
{
  geometry.setPolygonPropertyValue(QPointF(0.,700.),
"lithology", 5);
  geometry.setPolygonPropertyValue(QPointF(0.,1200.),
"lithology", 9);
}

lock.release();
```

Two anticlinal layers are added to the top base polygon as shown in this screenshot.

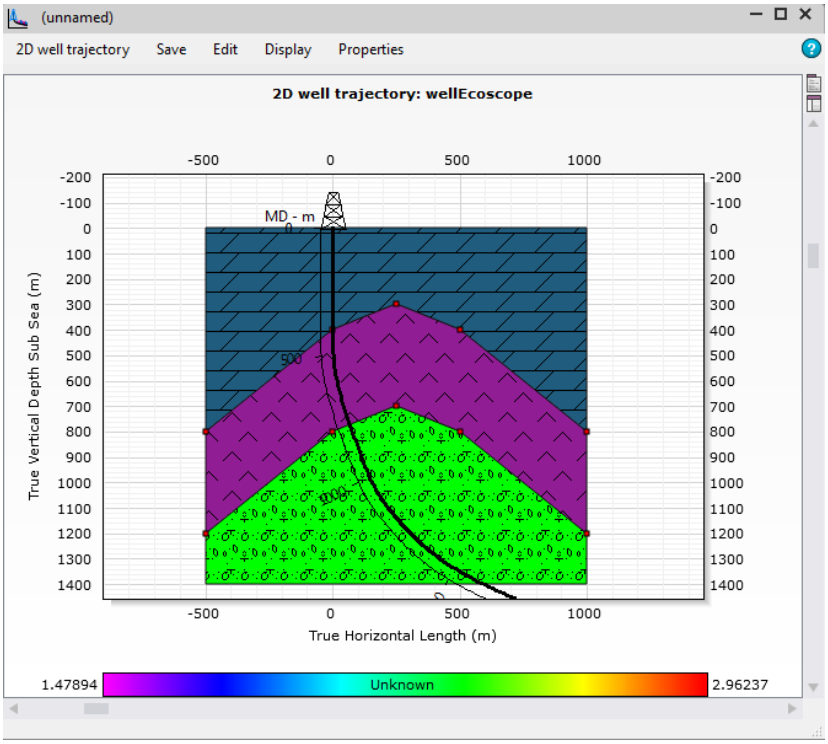


Figure 1-12 Boundaries added to a base polygon

Note: The boundaries can be calculated through any well log curves, like dip values.

There is one limitation when creating boundaries. A new boundary cannot cross an existing one. The **`tryAddBoundary`** function returns false when more than two polygons have been created.

```
lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock, geometry);

bool result = geometry.tryAddBoundary(QList<QPointF>() <<
QPointF(-250.,400.) << QPointF(0.,200.) << QPointF(250.,100.)
<< QPointF(500.,200.) << QPointF(750.,700.), polygonArg1,
polygonArg2);
if (!result)
  qWarning() << "Boundary can't be created";

lock.release();
```

The boundary shown in this screenshot is rejected because it crosses the first one.

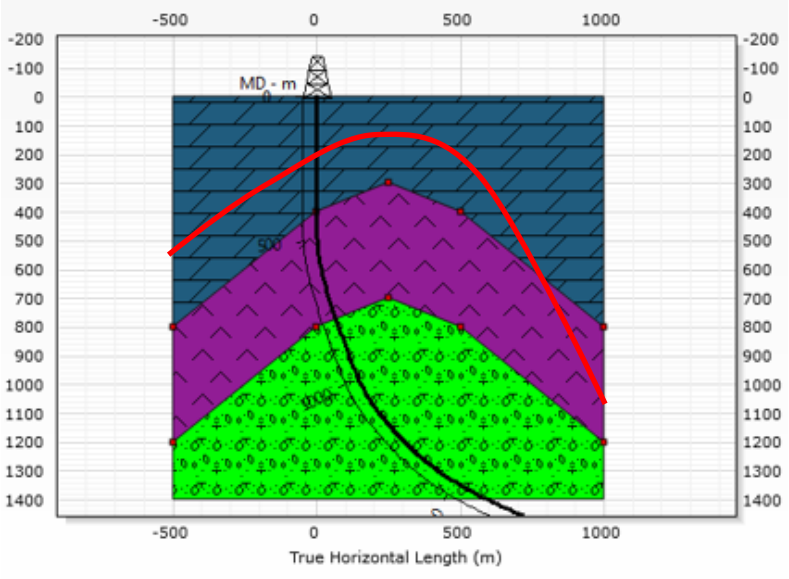


Figure 1-13 Invalid boundary in red

Well2DStrikeAngle object

In the cross-section view, the well trajectory is projected on the line of the section with a given angle. Natively in Techlog there is only one line of section angle for all the trajectories.

The end user modifies this parameter using the line of section toolbar in the 2D well trajectory plot or in the **Properties > Trajectory > Line of section**.

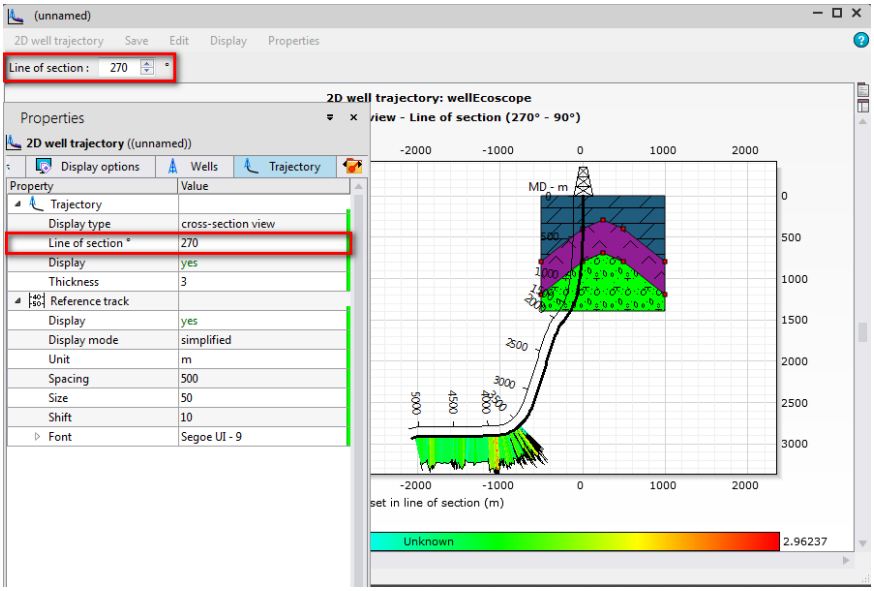


Figure 1-14 Line of section angle

With Ocean, you can have different sections of the well trajectory projected on different angles when you pass a list of angles to the 2D well trajectory plot. Do this with the **`setStrikeAngles`** function of **`Well2DTrajectoryPlot`** class.

```
class Well2DTrajectoryPlot : public Plot
```

```
{
public:
  ...
  QList<Well2DStrikeAngle> strikeAngles() const;
  void setStrikeAngles(const QList<Well2DStrikeAngle>
    &strikeAngles);
};
```

This method takes a list of **`Well2DStrikeAngle`** objects containing angle and bottom values of the well trajectory section.

```
class Well2DStrikeAngle
{
public:
  double angle() const;
  void setAngle(double angle);
  double bottom() const;
  void setBottom(double bottom);
};
```

The first angle is applied from well head to the bottom strike angle and the next angle is applied from the previous bottom strike angle to the current one.

This example has the well trajectory projected with an angle of 45° from the well head to 1500 meters and then projected with an angle of 270° from 1500 meters to 3000 meters.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

Workspace workspace = Session::current().currentWorkspace();
Project project = Session::current().mainProject();

Well well = project.getWell("well");

Dataset dipDataset = well.getDataset("Dip_All");
Variable dip = dipDataset.getVariable("Dip_TRU");

Dataset dataset = well.getDataset("ecoscope");
Variable density = dataset.getVariable("ROBB");

Well2DTrajectoryPlot well2DTrajectoryPlot =
Well2DTrajectoryPlot::create(workspace);

well2DTrajectoryPlot.setDipVariable(dip);
well2DTrajectoryPlot.setVariable1(density);

well2DTrajectoryPlot.setDisplayType(
Well2DTrajectoryDisplayType_CrossSectionView);

QList<Well2DStrikeAngle> strikeAngles;
```

```
strikeAngles << Well2DStrikeAngle(45.,1500.)
<< Well2DStrikeAngle(270.,3000.);

well2DTrajectoryPlot.setStrikeAngles(strikeAngles);

lock.release();
```

In this screenshot, the well trajectory projection changes from 1500 meters.

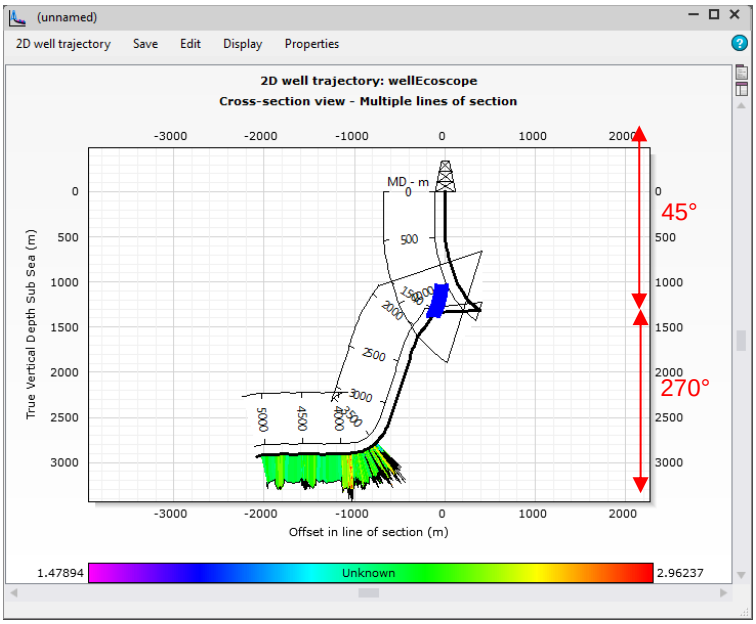


Figure 1-15 Well trajectory sections with different angles

Stereonet plot

A stereonet is a lower hemisphere graph on which you may plot a variety of geological data. Imagine a sphere with lines of latitude and longitude marked. A stereonet is the plane of projection of the lower half of this sphere – it is a lower hemisphere graph.

Stereonet plots in Techlog provide a comprehensive range of dip visualization and statistics functions. Projections available are simple azimuth and strike histograms (rose diagrams), polar plots, Schmidt plots, and Wulff plots. Data is displayed as lines (dip and azimuth) and planes or poles to planes in either upper or lower hemisphere projections (Schmidt and Wulff plots).

Stereonet plot is accessible from the **Geology** > **Wbi** > **Stereonet plot**.

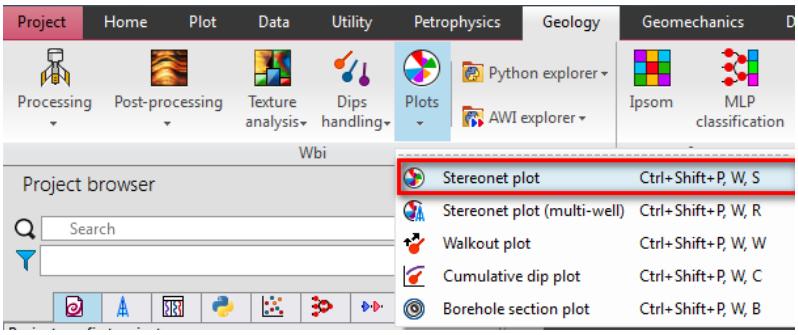


Figure 1-16 Stereonet plot in Techlog

The **Stereonet** window is composed of two primary areas:

- Variables side box: to drag dip, azimuth and color variables to be displayed.
- Display area: where the dip visualization is rendered following display and projection types.

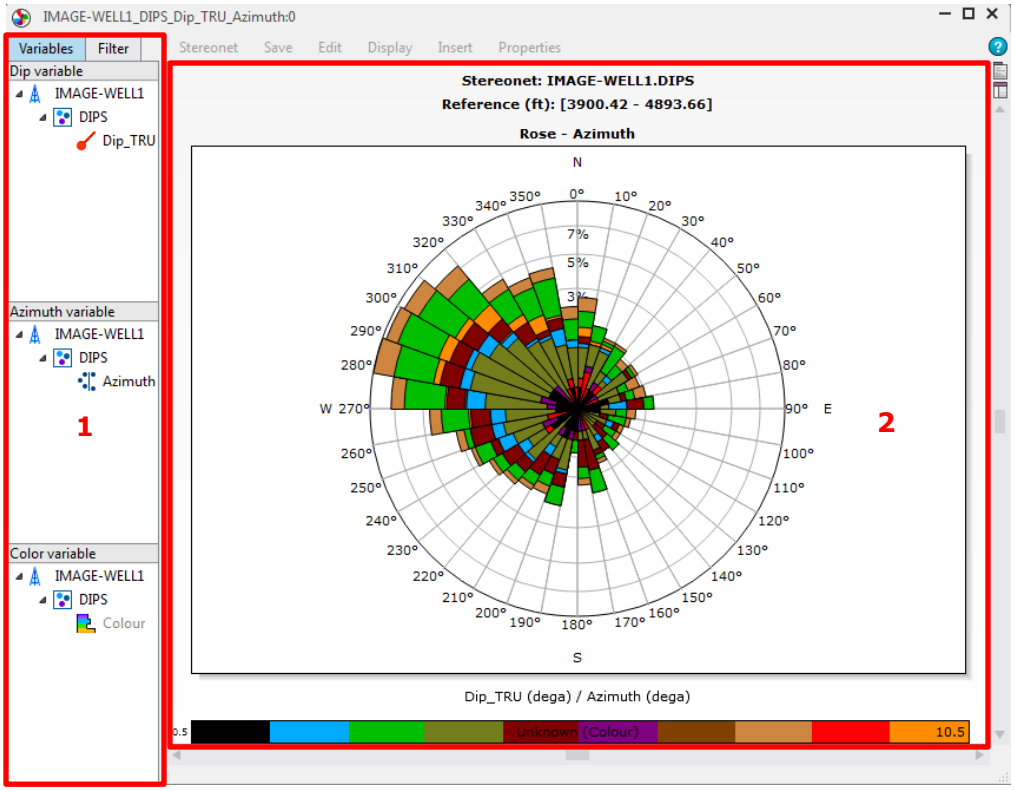


Figure 1-17 Stereonet window

When you drop a dip variable into the **Stereonet** plot, the corresponding azimuth, color and type variables are automatically loaded by the plot.

This association is done through properties with reserved names.

The true dip inclination variable is associated with:

- The true dip azimuth variable with property name "dipAzimuthVariable".
- The dip type variable with property name "dipTypeVariable". The associated variables are:
  - Color variable that stores the color for each dip with property name "dipColorVariable". When a dip variable is dragged into the **Stereonet** plot, Techlog uses the association of the type variable to determine the type of each dip and uses the association of the color variable to the type variable to determine the display color.
  - Shape variable that stores the display symbol for each dip with property name "dipShapeVariable" (this is not used by the **Stereonet** plot).

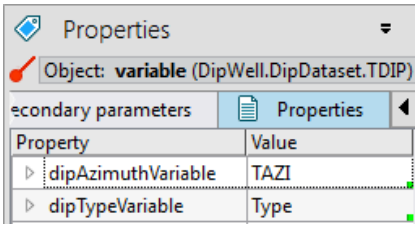


Figure 1-18 Variables associated to dip variable through properties

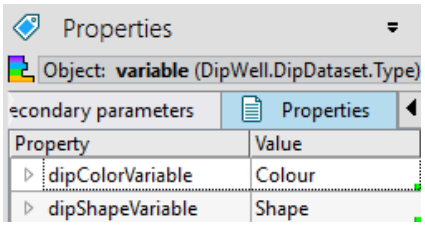


Figure 1-19 Variables associated to type variable through properties

Note: With Ocean, the dip associated variables are not considered when a dip variable is set on the **Stereonet** plot. The developer must independently set the dip, azimuth and color variables.

StereonetPlot domain object

The **`StereonetPlot`** domain object represents a stereonet plot. This domain object is used to create a stereonet plot as well as add dip, azimuth and color variables to it.

The **`StereonetPlot`** class inherits from the **`Plot`** class and does not support name. A **`StereonetPlot`** object is instantiated through the static **`create`** method passing the parent **`Workspace`** object as the argument. The uniqueness of a **`StereonetPlot`** object is managed by the system. The only way to retrieve a **`StereonetPlot`** object from the workspace is by its droid, typically stored as a private member variable of the plug-in.

A **`StereonetPlot`** can also be added to a container plot (matrix-plot) using the dedicated static **`create`** methods.

See the "ContainerPlot domain object" section in *Ocean for Techlog Developer Guide - Plots* for more information on how to create a Stereonet plot in a container plot.

```
class StereonetPlot : public Plot
{
public:
  ...
  static StereonetPlot create(Workspace workspace);
  void setDipVariable(const Variable &dipVariable);
  void unsetDipVariable();
  const Variable findDipVariable() const;
  void setAzimuthVariable(const Variable &azimuthVariable);
  void unsetAzimuthVariable();
  const Variable findAzimuthVariable() const;
  void setColorVariable(const Variable &colorVariable);
  void unsetColorVariable();
  const Variable findColorVariable() const;
  DisplayType displayType();
  void setDisplayType(DisplayType displayType);
};
```

Set the dip, azimuth and color variables for the plot with the public functions:

- **`setDipVariable`**, **`unsetDipVariable`** and **`findDipVariable`** allow you to respectively add, remove and retrieve a dip variable with **`VariableTypeDip`** type and a variable unit compatible with degree angle.
- **`setAzimuthVariable`**, **`unsetAzimuthVariable`** and **`findAzimuthVariable`** allow you to respectively add, remove and retrieve an azimuth variable with variable unit compatible with degree angle.
- **`setColorVariable`**, **`unsetColorVariable`** and **`findColorVariable`** allow you to respectively add, remove and retrieve a color variable.

The **`setDisplayType`** function switches the stereonet plot from one data display mode to another one. The function takes an enum value of **`DisplayType`** enum class.

```
enum DisplayType
{
  DisplayTypePolar,
  DisplayTypeRose,
  DisplayTypeSchmidt,
  DisplayTypeWulff
};
```

The display types are:

- Polar: data is displayed in a polar plot grid. The dip value is displayed increasing outwards along radius, and the azimuth is displayed around the circumference.

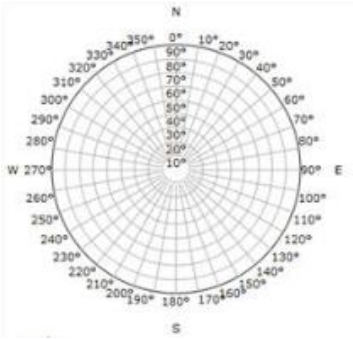


Figure 1-20 Polar mode

- Rose: data is displayed as a rose diagram (this is the default value returned by **`displayType`** method after **`StereonetPlot`** creation).

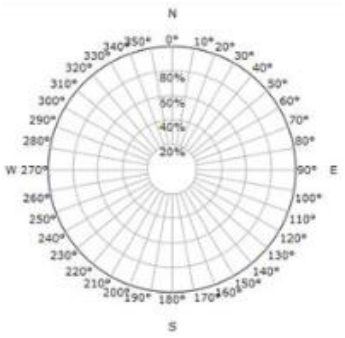


Figure 1-21 Rose mode

- Schmidt: data is displayed with a Schmidt projection.

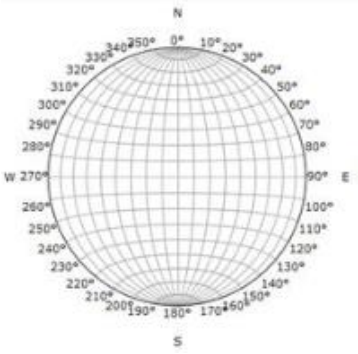


Figure 1-22 Schmidt mode

- Wulff: data is displayed with a Wulff projection.

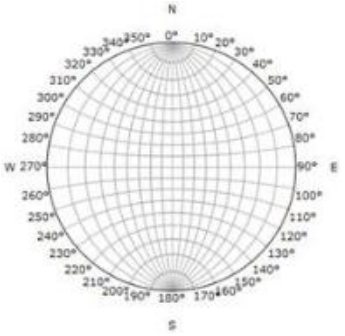


Figure 1-23 Wulff mode

Note: The projection type is not currently exposed in Ocean. The default "pole" projection type value is applied when the display type is set to Schmidt or Wulff and you cannot change it programmatically.

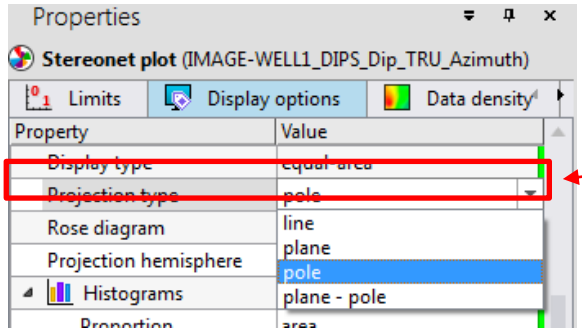


Figure 1-24 Display type property

This example creates a stereonet plot.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

Project project = Session::current().mainProject();
Well well = project.wells().get("well");
Dataset dataset = well.datasets().get("Dip_All");

Variable dip = dataset.variables().get("Dip_TRU");
Variable azimuth = dataset.variables().get("Azimuth");
Variable colour = dataset.variables().get("Colour");

Workspace workspace = Session::current().currentWorkspace();

StereonetPlot stereonet = StereonetPlot::create(workspace);

stereonet.setDipVariable(dip);
stereonet.setAzimuthVariable(azimuth);
stereonet.setColorVariable(colour);
// set display type to Polar (Rose by default)
stereonet.setDisplayType(DisplayTypePolar);
lock.release();
```

In polar mode, the stereonet plot displays a dip data point as a vector in the direction of the dip with a rose histogram of dip azimuth values.

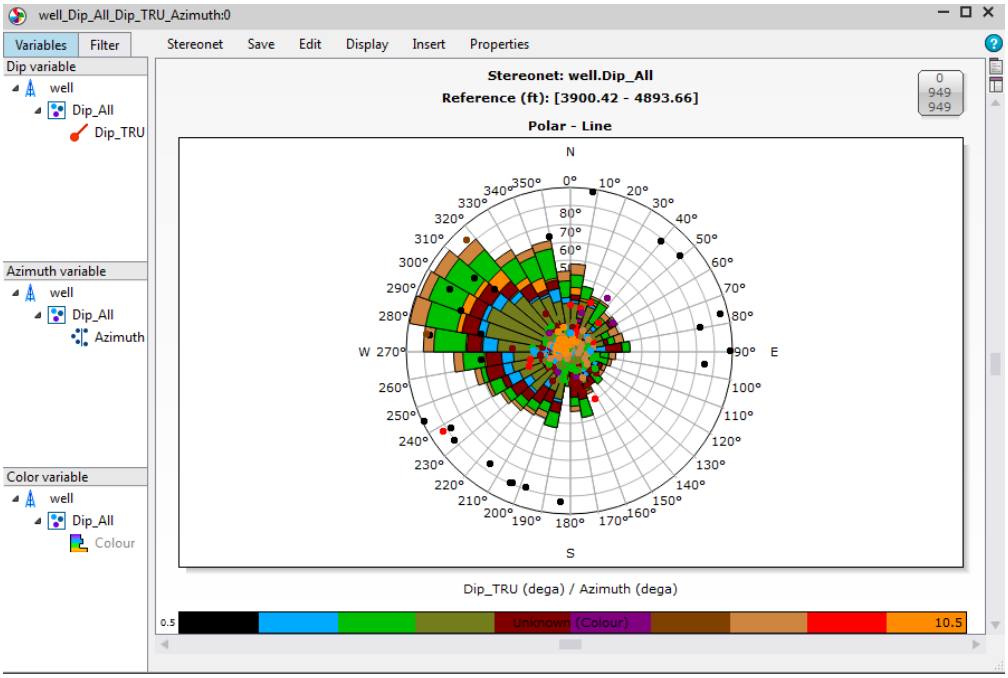


Figure 1-25 Stereonet plot in polar display type

The **`StereonetPlot`** has additional display options.

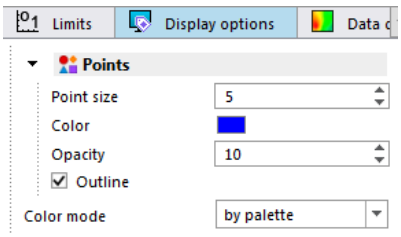
```
class StereonetPlot : public Plot
{
public:
  ...
  GlobalZonation findSelectedZonation() const;
  QList<GlobalZone> selectedZones() const;
  void setSelectedZones(const GlobalZonation
    &selectedZonation, const QList<GlobalZone> &selZones);

  QColor pointsColor();
  void setPointsColor(const QColor &color);
  float pointsSize();
  void setPointsSize(float size);
  bool isPointOutlineVisible() const;
  void setPointOutlineVisible(const bool visible);

  QString paletteName() const;
  void setPalette(const QString &paletteName, StorageLevel
    level);

  ColorType colorType() const;
  void setColorType(const ColorType colorType);

  RoseDiagram roseDiagram() const;
  void setRoseDiagram(const RoseDiagram roseDiagram);
```

```
  GridStep gridStep() const;
  void setGridStep(const GridStep gridStep);

  bool isDisplayTypeEditableByUser() const;
  void setDisplayTypeEditableByUser(const bool
    isDisplayTypeEditableByUser);
  bool isProjectionTypeEditableByUser() const;
  void setProjectionTypeEditableByUser(const bool
    isProjectionTypeEditableByUser);
  bool isRoseDiagramEditableByUser() const;
  void setRoseDiagramEditableByUser(const bool
    isRoseDiagramEditableByUser);
  bool isProjectionHemisphereEditableByUser() const;
  void setProjectionHemisphereEditableByUser(const bool
    isProjectionHemisphereEditableByUser);

  ProjectionHemisphere projectionHemisphere() const;
  void setProjectionHemisphere(const ProjectionHemisphere
    hemisphere);

  bool isGraduationVisible() const;
  void setGraduationVisible(const bool visible);
  GraduationOrder graduationOrder() const;
  void setGraduationOrder(const GraduationOrder order);
  QFont graduationFont() const;
  void setGraduationFont(const QFont font);
  QColor graduationColor() const;
  void setGraduationColor(const QColor color);

  bool isColorScaleVisible() const;
  void setColorScaleVisible(bool visible);

  bool isSideBoxVisible() const;
  void setSideBoxVisible(bool visible);

  bool isFullWindowDisplay() const;
  void setFullWindowDisplay(bool isFullWindowDisplay);
};
```

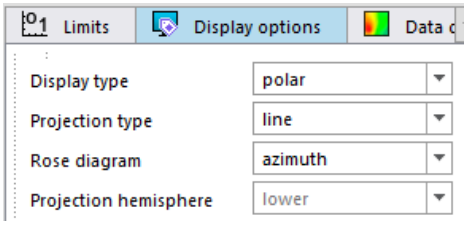
Use the additional display options to:

- select some **`GlobalZone`** in a **`GlobalZonation`** and display it in the **`CrossPlot`**
- change the size and color of the dip data points
  - the color property is overridden by the color variable set on the plot
- show or hide point outline

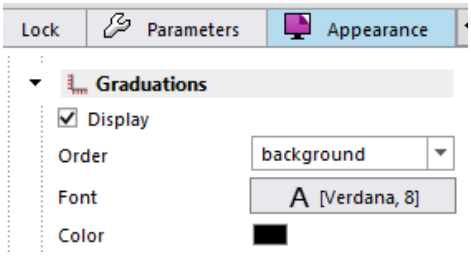
- set a color palette value to use in the **`StereonetPlot`** at a storage level (Techlog, Company, Project, User, Plug-in folder)
  - the palette overrides the colors defined in the color variable
- change the points coloring mode to palette (**`paletteName`** is applied), source (**`pointsColor`** is applied) or zone (colors of **`selectedZones`** are applied)



- get and set the **`RoseDiagram`** display type as azimuth, azimuth mirror, strike or none
- get and set the value in dega of the step displayed in the background grid of the stereonet
- enable or disable the display type, projection type, rose diagram and projection hemisphere for the end-user. Selecting a display type in the properties editor may disable related parameters as projection type, rose diagram and projection hemisphere. In this case attempting to enable a disabled parameter through one of these functions will fail.
- set the **`ProjectionHemisphere`** to lower or upper value



- set the **`ProjectionHemisphere`** to lower or upper value
- show or hide the graduations display around stereonet canvas
- change display order between the background and the foreground
- get and set the graduation font and color



- show or hide the variables side box on the left side of the **`StereonetPlot`**
- show or hide the coloration scale below the **`StereonetPlot`** chart area

GreatCircle object

Imagine a plane (defined by a vector in the direction of the dip) cutting through the center of a lower hemisphere (figure 1-26).

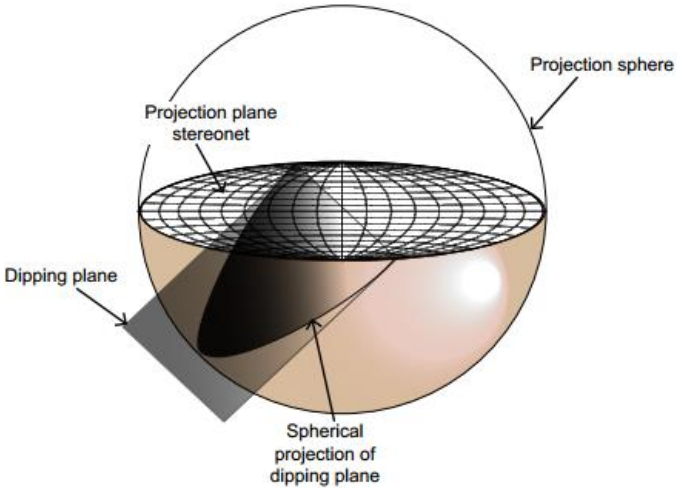


Figure 1-26 Dipping plane

The stereonet constitutes the surface of this lower hemisphere. There is an arc where the plane touches the edge of the lower hemisphere. This arc is projected back up on to the stereonet to form a great circle as shown in Figure 1-27.

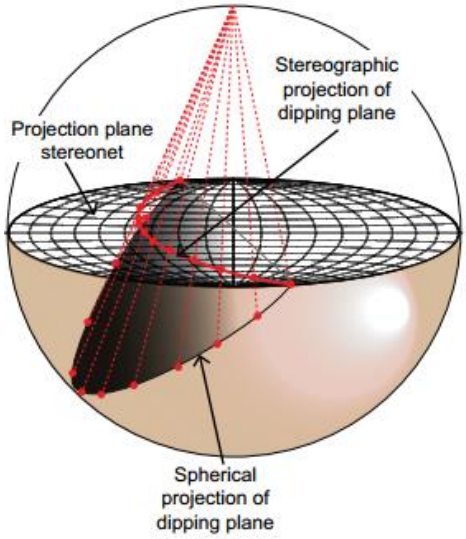


Figure 1-27 Intersection between a plane and the displayed hemisphere

Figure 1-28 shows the resulting plot.

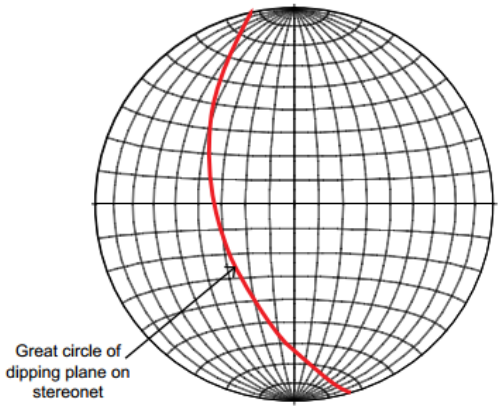


Figure 1-28 Great circle

The **GreatCircle** class represents this great circle; it is not a domain object. A **GreatCircle** object is instantiated through a constructor.

```
class GreatCircle
{
public:
  GreatCircle(const QString &name);
  GreatCircle(const QString &name, quint64 dataCount, float
    dip, float azimuth, const QColor &color, float markerSize,
    bool isInteractive);
  const QString& name() const;
  quint64 dataCount() const;
  float dip() const;
  float azimuth() const;
  const QColor& color() const;
  float markerSize() const;
  bool isInteractive() const;
  bool isEmpty() const;
};
```

The **StereonetPlot** domain object contains a list of **GreatCircle** objects.

```
class StereonetPlot : public Plot
{
public:
  ...
  QList<GreatCircle> greatCircles() const;

  void addGreatCircle(const GreatCircle &greatCircle);
  void updateGreatCircle(const GreatCircle &greatCircle);
  void removeGreatCircle(const QString &greatCircleName);
  void setGreatCircles(const QList<GreatCircle>
    &greatCircles);
  bool greatCirclesVisible() const;
  void setGreatCirclesVisible(bool visible);
  bool constainsGreatCircle(const QString &name) const;
```

```
  GreatCircle findGreatCircle(const QString &name) const;
  GreatCircle getGreatCircle(const QString &name) const;
};
```

The **`GreatCircle name`** property is used as a unique identifier within the list of **`GreatCircle`** objects contained by **`StereonetPlot`** domain object. The name of the property cannot be empty and name uniqueness is validated by the parent **`StereonetPlot`** when adding or updating it. You must always check for the existence of a **`GreatCircle`** object with a given name before adding a new one to the **`StereonetPlot`** domain object.

A **`GreatCircle`** is a value-semantic object, which means that it is created in the stack. It has no knowledge of its parent. To modify (add, update, or remove) it in its parent **`StereonetPlot`**, you must modify it locally, and then pass it to its parent to be changed in the parent container.

Note: If **`GreatCircle`** cannot be found in the **`StereonetPlot`**, the **`isEmpty`** method returns true.

Access **`GreatCircle`** property values using the corresponding getters from the **`GreatCircle`** instance. The property values can be set only at the object creation.

Note: To project the dipping plane on the lower hemisphere the stereonet plot must be either the Schmidt or Wulff display type. A great circle is not displayed when **`DisplayType`** is either polar or rose.

This example adds a great circle to the stereonet plot.

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

Project project = Session::current().mainProject();
Well well = project.wells().get("well");
Dataset dataset = well.datasets().get("Dip_All");

Variable dip = dataset.variables().get("Dip_TRU");
Variable azimuth = dataset.variables().get("Azimuth");
Variable colour = dataset.variables().get("Colour");

Workspace workspace = Session::current().currentWorkspace();

StereonetPlot stereonet = StereonetPlot::create(workspace);

stereonet.setDipVariable(dip);
stereonet.setAzimuthVariable(azimuth);
stereonet.setColorVariable(colour);

stereonet.setDisplayType(DisplayTypeSchmidt);

GreatCircle greatCircle ("greatCircle1", 1, 50., 60., Qt::red,
2, true);
if (!stereonet.constainsGreatCircle("greatCircle1"))
  stereonet.addGreatCircle(greatCircle);
```

```
else
  stereonet.updateGreatCircle(greatCircle);

lock.release();
```

`GreatCircle` property values set through the constructor are shown in this screenshot:

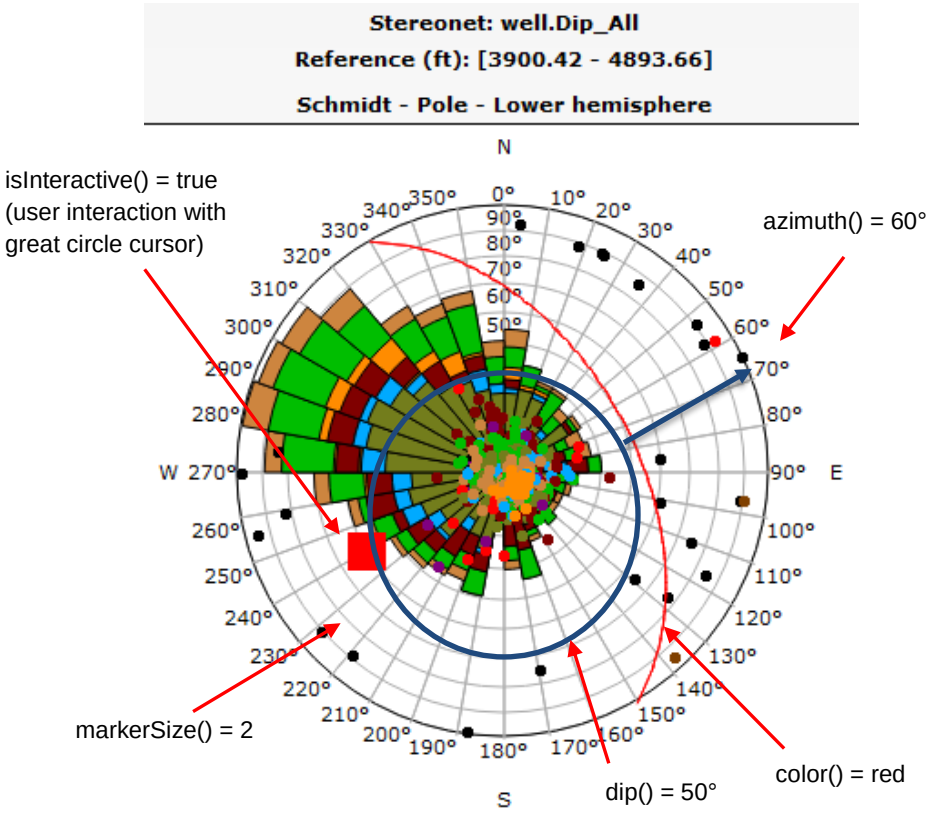


Figure 1-29 Great circle properties

StereonetPlot signal

Subscribe to the **`GreatCircleChanged`** signal for a **`StereonetPlot`** domain object to be notified if one of its **`GreatCircle`**s has changed.

```
class StereonetPlot : public Plot
{
public:
  ...
  enum EventType
  {
    GreatCircleChanged
  };
}
```

This signal is triggered when the dip or azimuth values of the great circle have changed. The **`GreatCircle`** object must have the **`isInteractive`** property set to true to change its dip and azimuth values by moving the great circle cursor.

Include the signal argument and declare the slot receiver in the activity header file:

```
#include "tsdkgreatcirclechangedargs.h"
```

```
private slots:
  void onGreatCircleChanged(const
  Slb::Ocean::Techlog::GreatCircleChangedArgs &args);
```

A **`StereonetPlot`** domain object connects to this signal.

```
Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock,
stereonet);

stereonet.connect(StereonetPlot::GreatCircleChanged, this,
SLOT(onGreatCircleChanged(const
Slb::Ocean::Techlog::GreatCircleChangedArgs&)));

lock.release();
```

The **`GreatCircleChanged`** signal has a **`GreatCircleChangedArgs`** argument that gives the name of the changed **`GreatCircle`** and all its new property values.

```
class GreatCircleChangedArgs : public
  SignalArgsT<StereonetPlot>
{
public:
  const QString& name() const;
  quint64 dataCount() const;
  float dip() const;
  float azimuth() const;
  const QColor& color() const;
  float markerSize() const;
  bool isInteractive() const;
};
```

This example shows how to change the **`color`** property of the changed **`GreatCircle`**:

```
void activity::onGreatCircleChanged(const
Slb::Ocean::Techlog::GreatCircleChangedArgs &args)
{
  StereonetPlot stereonet = args.sender();

  Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock,
  stereonet);

  QColor color;
  if (args.color() == Qt::red)
    color = Qt::darkBlue;
  else
    color = Qt::red;
```

```
  stereonet.updateGreatCircle(GreatCircle(args.name(),
  args.dataCount(), args.dip(), args.azimuth(), color,
  args.markerSize(), args.isInteractive()));

  qWarning() << "Great circle " << args.name()
  << " has changed and is now colored in " << color.name();

  lock.release();
}
```

This screenshot shows that the color of the great circle is updated when **`GreatCircleChanged`** signal is emitted.

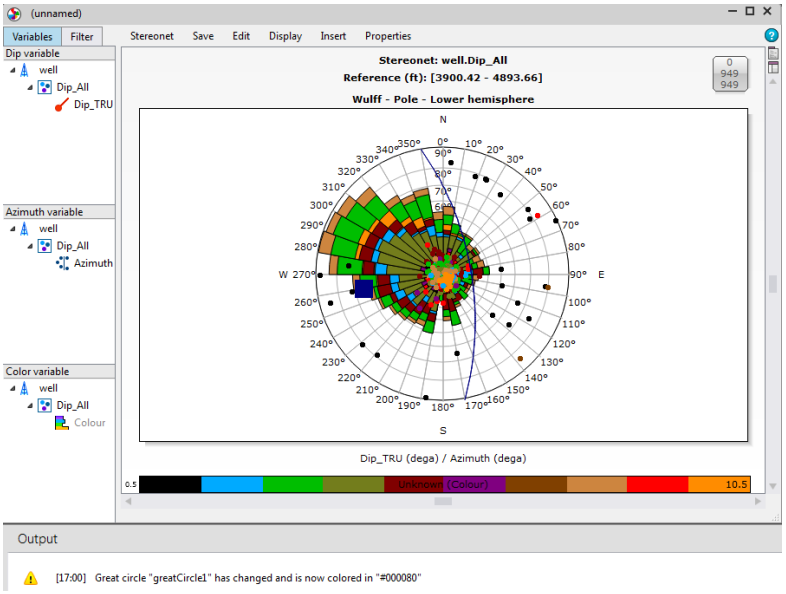


Figure 1-30 Great circle changed signal

2 Dip picking

In This Chapter

Introduction

In Techlog, the end user creates a dip dataset from a borehole image array (variable type: matrix array) or a borehole image raster image (variable type: borehole image). To use dip creation, the borehole image array must have an associated caliper, hole deviation and hole azimuth variable. If the links are not populated, edit them first in Techlog in the **Properties** dock window:

- **caliperDatasetNameVariable**: name of the dataset that contains the associated caliper variable
- **caliperNameVariable**: name of the associated caliper variable used to compute the angular width of the individua lpad arrays and the position of the tool in the hole
- **holeDeviationNameVariable**: name of the associated hole deviation variable used to calculate true dip from the apparent dip picked on the image
- **holeAzimuthNameVariable**: name of the associated hole azimuth variable used to calculate true dip from the apparent dip picked on the image

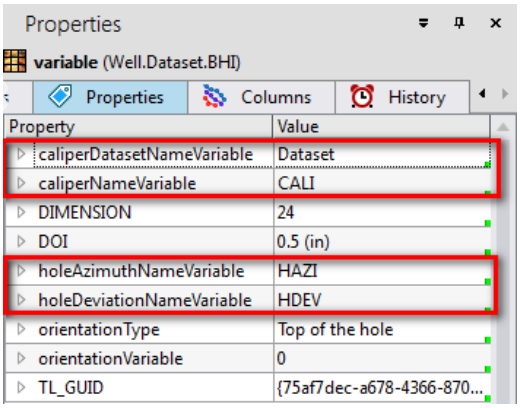


Figure 2-1 BHI associated variables

Dip picking in Techlog is done through the interaction between the end user and the **Logview** GUI:

1. Drop a borehole image variable to the Logview.

2. Right-click the image in the track, and then select "Create a dip variable" in the toolbar. This adds a new track to the **Logview** that contains a temporary dip variable. Once the dip variable is created, you can "Disable dip picking" or "Enable dip picking" .

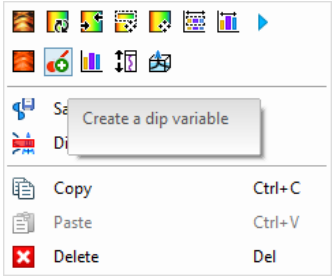


Figure 2-2 Create a dip variable

3. Enable the tool that you want to use pick the dips on the image (only the types of dip listed below are currently exposed with Ocean):
4. Pick three points or more to define a sinusoid. 
5. For the planar features covering only part of the borehole, you can pick dips with partial sinusoids. 

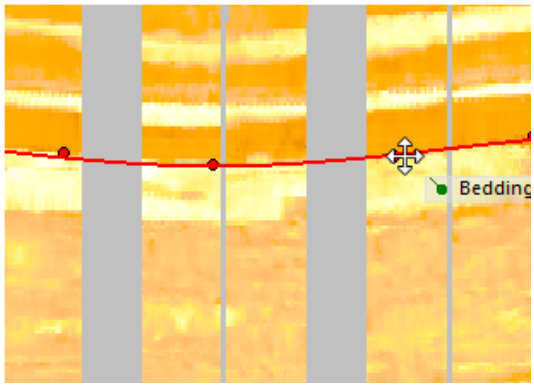


Figure 2-3 Pick dips on the matrix array variable

6. Right-click to validate the dip creation.

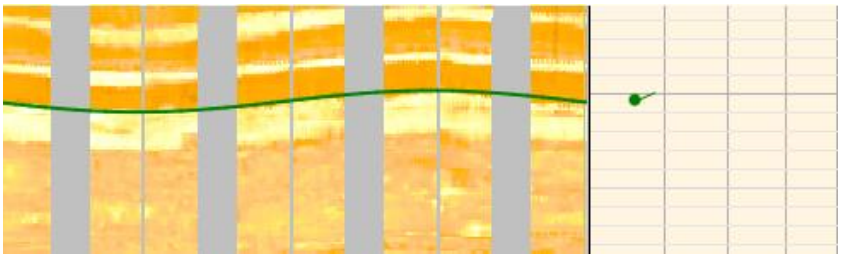


Figure 2-4 Validate dip creation

7. Right-click on the true log dip track to save the new picked dips into a dip variable. After the dip variable is saved, a new dataset is created. The dataset type for dip data is **Point data**.

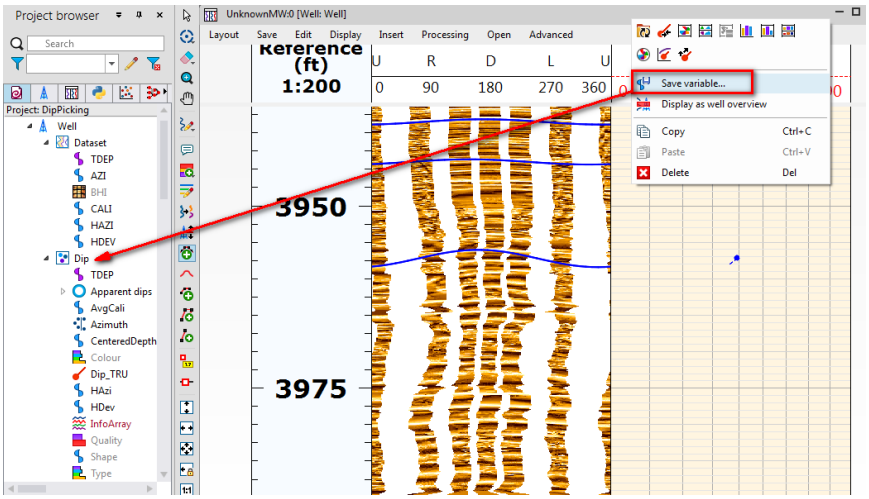


Figure 2-5 Save dip data into dip dataset

Dip picking with Ocean

Ocean gives you the ability to programmatically create a **Logview**, add a track to the **Logview** and display the array borehole image in the track.

See "Logview domain object" in *Ocean for Techlog Developer Guide - Plots* for more information on how to create a Logview and "ArrayBHITrackItem domain object" in *Ocean for Techlog Developer Guide - Plots* for more information on how to display an array variable with the "matrix array" type in the track.

Once the track has been created with the array borehole image, you may enable dip picking through the Ocean dip picking API.

Logview domain object

The **Logview** domain object has methods to enable dip picking and access dip picking results.

```
class Logview : public Plot
{
public:
  void enableDipPickingFor(const ArrayBHITrackItem &item);
  DomainObjectCollection<DipTrackItem>
    trueDipPickingTrackItems() const;
  ...
};
```

Plot the matrix array **Variable** in a **Logview** **NormalTrack** using an **ArrayBHITrackItem** domain object and commit the changes, as shown in this example:

```
Lock lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

Logview logview = Logview::create(workspace);

NormalTrack track =
NormalTrack::create(QLatin1String("ArrayBHI"), logview);

Variable arrayBHI = dataset.variables().get("BHI");

ArrayBHITrackItem BHITrackItem =
ArrayBHITrackItem::create(track, arrayBHI);

lock.release();
```

The **enableDipPickingFor** method allows you to enable dip picking for several matrix array variables plotted through different **ArrayBHITrackItem** domain objects (one matrix array **Variable** per **ArrayBHITrackItem** object).

```
lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock, logview);

logview.enableDipPickingFor(BHITrackItem);
// ...
```

```
lock.release();
```

When the changes on the **Logview** are committed, dip picking is activated for **ArrayBHITrackItem** domain objects added to the list and a new track is added to the **Logview** that contains the temporary true dip variable displayed in the track using a **DipTrackItem** domain object.

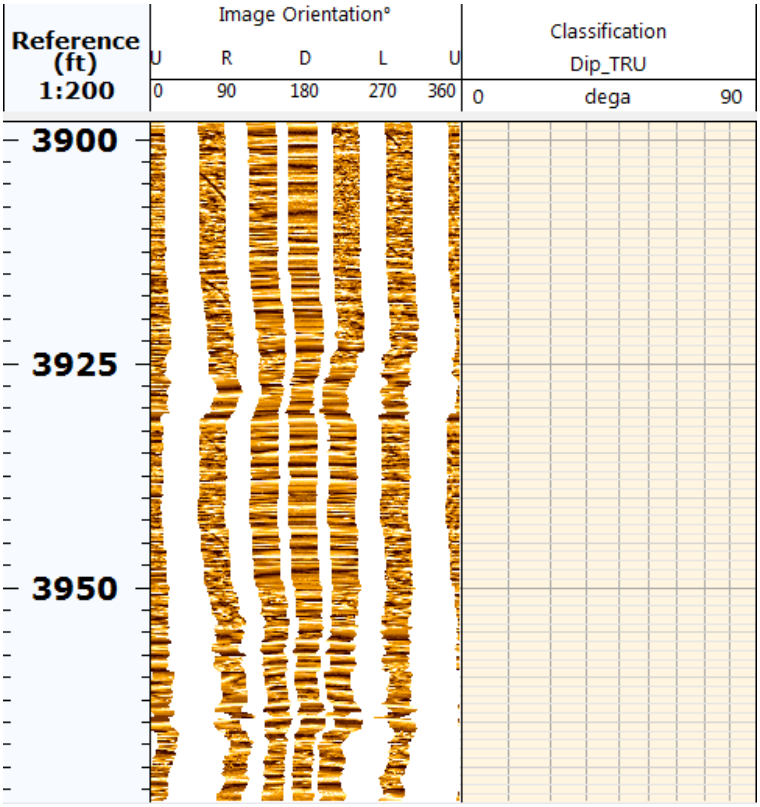


Figure 2-6 DipTrackItem added to Logview

Retrieve the **DipTrackItem** domain object that contains the true dips resulting from dip picking from **trueDipPickingTrackItems** collection, as this example shows.

```
lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock, logview);

DipTrackItem dipTrackItem =
logview.trueDipPickingTrackItems().first();
//...
lock.release();
```

Note: At this stage, the true dip variable is stored in a temporary project and its members are not accessible.

DipTrackItem domain object

The **DipTrackItem** domain object inherits from **TrackItem** base class and is used to display a dip variable (not greater that one column) with "dip" type (**VariableTypeDip**) in the **Logview**.

```
class DipTrackItem : public TrackItem
{
public:
  ...
  static DipTrackItem create(NormalTrack track, const Variable
    &variable);
  DipModel getDipModel() const;
  DipModel findDipModel() const;
  QVector<quint64> selectedDipIndexes() const;

  enum EventType
  {
    DipSelectionChanged
  };
};
```

From the **findDipModel** public method you get a **DipModel** object that handles the dip values and all associated variables.

See the "DipModel domain object" section on page 2-8 for more information on how to manipulate dip data.

When subscribing to the **DipSelectionChanged** signal, a **DipTrackItem** domain object is notified if true dips are selected in the track when interactive selection is turned on. 

Include the signal argument and declare the slot receiver in the activity header file:

```
#include "tsdkdipselectionchangedargs.h"
```

```
private slots:
  void onDipSelectionChanged(const
  Slb::Ocean::Techlog::DipSelectionChangedArgs &args);
```

This example shows how a **DipTrackItem** domain object connects to this signal.

```
lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock, dipTrackItem);

dipTrackItem.connect(DipTrackItem::DipSelectionChanged, this,
SLOT(onDipSelectionChanged(const
Slb::Ocean::Techlog::DipSelectionChangedArgs&)));

lock.release();
```

The **DipSelectionChanged** signal has a **DipSelectionChangedArgs** argument which gives the sender and the indexes of the selected dips in the track from the **DipTrackItem** object using the **selectedDipIndexes** function.

```
class DipSelectionChangedArgs : public
      SignalArgsT<DipTrackItem>
{
};
```

DipModel domain object

The **DipModel** domain object cannot be created; the only way to get a **DipModel** instance is from the **DipTrackItem** domain object using the **findDipModel** method.

```
class DipTrackItem : public TrackItem
{
public:
  DipModel findDipModel() const;
  ...
};
```

The following dip picking actions must be implemented in separated transactions:

- Get the **DipTrackItem** object created when the dip picking mode is enabled on a borehole image array; it is returned from the **Logview::trueDipPickingTrackItems()** collection.
- Retrieve the **DipModel** instance from the **DipTrackItem** domain object.
- Access the members of the **DipModel** domain object.

```
Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock, logview);

DipTrackItem dipTrackItem =
logview.trueDipPickingTrackItems().first();

lock.release();

lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock, dipTrackItem);

DipModel dipModel = dipTrackItem.findDipModel();

lock.release();

lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock, dipModel);

Variable dipVariable = dipModel.dipVariable();
// ...

lock.release();
```

The **DipModel** domain object handles the dip values and all associated variables.

```
class DipModel : public DomainObject
{
public:

  Dip getDip(quint64 index) const;
  void setDip(quint64 index, const Dip &dip);
  QVector<Dip> getDips() const;
  void setDips(const QVector<Dip> &dips);
  void insertDip(const double depth, const Dip &dip);
  void removeDip(const quint64 index);

  DipCategory dipCategory() const;

  const Variable dipVariable() const;

  const Variable findAzimuthVariable() const;
  const Variable findCenteredDepthVariable() const;
  const Variable findHeightVariable() const;
  const Variable findQualityVariable() const;
  const Variable findTypeVariable() const;
  const Variable findColorVariable() const;
  const Variable findShapeVariable() const;
  const Variable findInfoArrayVariable() const;
};
```

The **DipModel** domain object allows you to access dip values (a **Dip** object):

- get a **Dip** at a given row index of the dip dataset
- set a **Dip** at a given row index of the dip dataset, replacing the previous dip values at the row index
- get or set a QVector<**Dip**> equal to the size of the dip dataset
- insert a **Dip** at a depth value; the depth value is expressed with the same unit as the dataset reference
- remove a **Dip** at a given index

See the "Dip object" section on page 2-10 for more information on dip values.

From the **DipModel** domain object, retrieve the **Dip** variable and its associated dip picking variables saved in the dip dataset. If the variable does not exist, the Ocean **find** pattern returns a null **Variable** object.

- **dipVariable** – The true dip variable (cannot be null).
- **findAzimuthVariable** - The true dip azimuth associated with the true dip variable.

- **`findTypeVariable`** - The dip type associated with the true dip variable. A color and a shape variable are associated with it. With these variables, Techlog determines the display properties for each dip type.
- **`findColorVariable`** – The color variable stores the display color for each dip. When a dip variable is dragged into a layout or other interpretation plot, Techlog uses the type variable association to determine the type of each dip and uses the color variable associated with the type variable to determine the display color.
- **`findShapeVariable`** – The shape variable stores the display symbol for each dip. Techlog uses the shape variable associated with the type variable to determine the display symbol.
- **`findQualityVariable`** – The quality variable stores a quality value (scaled from 0-1) for each dip.
- **`findCenteredDepthVariable`** – The centered depth variable stores the depth where the feature would cross the center of the borehole for each dip.
- **`findHeightVariable`** – The height variable stores the height along the hole of the sine wave in the borehole wall for each dip.
- **`findInfoArrayVariable`** – The info array variable stores an array of data needed by Techlog to use in redrawing partial sine waves, breakouts, and induced fractures on the images (only partial sine waves are currently exposed in Ocean).

The **`dipCategory`** property returns if dips handle by the **`DipModel`** are true or apparent dips.

Dip object

The **`Dip`** class represents the dips picked on the matrix array variable as full sinusoids (three points or more to define a sinusoid).

```
class Dip
{
public:
  Dip(const float dipValue);
  Dip(const float dipValue, const float azimuth);
  Dip(const float dipValue, const float azimuth, double
      height);
  Dip(const float dipValue, const float azimuth, double
      height, const float quality);
  Dip(const float dipValue, const float azimuth, double
      height, const float quality, const DipClassification
      &classification);
  float dipValue() const;
  float azimuth() const;
  double height() const;
  float quality() const;
  const DipClassification& classification() const;
  ...
};
```

Access the **Dip** property values using the corresponding getters from the **Dip** instance. You can only set the property values when the object is created.

This class is value semantic. It means that the value of each instance is duplicated after each assignment. **Dip** objects are accessible through the **DipModel** parent instance.

This example shows how to insert dips between two dips selected with the Techlog interactive selection mode turned on.

```
void activity::onDipSelectionChanged(const
Slb::Ocean::Techlog::DipSelectionChangedArgs &args)
{
  DipTrackItem dipTrackItem = args.sender();
  Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock,
  dipTrackItem);
  DipModel dipModel = dipTrackItem.findDipModel();
  QVector<quint64> selectedIndexes =
  dipTrackItem.selectedDipIndexes();
  // ...
  lock.release();

  if (selectedIndexes.count() == 2)
  {
    lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock, dipModel);
    Variable dipVariable = dipModel.dipVariable();
    Dataset dipDataset = dipVariable.dataset();
    Variable ref = dipDataset.findReferenceVariable();
    double topDepth, bottomDepth;
    if (ref.getFloatValue(selectedIndexes.at(0)) <
    ref.getFloatValue(selectedIndexes.at(1)))
    {
      topDepth = ref.getFloatValue(selectedIndexes.at(0));
      bottomDepth = ref.getFloatValue(selectedIndexes.at(1));
    }
    else
    {
      topDepth = ref.getFloatValue(selectedIndexes.at(1));
      bottomDepth = ref.getFloatValue(selectedIndexes.at(0));
    }
    double samplingRate = (bottomDepth - topDepth)/5;
    Dip dip = dipModel.getDip(selectedIndexes.at(0));
    for (int i = 1; i < 5; i++)
    {
      float dipValue = dip.dipValue();
      if (dipValue + i*10 <= 90)
        dipValue = dipValue + i*10;
```

```
      float azimuth = dip.azimuth();
      if (azimuth + i*45 <= 360)
        azimuth = azimuth + i*45;

      QString dipType = dip.classification().getType();
      QColor dipColor = dip.classification().getColor();
      DipShape dipShape = dip.classification().dipShape();
      switch (i)
      {
        case 1 :
          dipType = "MyType1";
          dipColor = Qt::green;
          dipShape = DipShapeHexagon;
          break;
        case 2 :
          dipType = "MyType2";
          dipColor = Qt::red;
          dipShape = DipShapeTriangle;
          break;
        case 3 :
          dipType = "MyType3";
          dipColor = Qt::black;
          dipShape = DipShapeDiamond;
          break;
        case 4 :
          dipType = "MyType4";
          dipColor = Qt::darkMagenta;
          dipShape = DipShapeStar;
          break;
      }

      DipClassification dipClassification (dipType, dipColor,
      dipShape);

      Dip newDip (dipValue, azimuth, dip.height(),
dip.quality(),
      dipClassification);

      dipModel.insertDip(topDepth + (i*samplingRate), newDip);
    }
    lock.release();
  }
}
```

This screenshot shows the result of the dips created in the **`DipTrackItem`** with different inclination, azimuth, type, color and shape values.

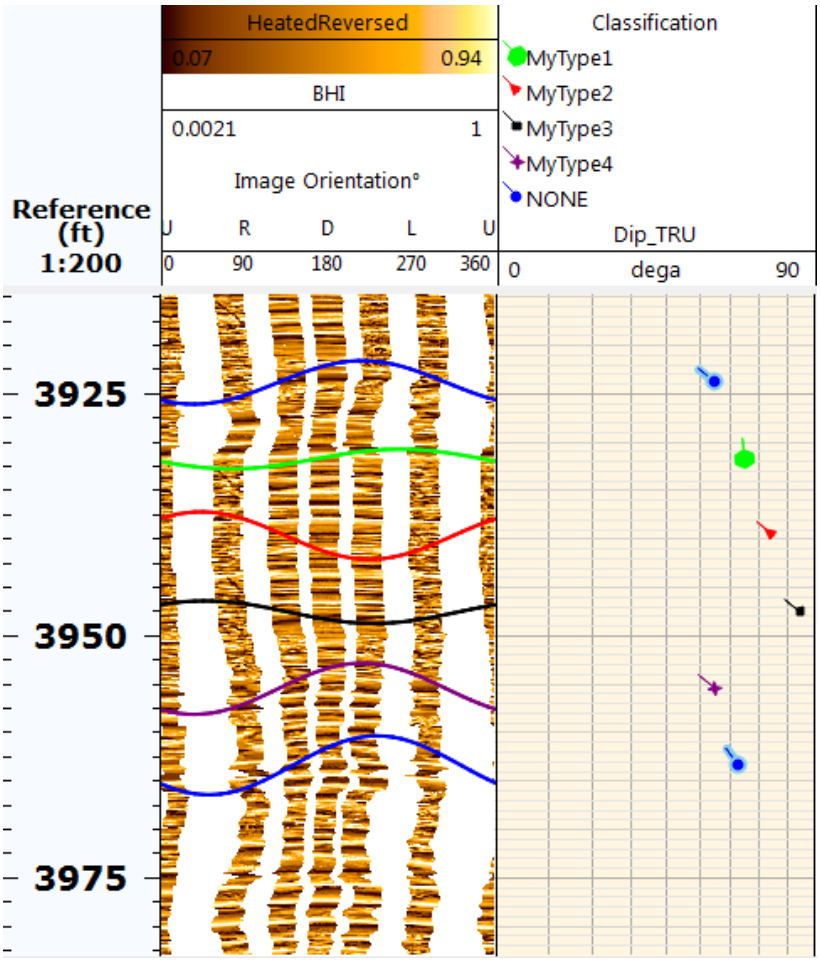


Figure 2-7 Dips inserted to the DipModel

The dip data passed to the **`Dip`** constructor is stored in the corresponding dip variable and its associated variables when the dip dataset is saved at the end of the dip picking session.

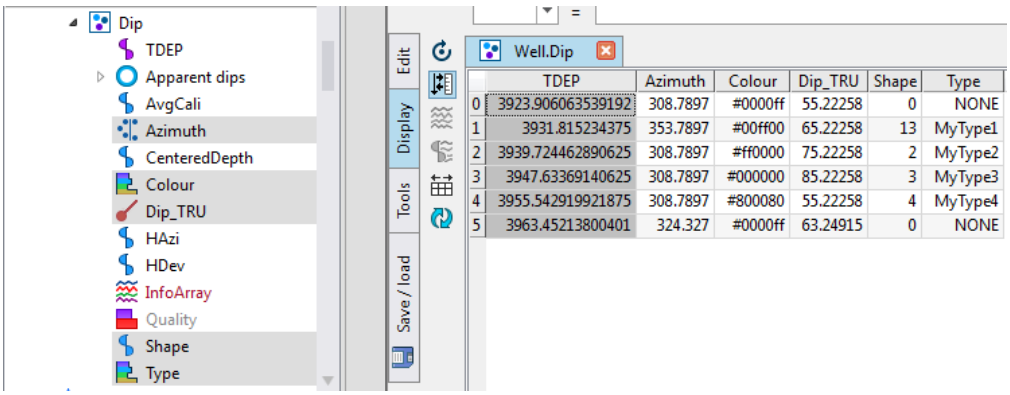


Figure 2-8 Dip dataset

See "DipModel domain object" on page 2-8 for more information on how to retrieve the dip variable and its associated variables.

PartialDip object

For the planar features covering only part of the borehole, you may pick dips with partial sinusoids represented by the **`PartialDip`** class.

In Techlog, the partial sinusoids are validated, edited and saved in the exact same way as full sinusoids. Therefore, the **`PartialDip`** class derives from the **`Dip`** class.

```
class PartialDip : public Dip
{
public:
  PartialDip(const float dipValue);
  PartialDip(const float dipValue, const float azimuth);
  PartialDip(const float dipValue, const float azimuth, const
    QList<PartialDipSegment> &segments);
  PartialDip(const float dipValue, const float azimuth, const
    QList<PartialDipSegment> &segments, double height);
  PartialDip(const float dipValue, const float azimuth, const
    QList<PartialDipSegment> &segments, double height, const
    double centeredDepth);
  PartialDip(const float dipValue, const float azimuth, const
    QList<PartialDipSegment> &segments, double height, const
    double centeredDepth, const float quality);
  PartialDip(const float dipValue, const float azimuth, const
    QList<PartialDipSegment> &segments, double height, const
    double centeredDepth, const float quality, const
    DipClassification &classification);
  double centeredDepth() const;
  const QList<PartialDipSegment>& segments() const;
};
```

With the partial sinusoid picking mode, you pick dip features using between 1 and 3 individual segments. The segments drawn on the image represent the segments on the sinusoid of the final dip rather than rigorously following the points picked in each of the individual segments.

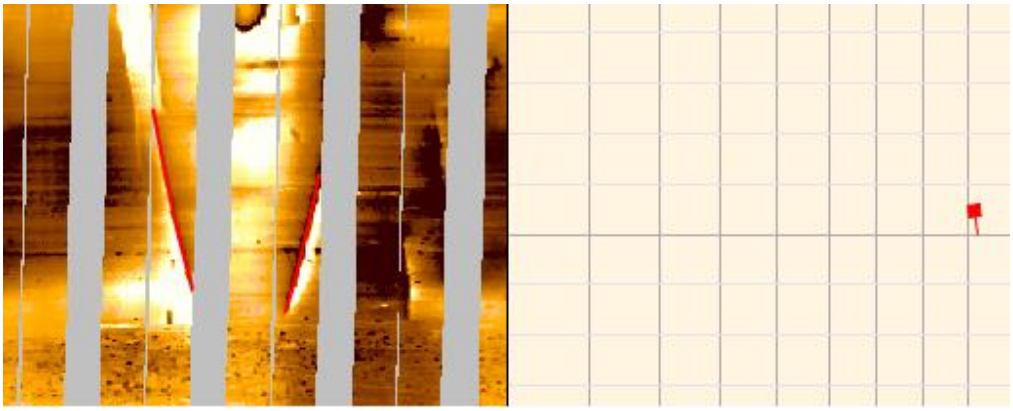


Figure 2-9 Partial sinusoid

Get the dip value and associated values like azimuth, height, quality, type, shape and color using members of the **`Dip`** base class.

See the "Dip object" section on page 2-10 for more information on **`Dip`** base class.

Specific members of **`PartialDip`** class are **`centeredDepth`** and a list of **`PartialDipSegments`** used to draw the partial sinusoid on the borehole image array.

- centered depth values are stored in the **`CenteredDepth`** variable of the dip dataset
  - For partial features, this depth is different from the depth at which the dip is recorded (taken at the center of the feature)

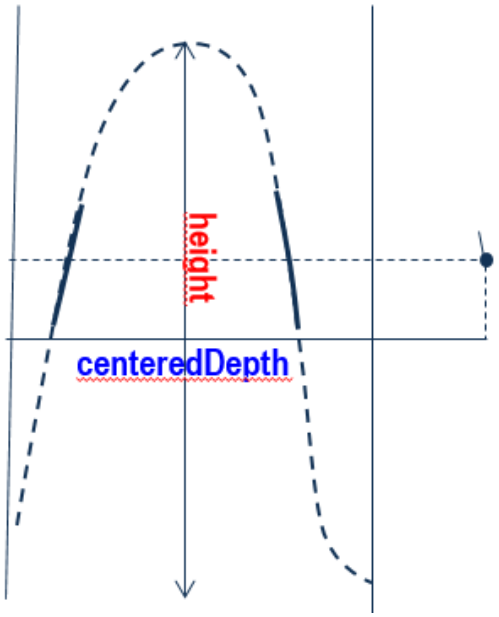


Figure 2-10 Partial sinusoid centered depth

- points composing the partial dip segments are stored in the **`InfoArray`** variable of the dip dataset as follows:
  - Column 1: start point of first segment of partial sinusoid
  - Column 2: end point of first segment of partial sinusoid
  - Column 11: start point of second segment of partial sinusoid
  - Column 12: end point of second segment of partial sinusoid
  - Column 13: start point of third segment of partial sinusoid
  - Column 14: end point of third segment of partial sinusoid

The **`PartialDipSegment`** class handles segment point values of the partial dip.

```
class PartialDipSegment
{
public:
  PartialDipSegment(const float begin, const float end);
  float begin() const;
```

```
  void setBegin(const float begin);
  float end() const;
  void setEnd(const float end);
};
```

This example shows how to insert partial dips between two dips selected with the Techlog interactive selection mode turned on.

```
void activity::onPartialDipSelectionChanged(const
Slb::Ocean::Techlog::DipSelectionChangedArgs &args)
{
  DipTrackItem dipTrackItem = args.sender();
  Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock,
  dipTrackItem);
  QVector<quint64> selectedIndexes =
  dipTrackItem.selectedDipIndexes();
  lock.release();

  if (selectedIndexes.count() == 2)
  {
    lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock,
    dipTrackItem);
    DipModel dipModel = dipTrackItem.findDipModel();
    lock.release();

    lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock, dipModel);
    Variable dipVariable = dipModel.dipVariable();
    Dataset dipDataset = dipVariable.dataset();
    Variable ref = dipDataset.findReferenceVariable();
    double topDepth, bottomDepth;
    if (ref.getFloatValue(selectedIndexes.at(0)) <
    ref.getFloatValue(selectedIndexes.at(1)))
    {
      topDepth = ref.getFloatValue(selectedIndexes.at(0));
      bottomDepth = ref.getFloatValue(selectedIndexes.at(1));
    }
    else
    {
      topDepth = ref.getFloatValue(selectedIndexes.at(1));
      bottomDepth = ref.getFloatValue(selectedIndexes.at(0));
    }
    double samplingRate = (bottomDepth - topDepth) / 5;
    Dip dip = dipModel.getDip(selectedIndexes.at(0));

    if (dip.isA<PartialDip>())
    {
      PartialDip partialDip = dip.cast<PartialDip>();
```

```
      for (int i = 1; i < 5; i++)
      {
        float dipValue = partialDip.dipValue();
        if (dipValue + i * 10 <= 90)
          dipValue = dipValue + i * 10;
        float azimuth = partialDip.azimuth();
        if (azimuth + i * 45 <= 360)
          azimuth = azimuth + i * 45;

        PartialDip newPartialDip(dipValue, azimuth,
        partialDip.segments());
        dipModel.insertDip(topDepth + (i*samplingRate),
        newPartialDip);
      }
    }
    lock.release();
  }
}
```

DipClassification object

You classify dips with predefined types, and create or edit new custom types.

1. In Techlog, click **Geology > Dips handling > Dip classification** to open the **Dip classification** window.
2. Select a type and apply the selected type as the current dip classification.

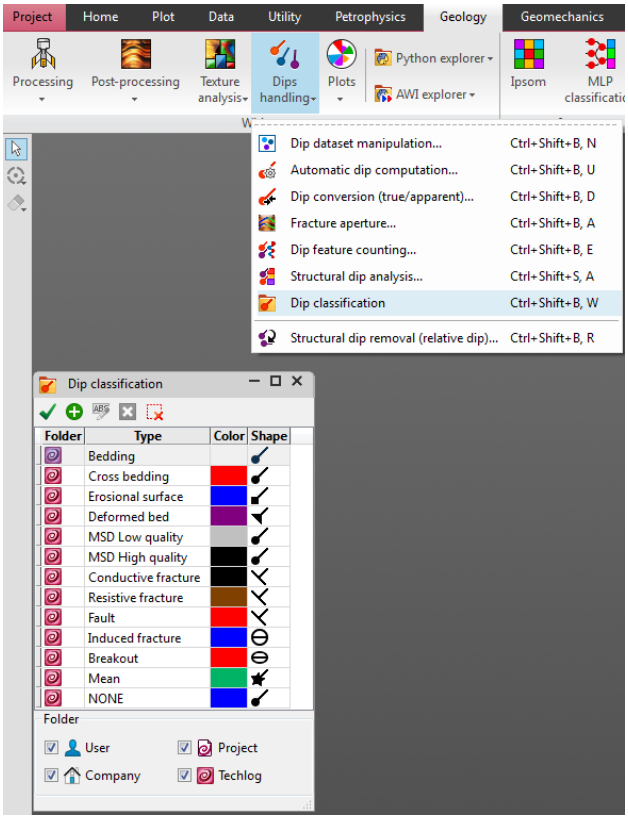


Figure 2-11 Dip classification window

Retrieve the current dip classification from the **`Workspace`** domain object.

```
class Workspace : public DomainObject
{
public:
  ...
  DipClassification currentDipClassification() const;
};
```

In Ocean, you may also create a new type associated with a color and a shape using the **`DipClassification`** constructor.

```
class DipClassification
{
public:
  DipClassification(const QString type, const QColor color,
     const DipShape shape);
  const QString& getType() const;
  void setType(const QString &type);
  const QColor& getColor() const;
  void setColor(const QColor &color);
  const DipShape& dipShape() const;
  void setShape(const DipShape &shape);
};
```

Modify property values of an existing **`DipClassification`** object using the setters. The **`DipClassification`** object is passed as an argument to the **`Dip`** and **`PartialDip`** constructors.

See the "Dip object" section on page 2-10 for more information on how to create a new **`DipClassification`** and pass it to the **`Dip`** constructsor.

This example shows how to retrieve the current dip selection, change its property values and apply the changes by interactive selection to a dip that has been picked on the borehole image.

```
void activity::onDipSelectChanged(const
Slb::Ocean::Techlog::DipSelectionChangedArgs &args)
{
  DipTrackItem dipTrackItem = args.sender();
  Lock lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock,
  dipTrackItem);
  QVector<quint64> selectedIndexes =
  dipTrackItem.selectedDipIndexes();
  lock.release();

  if (selectedIndexes.count() == 1)
  {
    lock = LOCK_CREATE_AND_ACQUIRE_ALL(lock);

    Workspace workspace =
Session::current().currentWorkspace();
    DipClassification dipClassification =
    workspace.currentDipClassification();

    qWarning() << "Current dip classification selected type is
"
    << dipClassification.getType();
    lock.release();

    DipTrackItem dipTrackItem = args.sender();

    lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock,
    dipTrackItem);

    DipModel dipModel = dipTrackItem.findDipModel();
    // ...
    lock.release();

    lock = LOCK_CREATE_THEN_ACQUIRE_OR_RETURN(lock, dipModel);

    Dip dip = dipModel.getDip(selectedIndexes.at(0));

    dipClassification.setColor(Qt::red);
```

```
    dipClassification.setShape(DipShapeEmptyTriangle);

    Dip newDip (dip.dipValue(), dip.azimuth(), dip.height(),
    dip.quality(), dipClassification);

    dipModel.setDip(selectedIndexes.at(0), newDip);

    lock.release();
  }
}
```

In this screenshot, when the dip is selected in the **`DipTrackItem`**, the current "Deformed bed" **`DipClassification`** type has its color and shape changed to red and **`DipShapeEmptyTriangle`** values respectively.

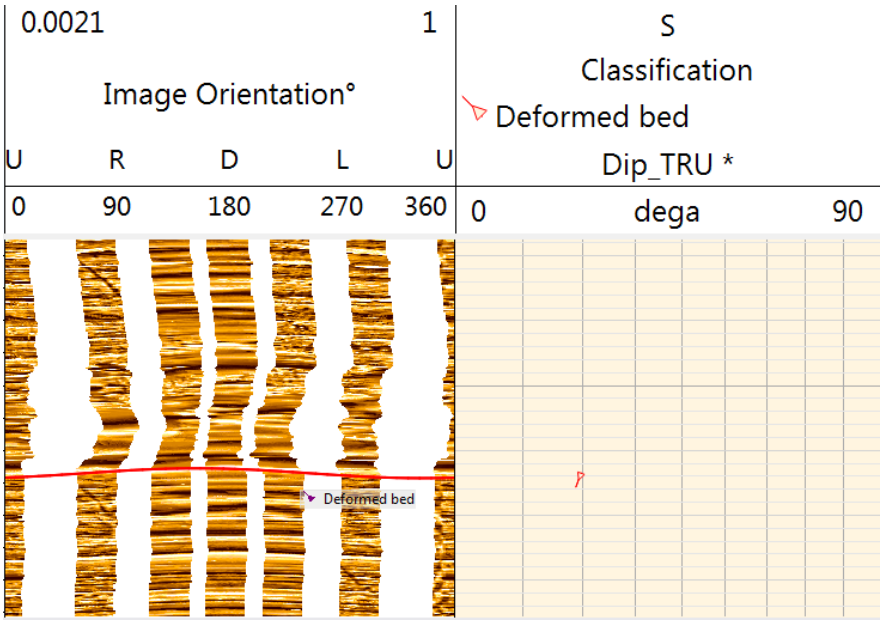


Figure 2-12 Current dip classification properties changed