Chapter 1. User Oriented Documentation

Introduction

Purpose

Softree Optimal is a technology calculating and minimizing the costs of corridor or alignment-based infrastructure projects such as roads, railways, pipelines etc. *Softree Optimal API* is a programming interface to Softree Optimal allowing:

- Costs calculation for a given alignment including optimal haul calculations.
- Feasibility checking.
- Vertical alignment optimization.
- Horizontal alignment optimization.

Languages

Softree Optimal API is currently only available on MS Windows platforms. It has been implemented with both C++ and standard C bindings. Examples are included for implementation in C++, C# and VB .NET

Licensing

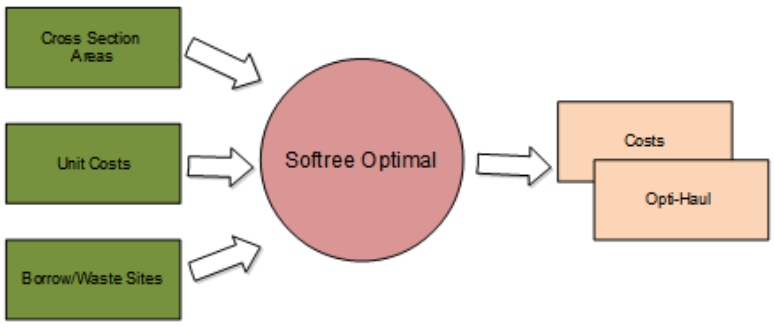
Softree Optimal API is available through a proprietary software license described in the following link:

https://support.softree.com/knowledge-base/getting-started/software-license-agreements

Functions Overview

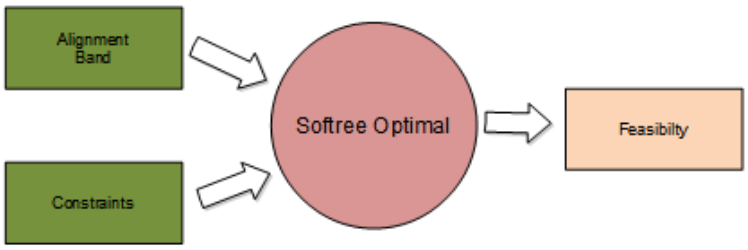
Costing

Softree Optimal provides functions for calculating excavation, embankment, movement, and other costs. Movement costs calculation include automatic determination of the optimum material movement (Opti-Haul).



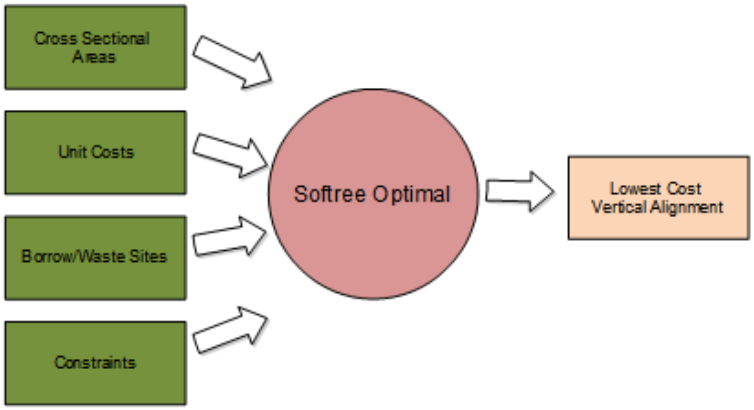
Feasibility Checking

Softree Optimal provides functions for checking the feasibility of a set of constraints (K value, grades max. min, etc.) with respect to a vertical alignment band.



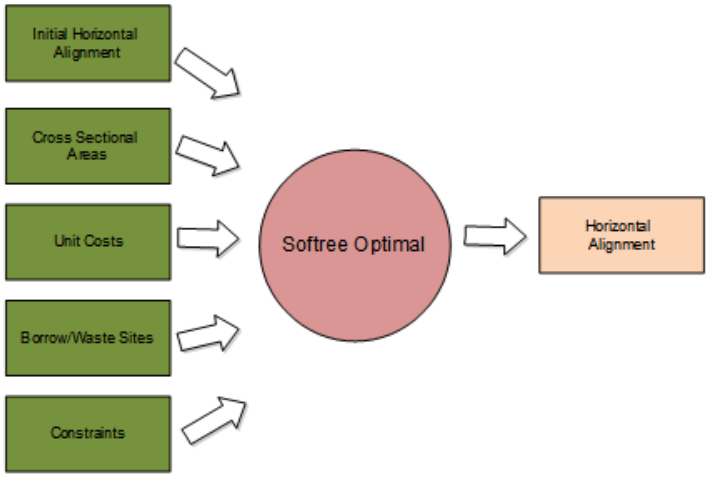
Vertical Optimization

Softree Optimal provides functions for determining the lowest cost vertical alignment based on cross sectional areas, unit costs, borrow/waste sites and constraints.



Horizontal Optimization

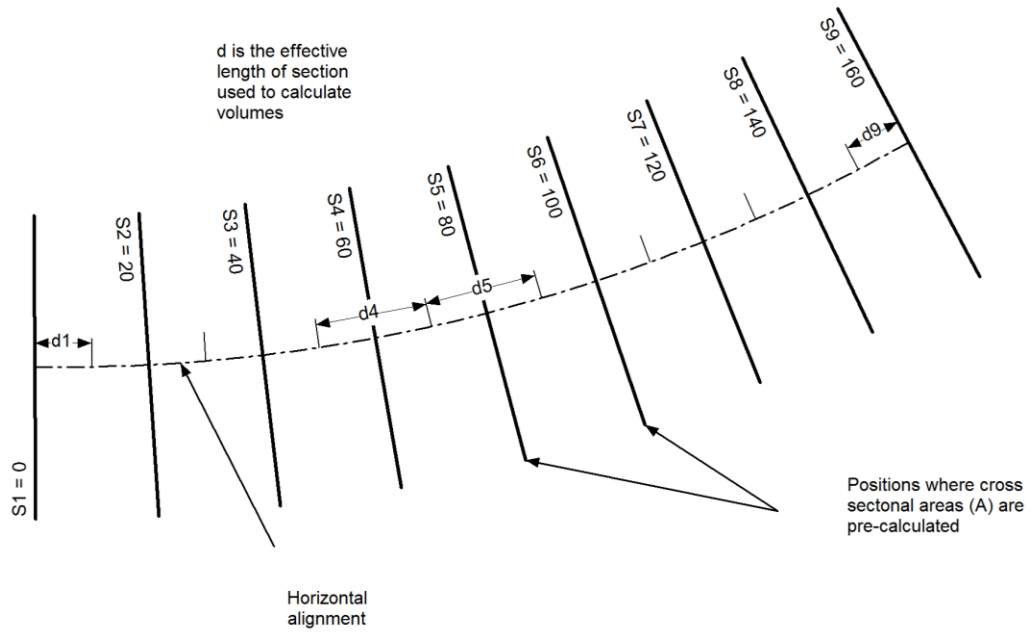
Softree Optimal provides functions to search for the optimal horizontal vertical alignment based on an initial horizontal alignment, cross sectional areas, unit costs, borrow/waste sites and constraints.



Concepts

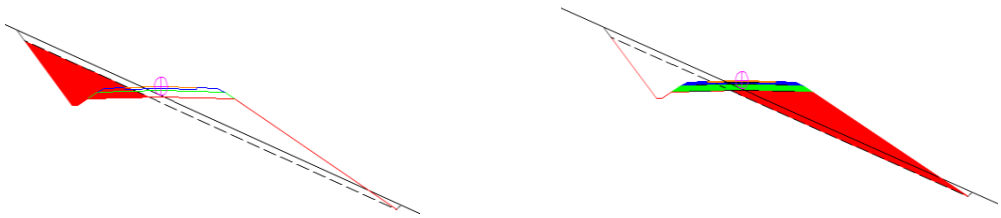
Cross Sectional Model

Softree Optimal calculations are based on a *Cross Section Model*. A cross section model consists of a baseline horizontal alignment and a collection of perpendicular sections.



Cross Section Model

Cut and fill costs are determined by the cross sectional areas (calculated and saved on each section) and their corresponding unit cost for each material. Sections are be referred to by station range.



Cut and Fill Areas

Materials are used to track costs and allowable movements. Each section contains a list of materials, and the corresponding cut and fill areas. For example:

- OB Cut , 120 Sq. M. (overburden cut)
- SR Cut, 54 Sq. M. (solid rock cut)
- Q2, Fill, 111 Sq. M. (general fill q2 or higher)
- ASH Fill, 1.6 Sq. M. (asphalt surfacing)

For costing and optimization, a table of materials is required. Each material in the table has the following fields:

- *Identifier* - A unique 2 character identifier. E.g. OB
- *Description* - A unique 50 character description. E.g. Overburden
- *Excavation Unit Cost* - Each material has a designated *Excavation Cost* (in $/Cu. M. or $/CU. Yd. depending on project settings).
- *Embankment Unit Cost* - Each material has a designated *Embankment Cost* (in $/Cu. M. or $/CU. Yd. depending on project settings).
- Quality - Each material has a designated *Quality* factor (Q1, Q2, Q3, Q4). The Quality factor indicates the usefulness of a material for fill. The *Quality* indicator can is used to control fill operations. When fill material of a given quality is required, any material with the same or higher quality can be used as fill. Q1 is the best and Q4 is the worst.

Movement Costs

Movement costs are divided into 3 categories (Freehaul, Overhaul, and Endhaul). The distance for each type of haul depends on the *Hauling* and *Loading* costs. Press the *Haul costs* button to change these costs and update the Freehaul and Overhaul threshold distances. Movement cost units are $ per (Cu. m. x km) or $ per (Cu. Yd. x mile).

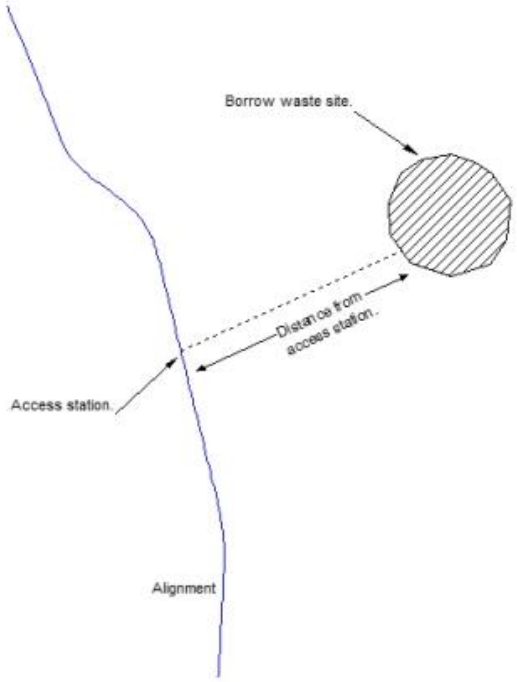
Note: The movement costs are common to all material types.

Material Sources and Balancing

To calculate movement costs the Softree Optimal assumes that all material quantities must be balanced (materials can not 'magically' appear or disappear). Materials can either come from the corridor (*Cross Section Model)* or from external *Borrow /Waste Sites*.

Borrow/Waste Sites

Borrow or waste sites can be included in the optimization to account for external material sources. The following information can be attributes to each site:



- *Borrow or Waste* – indicates if the type of pit.
- *Smart Pit Indicator* – if set the pit is only used if it is economic (as determined by Softree Optimal).
- *Access station* - station on the alignment from which the pit is accessed.
- *Access Distance* - from access station to the borrow/waste site (sometimes called *dead-haul* distance).
- *Comment* – for tracking purposes (optional).
- *Waste quality* – The minimum material quality required (non-variable only).
- *Material* - available (borrow pit only).
- *Excavation $* - Cost to excavate (borrow pit only).
- *Capacity limit* - Maximum volume of borrow or waste (*variable* only).
- *Volume* - Exact amount of borrow or waste (non-*variable* only).

Constraints

Constraints are used to control or restrict an alignment.

Vertical Curves and Grade Constraints

These constraints control the smoothness and steepness of the alignment.

Definitions:

K Value represents the horizontal distance along which a 1% change in grade occurs on the vertical curve. It expresses the abruptness of the grade change in a single value. Speed tables or other design tools often provide a target minimum K value.

Maximum grade break (%) - Is the change in slope % between to consecutive tangents.

Min. curve length – is the minimum length of a curve measured horizontally (in station dimension) from the BVC to the EVC.

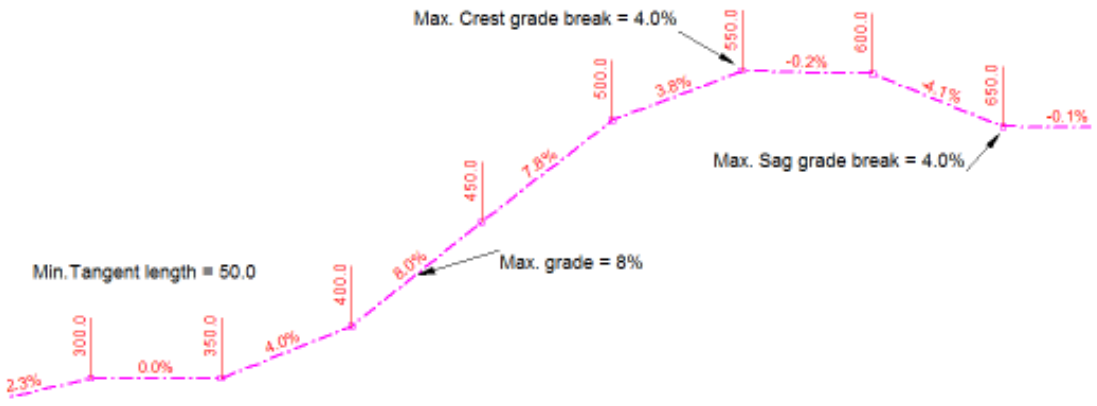
Min. tangent length is the minimum length between curves measured horizontally (in station dimension) from the EVC of a curve to the BVC of the next curve.

Base length represents the smallest segment length that will be used in the alignment optimization; curves and tangents will be a multiple of this length. It must be less than or equal to the minimum curve length and the minimum tangent length. Its value is controlled using the slider *Accuracy* or by entering a value between 1 and 5 in the edit box to the right of the slider. A smaller base length will give better results but the resolution time may increase significantly.

Note: the base length should be greater than the spacing between stations, otherwise, unnecessary curves may appear.

Several types of vertical curves constraints are available. Each type provides individual control over curvature (K), grades, tangent and curve lengths.

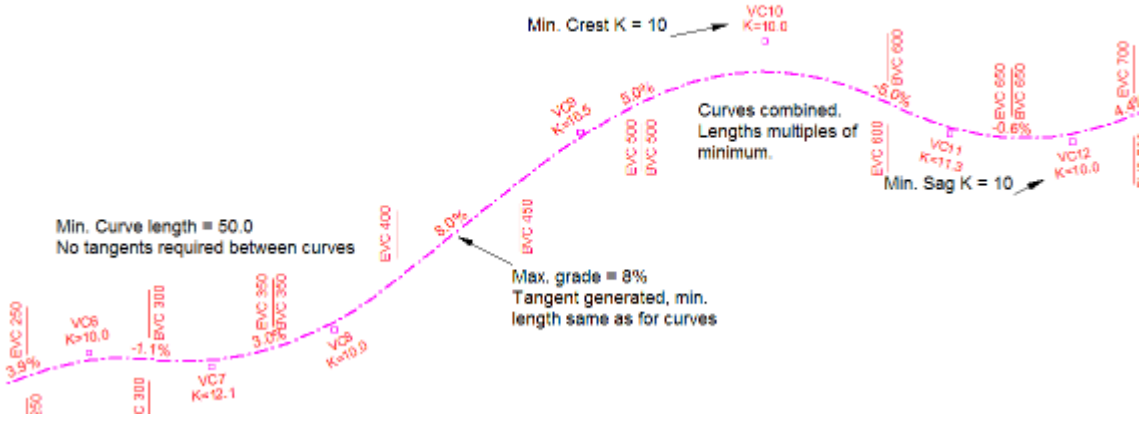
PolyLine (OPT_POLYLINE)
This will generate an alignment with straight segments only (no curves).



In *Polyline* mode a *Minimum Tangent Length*, *Min. / Max. Grades* and *Max. Grade Break* (maximum change in grade) can be set for both Sag and Crest curves.

Curves [Fast] (OPT_CURVE)

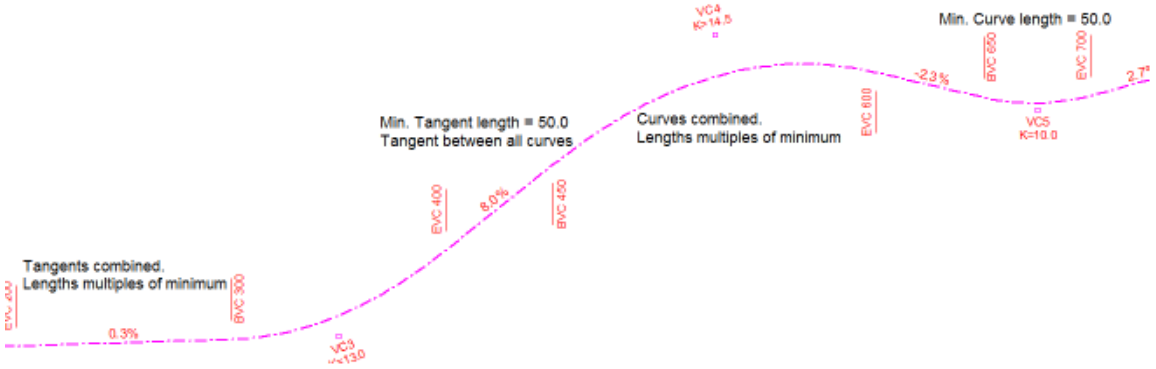
Sometimes called back to back curves, this option will generate an alignment with curves where tangents are not required between curves.



In *Curve [Fast]* mode a *Minimum Curve Length*, *Min. / Max. Grades* and *K value* can be set for both Sag and Crest curves.

Curves and Tangents [Slow] (OPT_CURVE_TAN)

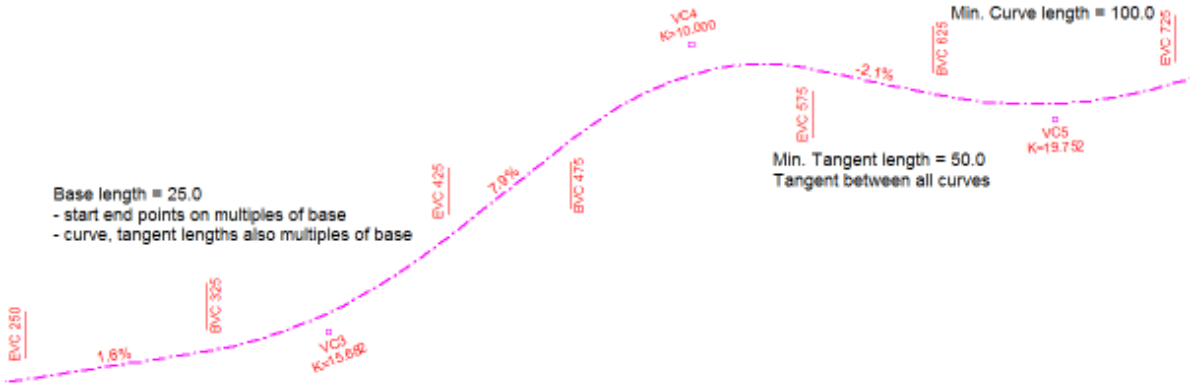
Generate an alignment where a tangent is inserted between curves. With this choice, the resolution time can be significantly slower.



In *Curves and Tangents [Slow]* mode a *Minimum Curve Length*, *Minimum Tangent Length*, *Min. / Max Grades* and *K value* can be set for both Sag and Crest curves.

Variable Curves and Tangents [Slowest] (OPT_CURVE_TAN_GENERAL)

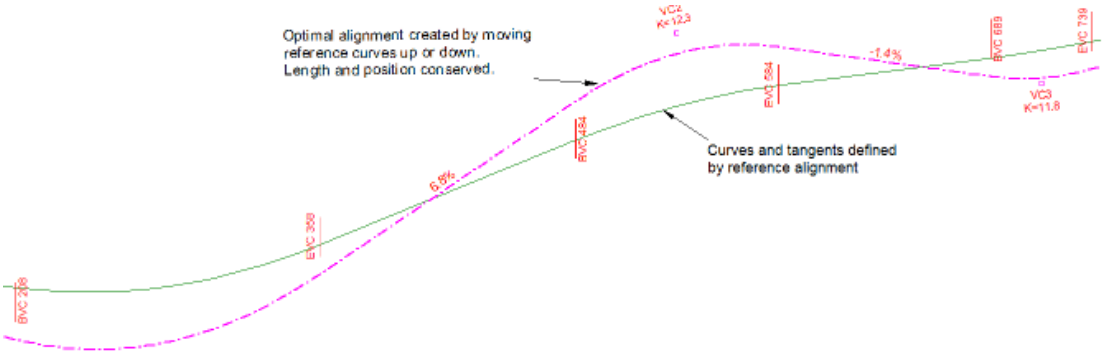
Similar to *Curves and tangents [Slow]* but curve, tangent and base length can vary. With this choice, the resolution time is the slowest.



In *Variable Curves and Tangents [Slowest]* mode a *Minimum Curve Length*, *Minimum Tangent Length*, *Min. / Max Grades* and *K value* can be set for both Sag and Crest curves.

User Defined (USER_DEFINEDCURVE)

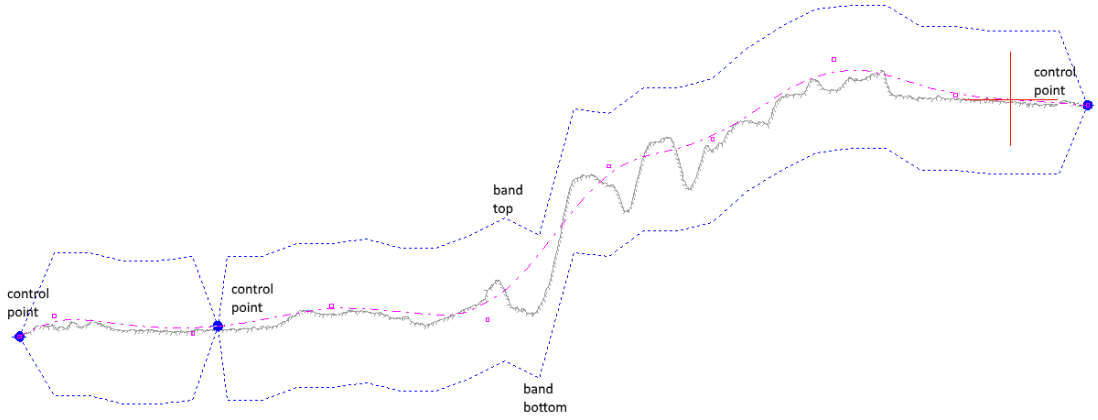
Preserves the location and length of the curves and tangents as defined in the *Reference alignment*. Similar in speed to *Curves [Fast]*.



In *User Defined* mode a *Min. / Max. Grades* and *K value* can be set for both Sag and Crest curves.

Vertical Band

The Vertical Band is the allowable region for vertical optimization. The band is controlled by *vertical offsets* and *control points*.



Chapter 2. Programming with Softree Optimal API (SO API)

Background

Softree Optimal API (hereafter referred to as ***SO API***) is a library of functions for costing, feasibility checking and alignment optimizing of corridor based civil engineering projects. It is provided as a set of DLLs, libs and header files, and currently runs in the Microsoft Windows environment, It is compatible with C#, VB or C++ MFC based applications.

Softree Optimal API was initially developed by Softree for use within the RoadEng road and site design software, using C++. An additional C-style API interface was built around the C++ API providing portability across MS languages such VB and C#.

Assumptions and Requirements

This documentation assumes that the reader has a working knowledge of software development using one or more of the aforementioned programming languages.

This document assumes that the programmer has an installed licensed copy of Microsoft Visual Studio (version 2013 or newer. v2013 is recommended since this was the version used to develop the library).

The programmer should also have a licensed copy of RoadEng to generate the required CSV file that SO API uses to load road area details.

This document also assumes the programmer has a basic understanding of civil engineering terminology.

Key notes about the Softree optimization API

A. For those unfamiliar with cross language development using *.Net, here are some important gotchas.

Strings are handled quite differently in the C/C++ world versus the C#/VB world. We use some programming tricks to send and retrieve strings between those two worlds.

In C/C++, a string is just an array of char and can be modified freely (in 1998, C++ introduced the ANSI standard std::string for string operations, but since this API uses a C-style interface, we use arrays of char, and arrays of wide-char).

However, C#/VB considers strings immutable. Once set, strings cannot be altered without causing memory corruption.

This is an issue since we need the Softree C/C++ DLL to populate strings and return them to .Net code

So, to retrieve string values from a C/C++ DLL, the .Net code should:

- Use a StringBuilder and pass a stringbuilder object to the DLL, if retrieving a string value as a sole parameter.
- Use an IntPtr, *allocated and populated* with a string of empty spaces, if retrieving a string as part of a data structure. The IntPtr must be *deallocated* explicitly by the programmer

Why the different approaches, one with strings as a single parameter, and one for strings in structures? Because C# and VB structures cannot include StringBuilders as data members. So, for the C/C++ DLL to populate a string within a structure, and return it to .Net code, we use this approach.

In .Net, an IntPtr is basically a void pointer and it must these be explicitly allocated AND released, in unmanaged memory, by the programmer, because .Net code does not garbage-collect memory that is allocated in unmanaged memory.

B. All Softree API functions are prefixed

SOptXXXXXXXXXXXX()

C. Most functions, that return a value, will return a 0 to indicate success, and a positive number to indicate an error.

You can retrieve descriptive error details with the following function call

bool SOptGetErrorDetail(long lErrorCode, wchar_t* buffer, int bufferLength)

where the lErrorode parameter is the error code returned in the last **SOpt**XXXXXXXXXXXX function call, the buffer is a pointer to a c-style array of wide chars, and bufferLength is the number of characters in the buffer. Upon return, buffer will contain a description of the error.

D. Most functions take an iOptimizerId as the last parameter. This is a 32-bit integer identifying the specific instance of the optimizer the programmer wants to configure or execute. In the initial release of SO API, the only valid value for this parameter is 0. A future version of SO API may permit multiple instances of the optimizer.
E. The Softree API uses a number of Structures for passing information to the DLL and for retrieving information from the DLL. See the *Data Structures* for some of the details about these structures.
F. .Net programs expect that dependant dlls should be in either the local folder that holds the .exe, a subfolder of the local folder, or in the Global Assembly Cache (GAC), or in the system PATH. SO API installer will request the RoadEng\bin install folder and add that to the system PATH.

Demo programs

We've provided three demo programs that illustrate how to use SO API. One in each C++, C# and VB. C# and VB were the target languages when Softree began writing this version of SO API, and those demo programs are more detailed than the C++ demo. The C++ demo is functional (it lacks the graphing feature of the .Net demos) but was built as a testing and debugging harness for SO API.

If you use the C++ demo, you will need adjust the include directories and library directories to point to where ever you've installed your SO API files.

The C# demo and VB demo are functionally identical. They execute the same API calls, with the same parameters, in the same sequence. So, pick the language that you prefer.

Since they are so similar, this document will only illustrate the C# example, but the concepts and instructions should transfer over to VB fairly easily.

Key steps to setup a programming project that uses the Softree API:

1. Install the Softree Optimization External API package to your development computer.
2. Add the path to the RoadEng bin folder to the windows environment path, so that this program can find the required the SO_API.DLL as well as the RoadEng DLLs.
3. using visual studio, create a c# windows forms app
4. change build target to x64, remove "Any CPU" target ("Any CPU" is a .Net target. The Softree library was built for x86 and x64 platforms, any "Any CPU" has some compatibility issues).
5. copy the SOptStructs.cs and SOptConstants.cs files from the c# demo app, into your project folder
6. add the SOptStructs.cs and SOptConstants.cs files to your c# windows forms app project
7. add imported functions, either to the main form, or to a second code file that extends the partial class of the main form. You can use the Imports.cs file from the C# demo program as an example. This Imports.cs file is not comprehensive - It contains only the functions from the library that the demo program requires. If you chose to implement the other functions in the library, you will need to import those functions yourself.
8. create a driver function that will conduct the optimization run.
9. The first step, in the driver function, is to initialize the optimizer. This sets up and allocates the memory resources and internal objects required by the optimizer object.

```
INT ALIGNMENT = CONSTANTS.OPTIMIZATION_TYPE_VERTICAL;
INT OPTMODEL = CONSTANTS.FULLOPTIMIZATION;
INT COSTMODEL = CONSTANTS.GEN_COST;
INT ISIMPERIAL = 0;
RETCODE = SOPTINITIALIZE(ALIGNMENT, OPTMODEL, COSTMODEL, ISIMPERIAL, IOPTIMIZERID);
```

a. alignment is an integer, one of:
   - OPTIMIZATION_TYPE_HORIZONTAL - horizontal and vertical optimization
   - OPTIMIZATION_TYPE_VERTICAL - vertical optimization

b. optmodel is an integer, one of:
   - COSTOPTIMIZATION (0) - calculate cost of an alignment
   - FULLOPTIMIZATION (1)- optimize and determine lowest cost vertical alignment
   - FESOPTIMIZATION (2) - check if constraints are feasible
   - GENOPTIMIZATION (3) - generate a best fit profile

c. costModel is an integer, one of:
   - GEN_COST (0) - general cost (default)
   - SIMP_COST (1) – simplified cost model (**Deprecated**)
   - FOREST_COST (2) - forest cost model (**Deprecated**)
   - FES_COST(3) - for a feasibility check (no costing involved)
   - NOEARTH_COST (5) - cost model without earth movement

d. isImperial is an integer from the range [0,1]. 0 is for metric, 1 is for imperial

e. optimizerId is the enumerated id of the optimizer instance from [0....(maximum number of optimizers-1)]. This value defaults to zero.

10. configure the solver. The solver object is the mathematic engine that attempts to find the optimal road given the mathematic model. It has two available algorithms, the CBCSOLVER and the CPLEXSOLVER

```
INT SOLVERINDEX = CONSTANTS.CBCSOLVERINDEX;
RETCODE = SOPTPARAMSETSOLVER(SOLVERINDEX, IOPTIMIZERID);
```

a. solverIndex is an integer from the range 0 to 1 where 0 = CBCSOLVERINDEX, and 1 = CPLEXSOLVERINDEX

11. configure the cost model. Unless the optimization run is a feasibility check, the model needs costs for moving and loading materials over distances. That's what the SOptConfigureCostModel API call does

```
int useQuadratic = Constants.COSTMODEL_NETWORK;
CostModelStruct cms = new CostModelStruct(0);
cms.dMovCostFreehaul = 4.0;
cms.dMovCostOverhaul = 2.0;
cms.dMovCostEndhaul = 1.0;
cms.dLoadCostFreehaul = 0;
cms.dLoadCostOverhaul = 0.5;
cms.dLoadCostEndhaul = 2.5;
cms.dFreeHaulDistance = -1;
cms.dOverhaulDistance = -1;
retCode = SOptConfigureCostModel(cms, useQuadratic, iOptimizerId);
```

a. useQuadratic is an integer from the range 0 to 1 (0 is for network, 1 is for quadratic). 0-network should be used in most cases
b. cms is a data stucture (CostModelStruct) that contains the move and load costs, and haul distances of the model. See the CostModelStruct in the COptStructs.cs file or the COptStructs.h file for a more details about the members of the structure.

12. read the csv file into the station manager. The csv file, exported from RoadEng, contains all the information about the future road and the surrounding area, including the material/ground-types present.

```
OpenFileDialog theDialog = new OpenFileDialog();
theDialog.Title = "Open CSV File";
theDialog.Filter = "CSV files|*.csv";
theDialog.InitialDirectory = Environment.CurrentDirectory;
if (theDialog.ShowDialog() == DialogResult.OK)
{
 StringBuilder filePath = new StringBuilder(theDialog.FileName);
 retCode = SOptReadCSVFileIntoStationManager(filePath, iOptimizerId);
}
```

a. filePath is a fully qualified path to the file that the user wants to read. This file is exported from RoadEng and contains details about the road and area to be optimized. The filePath must be less than 256 characters long.

13. retrieve the names of required materials from station manager. After the csv file is loaded into the station manager, it contains the materials (aka ground types) in the area to be optimized. This function retrieves the names of those ground types. Use this information to populate the Materials manager with the costs (MaterialCosts) for excavating and filling this ground type.

```
int numMaterials = -1;
retCode = SOptGetNumMaterialsInStationManager(ref numMaterials, iOptimizerId);
IntPtr [] aMatNames = new IntPtr[numMaterials];
```

```
FOR( INT I = 0; I < NUMMATERIALS; I++)
{
  AMATNAMES[I] = MARSHAL.STRINGTOHGLOBALAUTO(CONSTANTS.BLANKBUFFER);
}
RETCODE = SOPTRETRIEVEMATERIALLISTFROMSTATIONMANAGER( AMATNAMES, NUMMATERIALS,
IOPTIMIZERID);
FOR (INT I = 0; I < NUMMATERIALS; I++)
{
 STRING SMATNAME = MARSHAL.PTRTOSTRINGAUTO(AMATNAMES[I]);
 // USE MATERIAL NAME LIST TO POPULATE THE MATERIAL MANAGER WITH THE MATERIAL COSTS
 MARSHAL.FREEHGLOBAL(AMATNAMES[I]);
}
```

a. numMaterials is a pointer to an integer. This is where the function returns the number of materials/ground-types
b. aMatNames is a pointer to an array of strings (within IntPtrs). the function populates this array with the materials from the station manager. *Note the calls to* `Marshal.StringToHGlobalAuto()`, `Marshal.PtrToStringAuto()` *and* `Marshal.FreeHGlobal()`. As mentioned in the key notes at the top of this document, this is the way .Net code retrieves string-based information from C/C++ DLLs. These marshal calls are allocating, retrieving, and releasing unmanaged memory.

14. add material costs to the materials manager. After using the `SOptRetrieveMaterialListFromStationManager()` function in the previous step, use the retrieved names of the material types to populate the materials manager with the costs associated with each material type.

```
MATERIALSTRUCT MS = NEW MATERIALSTRUCT("OB", "OVERBURDEN", 2.0, 4.0, 1, 1.0, 1.0, 1);
RETCODE = SOPTADDMATERIALCOST(MS, IOPTIMIZERID);
MS.RELEASE();
```

a. ms is a data structure (MaterialStruct) that contains the name, description, cost coefficients, cut and fill factor of the material (ground type) in question. See the MaterialStruct in the COptStructs.cs file or the COptStructs.h file or *Data Structures* for a more details about the members of this structure.

15. add pits. Now that the materials manager has the costs of moving and excavating the materials, add any pits that act as a local source for needed materials, or a local repository for excavated materials.

```
PitStruct pitStrct = new PitStruct("PitEx1", 0, 0, Constants.DEFAULTVALUE);
pit.dEmbankment = 12.0;
pit.dExcavation = 15.0;
pit.dBorrow = Constants.INFINITEVALUE;
pit.dWaste = Constants.INFINITEVALUE;
int options = 1;// add pit, capacity and cost
retCode = SOptAddPit(pitStrct, options, iOptimizerId);
```

a. pitStrct is a data structure containing the details of the pit to be added
b. options is an integer from the valid values of 0 or 1. 0= add pit and capacity, 1=add pit, capacity and cost

16. set the curve params for the spline manager. When the solver finds a solution for the optimization run, it will generate a set of cost results, and an optimal curve for the road. This function configures the characteristics of the curve.

```
int curveType = Constants.OPT_CURVE;
double exclusion = Constants.DEFAULTVALUE;
double gradeBreak = Constants.DEFAULTVALUE;
double mincurveLength = 20.0;
double mintangentLength = 0.0;
bool useCtrlPoints = false;
int curveAcc = 1;
retCode = SOptSetCurveParam(curveType, exclusion, gradeBreak, mincurveLength,
mintangentLength, useCtrlPoints, curveAcc, iOptimizerId);
```

a. curveType is an integer from the range [20..25] corresponding to the constants:
   - OPT_POLYLINE = poly line, no curve
   - OPT_CURVE = back to back curves, no tangents
   - OPT_CURVE_TAN = curve + tangent (curve length = tangent length
   - OPT_CURVE_TAN_GENERAL = variable curves (curve length = base n, tangent length = base m
   - USER_DEFINEDCURVE = user defined
   - OPT_SETFROMOUTSIDE = test mode, used for constraint checking

   the default value would be OPT_CURVE in most cases

b. exclusion is a double, represents the exclusion zone for straight segments
c. gradeBreak is a double, only used when curveType = USER_DEFINEDCURVE or OPT_SETFROMOUTSIDE
d. mincurveLength is a double, expressed as meters
e. mintangentLength is a double, expressed as meters
f. useCtrlPoints is a boolean, if true, use control points and range
g. curveAcc is an integer, only use when curveType = OPT_CURVE_TAN_GENERAL

17. set constraints. If needed, provide the optimizer object with any constraints you define.

```
GradeConstraintStruct gcs = new GradeConstraintStruct(0.0, -10.0, 10.0);
retCode = SOptAddGradeConstraint(gcs, iOptimizerId);

KConstraintStruct kCS = new KConstraintStruct(0.0, 20.0, 20.0);
retCode = SOptAddKConstraint(kCS, iOptimizerId);

PointConstraintStruct pcs = new PointConstraintStruct(0.0, 361.174,
Constants.DEFAULTVALUE, 0.0, 0.0, 0.0);
retCode = SOptAddPointConstraint(pcs, iOptimizerId);

int iHard = 1;
retCode = SOptKConstraintManagerSetHard(iHard, iOptimizerId);
```

a. gcs is a data struct that contains the details of the grade constraint
b. kCS is a data structure (KConstraintStruct) that contains station, kSag and kCrest values of the constraint
c. pcs is a data structure (PointConstraintStruct) that contains the position and elevation of a control point that the resulting optimized road needs to pass through.
d. iHard is an integer from the valid values of 0 to 1, where 0=not hard, 1=is hard.

See COptStructs.cs file or the COptStructs.h file for a more details about the members of the structures.

Note: hard constraints are used in several objects in the optimizer (in the pointconstraintmanager, the kconstraintmanager, and the stationconstraintmanager.) A hard constraint tells the solver algorithm that it must adhere to this constraint, if it cannot, the solver will evaluate as infeasible. A soft constraint will tell the solver to find a solution that is as close as possible to the constraint, but it will not automatically fail if it cannot meet the constraint.

18. validate the setup. This is a necessary step before the running the optimize algorithm. This function checks all the configuration settings and range of values to ensure that the algorithm has enough information to carry out the operation.

```
int optimizationAlignment = Constants.OPTIMIZATION_TYPE_VERTICAL;
retCode = SOptOptimizerValidateSetup(optimizationAlignment, iOptimizerId);
```

a. optimizationAlignment is an integer from the range 0 to 1, corresponding to 0=OPTIMIZATION_TYPE_HORIZONTAL and 1 = OPTIMIZATION_TYPE_VERTICAL. The initial release of the Softree optimization API **supports OPTIMIZATION_TYPE_VERTICAL only**. A subsequent release will add support for OPTIMIZATION_TYPE_HORIZONTAL.

19. optimize! This is SO API that starts the algorithm to find the optimal road.

```
OptimizeResultsStruct result = new OptimizeResultsStruct();
retCode = SOptOptimize(ref result, null, iOptimizerID);
```

   a. Result is a pointer to a data structure (OptimizeResultsStruct). The function populates this structure with the results from its calculations. See the COptStructs.cs file or the COptStructs.h file for a more details about the members of the structure.

At this point, SO API has calculated the cost of the optimal road, if it is feasible. An optimal curve is also available and can be retrieved for graphing.

20. Cleanup the optimizer. This releases the memory and resources used by the optimizer control

```
SOptRelease(iOptimizerID);
```

See the included CSharpWindowsFormsApp project for a complete working example of how to implement SO API, including a simple implementation of graphing the resulting curves.

Data Structures

The Softree API uses a number of Structures to send information to the library DLL, and to retrieve information from the library DLL.

These structures are defined the SOptStructs files (you will find 3 of them, SOptStructs.h, SOptStructs.cs and SOptStructs.vb, one in each of the corresponding C++, C# and VB languages).

A few notes about those structures.

Since we are using a C-style (non-object-oriented interface) for SO API, the structures were necessary, but where possible, these structures have at least one constructor (to initialize the member variables to default values), and some have a second constructor (with multiple parameters to initialize the data members to specific values during construction). *.Net code requires that custom constructors must have a parameter list, so in the SOptStructs.vb and SOPtStructs.cs files, those constructors may have a single parameter (int a = 0) that is not actually used.

Since *.Net requires some programming acrobatics for string handling in concert with C/C++ code, the some of structures in the SOptStructs.vb and SOPtStructs.cs files (ie. those structures that have strings within) have special methods for allocating and deallocating the memory for strings. They will have methods like SetName() or SetDescription() to allocate memory and assign a string value to that memory, and those structures will also have a Release() method that you must explicitly call to free the memory after you finish using the structure.

To illustrate one such structure, here is the MaterialsStruct as defined in the SOptStructs.cs file

```
public struct MaterialStruct
{
   public IntPtr sName;
   public IntPtr sDescription;
   public double dEmbankmentCostCoeff;
   public double dExcavationCostCoeff;
   public int iQuality;
   public double dCutFactor;
   public double dFillFactor;
   public int iIsTrueMaterial;
   public int iMaxStringLength;

         // detailed constructor, use this for Adding to the DLL
   public MaterialStruct(string _name, string _description, double _embankmentCostCoeff, double
_excavationCostCoeff, int _quality, double _cutFactor, double _fillFactor, int _isTrueMaterial)
   {
    sName = IntPtr.Zero;
    sDescription = IntPtr.Zero;
    dEmbankmentCostCoeff = _embankmentCostCoeff;
    dExcavationCostCoeff = _excavationCostCoeff;
    iQuality = _quality;
    dCutFactor = _cutFactor;
    dFillFactor = _fillFactor;
    iIsTrueMaterial = _isTrueMaterial;

    iMaxStringLength = Constants.MAX_STRING_LENGTH;
    SetName(_name);
    SetDescription(_description);
   }
   public MaterialStruct(int a = 0)//used for retrieving data from the DLL
   {
    sName = IntPtr.Zero;
    sDescription = IntPtr.Zero;
    dEmbankmentCostCoeff = 0;
    dExcavationCostCoeff = 0;
    iQuality = 1;
    dCutFactor = 0;
    dFillFactor = 0;
    iIsTrueMaterial = 0;
    iMaxStringLength = Constants.MAX_STRING_LENGTH;
    SetName(Constants.BLANKBUFFER);
    SetDescription(Constants.BLANKBUFFER);
   }
   public void SetName(string _name)
   {
    ReleaseName();
```

```
  int maxLength = Math.Min(iMaxStringLength, Constants.MAX_STRING_LENGTH);
  if (_name.Length < maxLength)
    sName = (IntPtr)Marshal.StringToHGlobalAuto(_name);
  else
    Debug.Assert(false, "input string too long: " + _name);
 }
 public string GetName()
 {
  if (sName != IntPtr.Zero)
  {
    return Marshal.PtrToStringAuto(sName);
  }
  else
  {
    return String.Empty;
  }
 }
 public void SetDescription(string _desc)
 {
  ReleaseDescription();
  int maxLength = Math.Min(iMaxStringLength, Constants.MAX_STRING_LENGTH);
  if(_desc.Length < maxLength)
    sDescription = (IntPtr)Marshal.StringToHGlobalAuto(_desc);
  else
    Debug.Assert(false, "input string too long: " + _desc);
 }
 public string GetDescription()
 {
  if(sDescription != IntPtr.Zero )
  {
    return Marshal.PtrToStringAuto(sDescription);
  }
  else
  {
    return String.Empty;
  }
 }
 private void ReleaseName()
 {
  if (sName != IntPtr.Zero)
  {
    Marshal.FreeHGlobal(sName);
    sName = IntPtr.Zero;
  }
 }
 private void ReleaseDescription()
 {
```

```
    IF (SDESCRIPTION != INTPTR.ZERO)
    {
      MARSHAL.FREEHGLOBAL(SDESCRIPTION);
      SDESCRIPTION = INTPTR.ZERO;
    }
  }
  PUBLIC VOID RELEASE()
  {
    RELEASENAME();
    RELEASEDESCRIPTION();
  }
}
```

Note that we are using ints instead of bools even when a bool might seem appropriate. That's because bools in C/C++ vs VB/C# are different sizes and different values. In C/C++ they are 8-bit char. In the Windows C API, BOOL is a 32-bit int. While in C#/VB, bools are 32-bit ints. So, to side step any potential conversion issues when sending structs containing bools across languages, we use 32-bit integers instead.

Notice that the sName and sDescription are declared as IntPtrs. This goes back to the Key note about sending string values between *.Net and C/C++. The structure has a SetName() method and a SetDescription() method that simplifies setting these members. There's also a GetName() and GetDescription() method for retrieving the string values from these members. Finally, there's a Release() method provided that frees up the memory for both of these string members. These Setters, Getters and Release methods are provided for your convenience and should be called within the *.Net code.

If you compare with the MaterialsStructure from the SOptStructs.h file, you'll notice it's much simpler. There are no getter or setter functions, no allocation or release.

C# Notes

A few notes about the CSharpWindowsFormsApp project.

It's separated into several key files:

SOptConstants.cs duplicates the SOptConstants.h file, as well as a handful of constants from the Optimal C++ code that are used in SO API.

SOptStructs.cs duplicates the SOptStructs.h file, and defines the structures used to pass sets of information to and from SO API.

Imports.cs is a partial class of the MainForm that centralizes all the functions imported from the DLL (the library that connects SO API to the underlying Softree optimization code). You can use this as a baseline and copy the DLLImports from this file into your own project

TestCases.cs is a partial class of the MainForm that centralizes the example optimization scenarios. This is primarily a set of examples that show how, and in what sequence, to call SO API functions to generate an alignment. In a real-world application, rather than write code from each scenario, using hard coded data, you'd probably write a more generalized application with the configuration information in external text files. Then application would read the configuration information, and use that information to populate and call API functions.

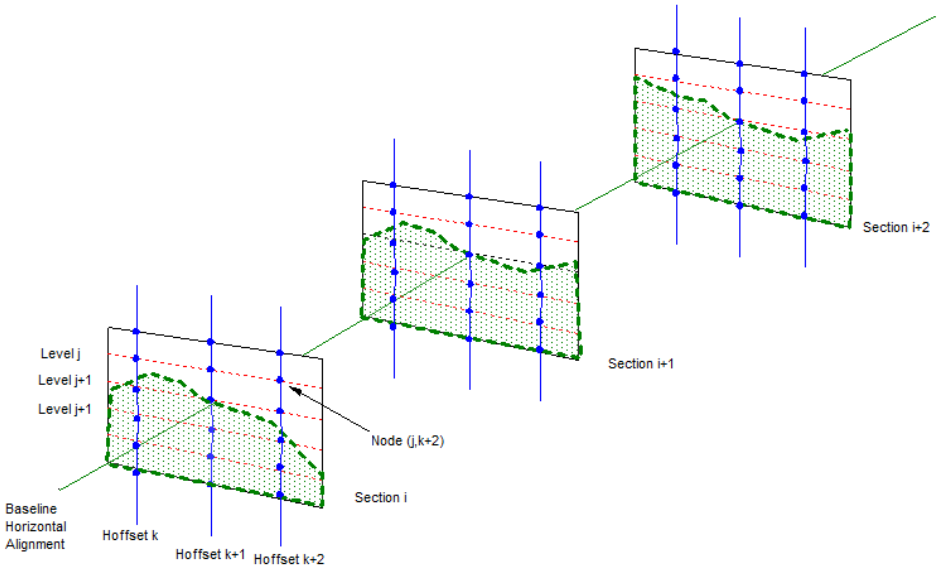
MainForm.cs is the UI code that handles the display and much of the event handling of the demo app.

Program.cs is auto-generated by Visual Studio.

Appendix 1. Softree Optimal® Areas CSV File Format

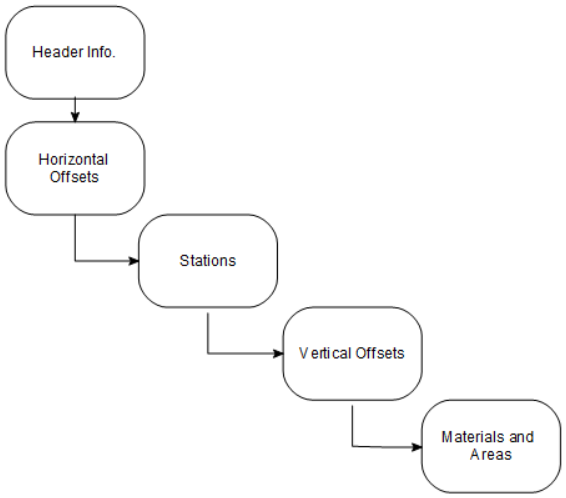
Introduction

The CSV file contains an array of areas and materials and/or pseudo materials (see below) saved by horizontal offset, station and level.



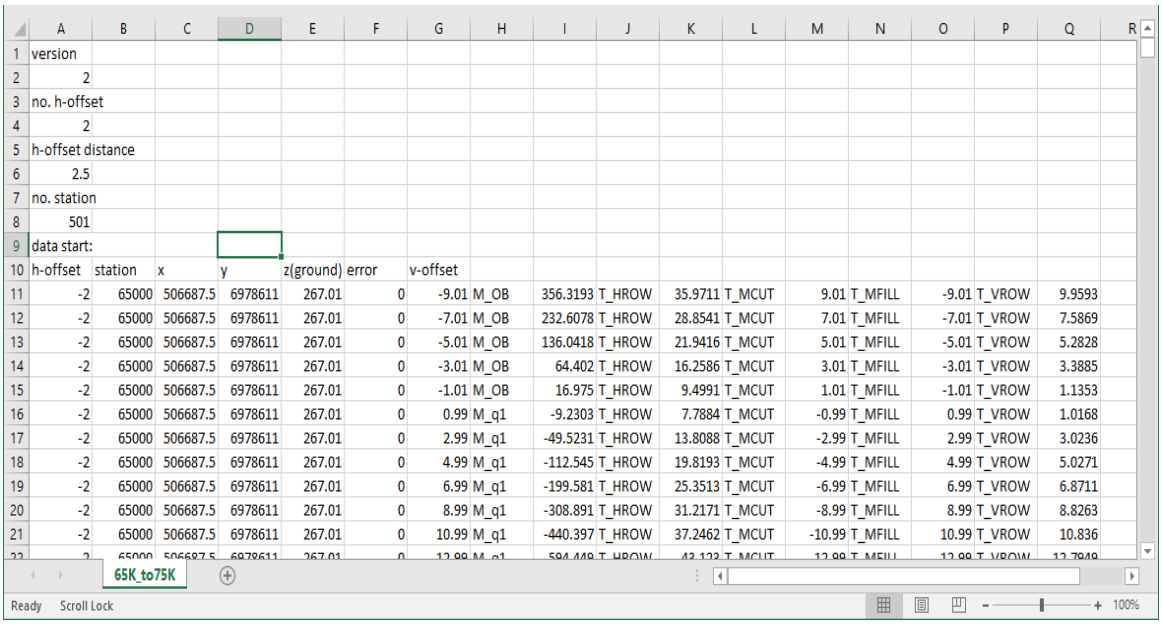
Sections, vertical and horizontal offsets

The main body of the file consists of a series of variable length records where each record has a Horizontal Offset, Station and Vertical Offset.



File Structure

Fields



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1	version																
2	2																
3	no. h-offset																
4	2																
5	h-offset distance																
6	2.5																
7	no. station																
8	501																
9	data start:																
10	h-offset	station	x	y	z(ground)	error	v-offset										
11	-2	65000	506687.5	6978611	267.01	0	-9.01	M_OB	356.3193	T_HROW	35.9711	T_MCUT	9.01	T_MFILL	-9.01	T_VROW	9.9593
12	-2	65000	506687.5	6978611	267.01	0	-7.01	M_OB	232.6078	T_HROW	28.8541	T_MCUT	7.01	T_MFILL	-7.01	T_VROW	7.5869
13	-2	65000	506687.5	6978611	267.01	0	-5.01	M_OB	136.0418	T_HROW	21.9416	T_MCUT	5.01	T_MFILL	-5.01	T_VROW	5.2828
14	-2	65000	506687.5	6978611	267.01	0	-3.01	M_OB	64.402	T_HROW	16.2586	T_MCUT	3.01	T_MFILL	-3.01	T_VROW	3.3885
15	-2	65000	506687.5	6978611	267.01	0	-1.01	M_OB	16.975	T_HROW	9.4991	T_MCUT	1.01	T_MFILL	-1.01	T_VROW	1.1353
16	-2	65000	506687.5	6978611	267.01	0	0.99	M_q1	-9.2303	T_HROW	7.7884	T_MCUT	-0.99	T_MFILL	0.99	T_VROW	1.0168
17	-2	65000	506687.5	6978611	267.01	0	2.99	M_q1	-49.5231	T_HROW	13.8088	T_MCUT	-2.99	T_MFILL	2.99	T_VROW	3.0236
18	-2	65000	506687.5	6978611	267.01	0	4.99	M_q1	-112.545	T_HROW	19.8193	T_MCUT	-4.99	T_MFILL	4.99	T_VROW	5.0271
19	-2	65000	506687.5	6978611	267.01	0	6.99	M_q1	-199.581	T_HROW	25.3513	T_MCUT	-6.99	T_MFILL	6.99	T_VROW	6.8711
20	-2	65000	506687.5	6978611	267.01	0	8.99	M_q1	-308.891	T_HROW	31.2171	T_MCUT	-8.99	T_MFILL	8.99	T_VROW	8.8263
21	-2	65000	506687.5	6978611	267.01	0	10.99	M_q1	-440.397	T_HROW	37.2462	T_MCUT	-10.99	T_MFILL	10.99	T_VROW	10.836

Example RoadEng Areas CSV File

Header

The first 9 fields of the file contain the following information:

Version – file version.

No. h-offset - number of horizontal offsets.

no. station – number of stations.

Record Fixed Fields

h-offset	Horizontal distance from the baseline horizontal alignment measured perpendicularly.
station	Station with respect to the baseline offset.
X, Y	X,Y coordinates of the offset alignment
Z(ground)	Elevation of ground at node horizontal offset.
error	Flag indicating an error (node position is invalid).
v-offset	Vertical offset from Z (ground)
Record Variable Fields….	See description below

Record Variable Fields (materials and and/or pseudo materials)

The remaining fields in the record consist of pairs of meta materials and corresponding areas in the form:

Code1, Value1, Code2, Value2, …

Where Code(i) is in the form of either:

M_xx = *Standard Material* where xx is the name of the material.

or

T_yy = *Pseudo Material* where yy is the name of the variable.

Standard Materials (M_xx)

xx = material id (for example OB for overburden or SR for solid rock) or a *Generic quality indicator* m_q1 to m_q4. Generic quality indicators are used to allow a variety of materials to be used for fill provided they meet the quality criteria.

Value(i) = the cut/fill area for each material (can be positive or negative). Fill is negative cut is positive.

Pseudo Materials (T_yy) Under Construction!!!

Pseudo materials are optional and can used to describe geometric qualities of the cross section. The can be used for constraint checking. The following Pseudo Materials are pre-defined:

T_MCUT – centerline cut depth
T_FILL – centerline fill depth
T_HROW – horizontal offset to right of way
T_VROW – vertical offset to right of way

Sample Code (C++) to read Softree Optimal Areas CSV File

The following sample code reads the materials for a record (corresponding to a single horizontal offset, station and vertical offset).

```
////////////////////////////////////////////////////////////////////////////
// c++ Code example -- read a record
#define MAX_INI_LEN             512

static TCHAR DELIMCHAR[] = _T(",");
static TCHAR TEMPLATEPREFIX[] = _T("T_");
static TCHAR VOLUMEPREFIX[] = _T("M_");

////////////////////////////////////////////////////////////////////
// Read a single station
BOOL ReadStationEntries(const _TCHAR *_pStr, bool readErrorLevel)
{

        double cut, fill;
```

```
	TCHAR buffer[MAX_INI_LEN];
	wcscpy_s(buffer, _pStr);

	_TCHAR *pToken, *pNextToken;

	// h-offset
	pToken = wcstok_s(buffer, DELIMCHAR, &pNextToken);
	if (pToken == NULL)  return FALSE;

	// Station
	pToken = wcstok_s(NULL, DELIMCHAR, &pNextToken);
	if (pToken == NULL)  return FALSE;
	double dStn = _wtof(pToken);
	if (getStn() != DEFAULTVALUE)
	{
		if (fabs(dStn - getStn()) > EPSILON)
		{
			ASSERT(FALSE);
			return FALSE;
		}
	}

	// X
	pToken = wcstok_s(NULL, DELIMCHAR, &pNextToken);
	if (pToken == NULL) return FALSE;
	double dX = _wtof(pToken);

	// Y
	pToken = wcstok_s(NULL, DELIMCHAR, &pNextToken);
	if (pToken == NULL) return FALSE;
	double dY = _wtof(pToken);

	// Z
	pToken = wcstok_s(NULL, DELIMCHAR, &pNextToken);
	if (pToken == NULL) return FALSE;
	double dZ = _wtof(pToken);

	init(dStn, dX, dY, dZ);

	// error
	pToken = wcstok_s(NULL, DELIMCHAR, &pNextToken);
	if (pToken == NULL) return FALSE;
	bool error = (_wtof(pToken) != 0);

	if (error)
	{
		if (!readErrorLevel)
		{
			//skip this level
			return TRUE;
		}
		m_bError = true;
	}
	m_levelError.push_back(error);


	// Level
```

```
	pToken = wcstok_s(NULL, DELIMCHAR, &pNextToken);
	if (pToken == NULL)
	{
		//no levels, bridge, skip the rest
		m_isBridge = true;
		return TRUE;
	}

	double dLevel = _wtof(pToken);

	while (true)
	{
		bool bMaterial = true;

		// Name
		pToken = wcstok_s(NULL, DELIMCHAR, &pNextToken);
		if (pToken == NULL)  break;

		_TCHAR Name[MAX_INI_LEN];
		wcscpy_s(Name, pToken);

		_TCHAR PrefixName[MAX_INI_LEN];
		wcsncpy_s(PrefixName, pToken, 2);
		if (PrefixName[1] != VOLUMEPREFIX[1])
		{
			ASSERT(FALSE);
			return FALSE;
		}
		else
		{
			if (PrefixName[0] == VOLUMEPREFIX[0])
				bMaterial = true;
			else
			{
				if (PrefixName[0] == TEMPLATEPREFIX[0])
					bMaterial = false;
				else
				{
					ASSERT(FALSE);
					return FALSE;
				}
			}
		}

		_TCHAR MatOrTemName[MAX_INI_LEN];
		wcscpy_s(MatOrTemName, pToken + 2);

		// Value
		pToken = wcstok_s(NULL, DELIMCHAR, &pNextToken);
		if (pToken == NULL)  return FALSE;
		double data = _wtof(pToken);

		// (MAT,Vol.) positive for cut and negative for fill
		if (bMaterial)
		{
			if (data > 0)
			{
```

```
                        // cut area
                        cut = data;
                        // setStationMaterialInfoFromFile(MatOrTemName,
dLevel, data, 0.0);
                }
                else
                {
                        // fill area
                        fill = data
                        // setStationMaterialInfoFromFile(MatOrTemName,
dLevel, 0.0, fabs(data));
                }
        }
        else
        {
                // template object
                // createTemplateObj(MatOrTemName);
                // setData(MatOrTemName, data, dLevel);
        }
    }

    return TRUE;
}
```