Getting Started with OpenTD

Assembly Version 92.0.0.0

Prepared, distributed, and supported by:

Ansys, Inc. Thermal Systems

Boulder, Colorado

(303) 971-0292

www.crtech.com

OpenTD Authors:

Matthew D. Garrett

Timothy D. Panczak

Mark J. Schmidt

Dave Wilkins

Michael A. Madden

Jessica T. Sia

Release Notes: Assembly Version 92.0.0.0

Notes

- For this release and all subsequent releases, OpenTD assemblies will no longer include the version in their filenames. The filenames will not change with each version.
- Assembly versions will no longer correspond to TD versions (24.2.0.0, etc.)
- The first assembly version chosen for this new scheme is 92.0.0.0.

New Features

- Comparer now supports different comparison algorithms for different types of data
- all methods that used to accept Dataset now accept IDataset for greater flexibility
- DatasetFactory.Load can now be used to load sav, CSR, or savx with path, without knowing type of file
- MCH_USED added to StandardDataSubtypes
- Now handles domain names from XREF dwgs
- Added TIMEN2 support to logic objects

Performance Improvements

- CaptureGraphicsArea is more reliable
- Fixed various Orbit parameters to correctly use flux or temperature conversions
- Fixed rename behavior for all properties, including stacks. Entities that use the property now reflect the rename.
- User preferences now set correctly from OpenTD
- CreateHeatLoad now works with tets
- Fixed bug where results in UDFA's associated with fluid objects returned NaN's
- Fixed bug where UDFA results were incorrect
- One-way conductors now show positive heat flow if flow is in the direction of "from" node to "to" node (was reversed)
- Now reading BotInsulationNodeSubmodelIdOption correctly for surfaces

Release Notes: OpenTD 2024 R2

New Features

- added case set group name parameter to TDSF_CoSolver constructors
- added Message member to CompareSuite; just returns Log member
- twin lump ids are now automatically sequenced if lump.TwinId == 0
- added auto start of OpenTD servers on _SAVEAS for all versions of OpenTD

Release Notes: OpenTD 2024 R1

Note

This was an administrative release. There is only one new feature: added Comparer.ToString() method.

Release Notes: OpenTD 2023 R2

Major New Features

- added support for controlling item visibility via the ThermalDesktop.VisibilityManager
- added ReadMeshDataFile method to read various mesh data formats into TD
- added support for stack aliases
- added DynamicSindaStatus class for interacting with messages generated by Dynamic Sinda
- added limited support for new beta SaveX file format

Other New Features

- added MeshDisplayer.BaseTrans property -- now you can move FEM's
- added experimental CaseSetManager.IsCaseRunning() method for asynchronous cases
- added Contactor.Comment property
- use lightweight SubmodelNameData's to identify submodels for FloCAD objects
- added GetNumberOfDbObjects(...) method
- mplemented ConvertFDtoFE method
- Matrix3d.SetOrigin now returns the new matrix instead of void

- Dataset.Factory.Load method to load sav, savx, or CSR
- Comparer ctor overload to accept IDatasets instead of Datasets
- added Close and ReOpen methods to IDataset
- added CloseDatasets and ReOpenDatasets methods to Comparer
- added ability to rename and delete aliases
- added CaseSetManagerOptions.ShowTextScreenDuringRun member
- added Conductor.UseGlobalAccelm member
- added RcEntityData.GlobalContactArray member
- added RcSolidElement.AnalysisGroupsVolumetric member
- added RcFdSolidData.AnalysisGroupsVolumetric member
- added FkLocator.Anchor member
- added Pipe.LengthDivisions and RadialDivisions members
- added Tie.LengthDivisions, RadialDivisions, and UseGlobalAccelm members

Performance Improvements

- CreateCone returns correct node names
- Polygon.Update() no longer duplicates vertices
- CoSolver.Continue() returns -1 if SF disconnected, instead of an exception
- New scr filename and pipe name conventions to avoid Windows Defender mistaking scr files for viruses
- avoid issues with duplicate handles in Assembly
- fix parsing mixed-case subtypes like "DeltaP"
- fix GetMeshFD(s) so it populates returned object(s) correctly
- GetPipe no longer returns pipes with blank signatures
- pipe ties now handle domains correctly
- CaseSet.Run and CaseSetManager.Run now handle casesets with drive symbols from Excel

Release Notes: OpenTD 6.3

Major New Features

- added Addin capability
- added support for Measures
- added full support for global dwg preferences
- added support for Post Processing Data Mappers
- added support for legacy TD COM interface via ThermalDesktop.SendLegacyComCommand method
- added SF_Launcher for running text-based input models in SINDA/FLUINT
- added SF_CoSolver for running and controlling text-based input models in SINDA/FLUINT
- added access to Logging namespace, allowing client programs to use our rich logging features
- added OpenTDMixedInterface.dll, providing access to some OpenTD features via C/C++/Fortran

Other New Features

- added GetSymbolValue method to return an evaluated symbol value from TD
- added Show/HideModelBrowser methods
- added support for getting and setting the UCS
- added ProgressBar class to control RadCAD progress bar
- added IUserBreak interface to allow end users to cancel an Addin operation
- added several AutoCAD commands for controlling OpenTD servers
- updated all public methods that accept List to accept IEnumerable instead, for LINQ and general flexibility
- updated LogicObjectRoutineTypes values for clarity
- ThermalDesktop.Quit() now defaults to a gentle quit, but can be forced to kill the acad process
- added DomainManager.GetDomain method for getting the entities in any type of domain
- added DataArray constructor that just accepts a list of doubles
- added Enable/Disable Undo methods -- disabling undo can speed up operations
- added overloaded ThermalDesktop.SetView methods, which wrap the Restore*View methods

- added ThermalDesktop.ClearSelection method
- added ThermalDesktop.GetMainWindow method to get the AutoCAD window to use as the owner of OpenTD dialogs
- added Notes.HasPassword list to identify notes that are password-protected
- added Comparer.SaveExceedancePlots method
- Comparer now works with registers
- added Comparer.CompareJustFinalRecord member
- added Get/SetInstanceData methods to allow storage of general data with a TD instance

6.3 New Features Included in 6.2

- added support for 3d polylines
- added ThermalDesktop.Get- and SetNodeCorrespondenceMap methods
- added GetSelection method to prompt the user to select entities manually in TD
- added SetSelection method to highlight objects in the TD graphics window
- added GetEntityTypes method to get entity types from handles
- test applied to objects for consistency before creating or updating heatload
- added GetXREFpathnames method
- added DomainManager.GetAnysetDomain method for getting the entities in an ANYSET domain
- added SS_CONSTANT_TEMP_APPLIED_NODES as an allowable value for Heater.SSMethod

6.3 Performance Improvements Included in 6.2

- sped up most operations by eliminating calls to CheckIfConnected()
- only allow one node in From list in CreateConductor
- use a single instance of CaseSetManager per ThermalDesktop to avoid confusion
- LocalTrans no longer deletes rotation expressions
- GetSubmodels and GetFluidSubmodels now scan database for submodel names before returning
- WeightedAverageDataArray no longer allows negative weights
- Getter speed improved by handling expression units more efficiently

Release Notes: OpenTD 6.2

Major New Features

- added FD Solids
- greater support for FEM's
- OpenTDv62.CoSolver to connect SINDA/FLUINT to other analysis codes
- added support for TD Datasets and contour plotting
- added FloCAD Compartments and Acceleration
- added Trackers
- added Pressure loads
- added logic objects: ArrayInterpolation, PID, UDFA
- added AutoCAD arcs, circles, ellipses, splines, helices, and polylines
- added CaseSetManager class to set Case Set Manager options and run in batch mode
- added UserPreferences class for manipulating some global preferences
- added support for reading PCS files to determine model topology from results
- added Results.Dataset.Browser class to determine heat flows between groups of entities
- added support for reading UDFA results
- overhauled OpenTDv62.Results to emphasize groups of data: submodels, domains, arbitrary groups
- OpenTDv62.Results now outputs NaN for missing data, and uses NaN in plots and DerivedDataArrays
- added support for Text Transient files
- added SpreadsheetDataFile method for reading csv or similar files
- added support for custom DataSubtypes (you're not restricted to T, PL, etc. -- can make your own)
- added FormulaDataArray for combining DataArrays with arbitrary formulae
- added WeightedAverageDataArray class
- added MaxDataArray and MinDataArray types (to return loci of extreme values across input arrays)
- added SumDataArray

Other New Features

- added CompareSuite and CompareAssertion classes, to compare multiple pairs of Datasets
- added Exceedances member, and PlotExceedances and GetExceedancePlots to Comparer
- added EllipticCylinder class (accidentally left out of 6.1)
- added methods to get all rectangles, cones, etc. (all FD Surface types)
- added TdConnectConfig.ShowAcadSplashScreen and .AdditionalAcadCommandline members
- added CreateIn(ThermalDesktop) methods to all entities
- added UpdateIn(ThermalDesktop) methods to all entities
- added Connection.IsEmpty() method
- added AttachedNodeHandles member to all FD surfaces (already included for FD solids and finite elements)
- added DataArrayCollection.Dimension and DataItemCount members
- added ThermalDesktop.GetOpticalPropDBPathname and .GetThermoPropDBPathname methods
- added RadiationAnalysisGroupManager.GetDefault method
- added DataArrayCollection.GetTranspose method
- superceded SubmodelDataArrayCollection and DomainNodeDataArrayCollection with new methods for specifying groups
- added CreatePipe(DbObject centerline) convenience method
- added convenient Write and WriteLine methods to StandardOutput
- updated Contactor/TEC with new features for 6.2
- added ThermalDesktop.GetViewNames method
- now allow CreatePort to create ports with no connections by passing an empty Connection (Handle == "")
- update Plot2d.AddSeries(DataArrayCollection) to allow using first array as x data
- added DataArray copy constructor
- added DataTypeFamilies enum (thermal, fluid, other) and methods to determine from DataTypes or DataSubtypes
- added ThermalDesktop.GetLayerByName method
- added ThermalDesktop.GetCurrentLayer() and .SetCurrentLayer(string name)
- added SaveFile.Close() method and made SaveFile IDisposable

- added CaseSet.ReplaceFilenames method
- added UnitsData constructor that accepts a units expression string
- added read-only members to Node class to help navigate FD/FEM networks (AttachedObjectHandles, etc.)
- added Matrix3d.ToString method
- ExportNodeInfo now defaults to returning a List<string> with the same strings that would otherwise be written to screen or file
- use RcEntityData.AnalysisGroups dictionary to simplify specifying FD Surface radiation analysis groups
- added ThermalDesktop.CreateNode(Point3d origin) method

Performance Improvements

- speed increase for DataArrayCollection Dataset.GetData(...)
- initialize RadCAD members in constructors to reduce errors with CreateIn methods
- fixed NodeBreakdownData to default to a blank Boundaries list
- fixed bug where Assembly.AxisSize wasn't scaling with WorkingUnits
- make SaveFile's threadsafe

Contents

1. Introduction and Prerequisites

OpenTD is an Application Programming Interface (API) for Thermal Desktop (TD) that allows you to automate many of the tasks currently performed interactively using TD's Graphical User Interface (GUI). OpenTD gives you the tools to programmatically create, query, edit and delete models, interact with end users, run and control solutions, and manipulate results. You can use any .NET language to interact with OpenTD (C#, VB.NET, F#, etc.) or any system that can load .NET assemblies such as MATLAB or Python.

Regardless of how you interact with OpenTD, you will need to have at least an intermediate understanding of .NET object-oriented programming. If you are starting from scratch, we recommend learning C#, since it is the language that we support. There are many excellent books and internet tutorials available for learning C#. For the remainder of this guide, we will assume you have at least an intermediate knowledge of object-oriented programming and C#, and some familiarity with the .NET global assembly cache (GAC).

OpenTD was installed beginning with TD 6.1, which was released in early 2019. (Preliminary versions of OpenTD were called "TD API".)

OpenTD targets .NET Framework version 4.8 and requires AutoCAD 2018 or above. It does not work with .NET Core or Standard. You must target the .NET Framework.

We will now introduce OpenTD concepts and syntax in a series of example programs. The latter programs build on concepts introduced in the earlier ones, so we recommend you try them in order.

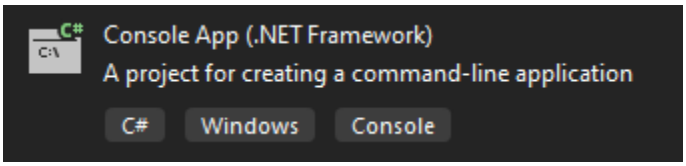
Copying code examples from pdf files is problematic, so we've made the examples available at https://www.crtech.com/forum/topic/getting-started-opentd-92000-examples .

2. Creating TD Models

We will start our tour of OpenTD by learning how to connect to TD and create new entities like nodes and conductors.

2.1. Hello World (Start TD and Create a Node)

Let's create a simple OpenTD program. Start by creating a C# console application in Visual Studio. Look for the template called "Console App (.NET Framework)", not ".NET Core" or ".NET Standard". It should look something like this:



Next, add a reference to the OpenTD.dll assembly, which you can find in the GAC. (Try looking under C:\Windows\Microsoft.NET\assembly\GAC_MSIL\OpenTD.) If there are multiple directories, use the one with the highest Assembly Version, which you will see in the directory name, for example "…_92.0.0.0__...".

Add the following code, then compile and run the program:

```
using OpenTD;
namespace OpenTDGettingStarted
{
    class HelloWorld
    {
        public static void Main(string[] args)
        {
            var td = new ThermalDesktop();
            td.Connect();

            var n = td.CreateNode();
            n.Comment = "Hello World!";
            n.Update();
        }
    }
}
```

When you run the program, an instance of TD should open and a node should be created at (0, 0, 0). When you edit the node, the comment should be "Hello World!". If any of these things are not true, check the Troubleshooting section at the end of this guide.

Assuming it worked, let's examine how. First, we created a *ThermalDesktop* object called td. This object represents one instance of TD. It has hundreds of methods for interacting with TD models. A single OpenTD client program can create an arbitrary number of ThermalDesktop instances, allowing you to manipulate several models and communicate between them.

Next, we called the ThermalDesktop.*Connect()* method. By default, this will start a new instance of TD using the latest version of AutoCAD installed. You can control how it works using the ThermalDesktop.*ConnectConfig* property (see Section 9.1).

Connect(), like most ThermalDesktop commands, is called synchronously, so it will only return control to your program once it finishes. If there is a problem, it will throw an exception. All OpenTD methods throw exceptions if there is a problem; you do not need to check return values for success.

Once Connect() returned, we called ThermalDesktop.*CreateNode()* to create a node in TD with default settings. We put the return value in a variable called n. This variable is of type *Node* and represents the TD node in our client program.

Next, we updated the *Comment* member of n. This only updated the comment for the client-program Node. To send that update to TD, we called the Node.*Update()* method. This is an important concept to understand; when you work with objects in your client program that represent objects in TD, they do not automatically propagate their changes to TD. To do that, you need to call the Update() method. (Some objects also have *UpdateFromTD()* methods to get the latest changes from TD.)

2.2. Create Nodes and a Conductor

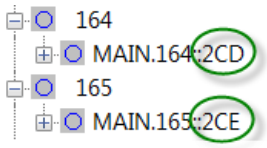
This program demonstrates how to create two nodes and connect them with a conductor. To try it, create a .NET Framework C# console application that references OpenTD, add the following code, then compile and run it.

```
using OpenTD;
namespace OpenTDGettingStarted
{
```

```
    class CreateNodesAndConductors
    {
        public static void Main(string[] args)
        {
            var td = new ThermalDesktop();
            td.Connect();

            var n1 = td.CreateNode();
            n1.Submodel = "bar";
            n1.Id = 100;
            n1.Update();

            var n2 = td.CreateNode();
            n2.Submodel = "bar";
            n2.Id = 110;
            n2.Origin = new Point3d(1, 1, 0);
            n2.Update();

            var c = td.CreateConductor(n1, n2);
            c.Submodel = "bar";
            c.Value = 10;
            c.Update();
        }
    }
}
```

How did this program work? After starting a new instance of TD, we created two Node objects, n1 and n2, using the CreateNode() method.

When TD created each node, it set the Node.*Handle* property to a unique identifier, the same string you may have noticed in the TD Model Browser:



Since TD allows duplicate SINDA names for some entities, OpenTD uses AutoCAD *handles* to identify most entities uniquely. (See Section 2.7 for a detailed discussion of handles and related concepts.)

Next, we called the *CreateConductor* method. This method accepts two *Connections* representing the nodes connected to the conductor. Each Connection consists of a handle and a *marker*. As mentioned above, a handle is a unique identifier for a TD entity. A marker is an integer that determines how something is connected such as Top or XMAX. (See Section 2.7.)

When we called CreateConductor, we simply passed it our two nodes, n1 and n2. OpenTD knows how to implicitly create new Connections from Nodes by reading the Node.Handle property and assuming a default value for the Connection.Marker property, which is fine because node connections do not use markers. (See Section 9.4 for a discussion of implicit casting.)

2.3. Use a Loop to Create Many Layers, Nodes, and Conductors

It is easy to create two nodes and a conductor using the GUI. The real usefulness of an API like OpenTD is in automating things that are tedious and/or time-consuming. For example, the following program creates 101 nodes in a sinusoid pattern connected by conductors and puts them on 101 randomly-colored layers. This program only uses a few new OpenTD concepts, but it demonstrates how OpenTD types can be combined with C# statements to quickly accomplish things that would take much longer using the GUI.

```
using System;
using System.Collections.Generic;
using OpenTD;
namespace OpenTDGettingStarted
{
    class UseLoop
    {
        public static void Main(string[] args)
        {
            // parameters
            const int numNodes = 101;
            const string submodel = "wavybeam";
            const int startingNodeNum = 200;
            const string layerPrefix = "Random Layer";
            const double Length = 10.0;
            const double amplitude = 2.0;
            const double freq = 4;
            var startingPoint = new Point3d(0, 1, 0);

            // start TD
            var td = new ThermalDesktop();
            td.Connect();

            // create nodes and put them in a list for later use
            var r = new Random();
            var nodes = new List<Node>();
            for (int i = 0; i < numNodes; ++i)
            {
                var layer = td.CreateLayer(layerPrefix + " " + i);
                layer.ColorIndex = r.Next(254) + 1;
                layer.Update();
                double x = Length / (numNodes - 1) * i;
                double y = amplitude * Math.Sin(freq * x / Length * 2 * Math.PI);
                var nodeDisplacementFromStartingPoint = new Vector3d(x, y, 0);
                var n = td.CreateNode();
                n.Layer = layer.Name;
```

```
                n.Submodel = submodel;
                n.Id = startingNodeNum + i;
                n.Origin = startingPoint + nodeDisplacementFromStartingPoint;
                n.Update();
                nodes.Add(n);
            }

            // create conductors
            for (int i = 0; i < nodes.Count - 1; ++i)
            {
                var c = td.CreateConductor(nodes[i], nodes[i + 1]);
                c.Layer = nodes[i].Layer;
                c.Submodel = submodel;
                c.Update();
            }

            // control view
            td.ZoomExtents();
        }
    }
}
```

In the above program, note that we performed vector addition using the + operator:

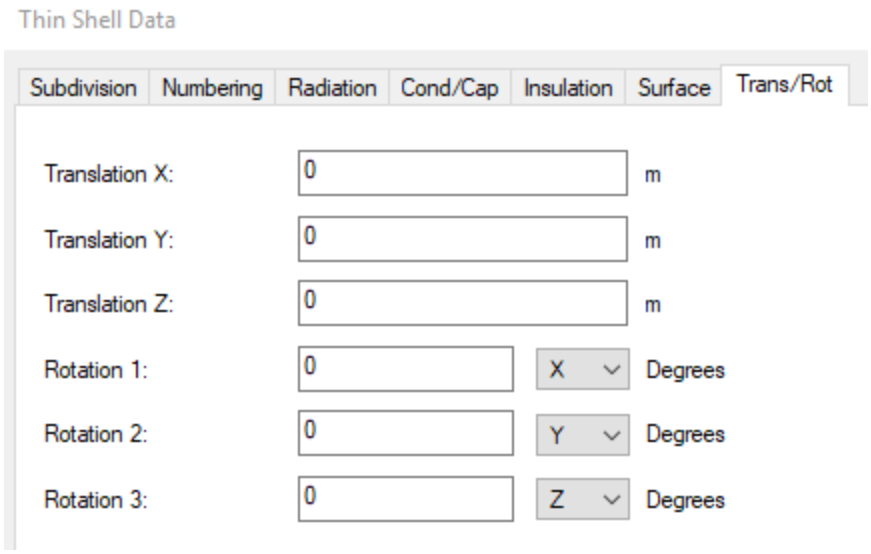
```
                n.Origin = startingPoint + nodeDisplacementFromStartingPoint;
```

OpenTD knows how to perform limited arithmetic with its *Point2d*, *Point3d*, *Vector3d*, and *Matrix3d* objects.

2.4. Create and Position Finite-Difference Surfaces and Solids

You can use OpenTD to create Finite Difference (FD) surfaces and solids, and other geometric entities. When working with geometric entities in OpenTD, you will use two class members to position and orient objects:

- **BaseTrans**: This Matrix3d represents the position and orientation of the entity's local coordinate system relative to the AutoCAD World Coordinate System (WCS). Modifying this member is equivalent to using AutoCAD commands such as MOVE or ROTATE3D to position and orient the entity.
- **LocalTrans**: This *Transformation* represents the position and orientation of the entity relative to its local coordinate system. Modifying this member is equivalent to editing the "Trans/Rot" tab in the GUI:



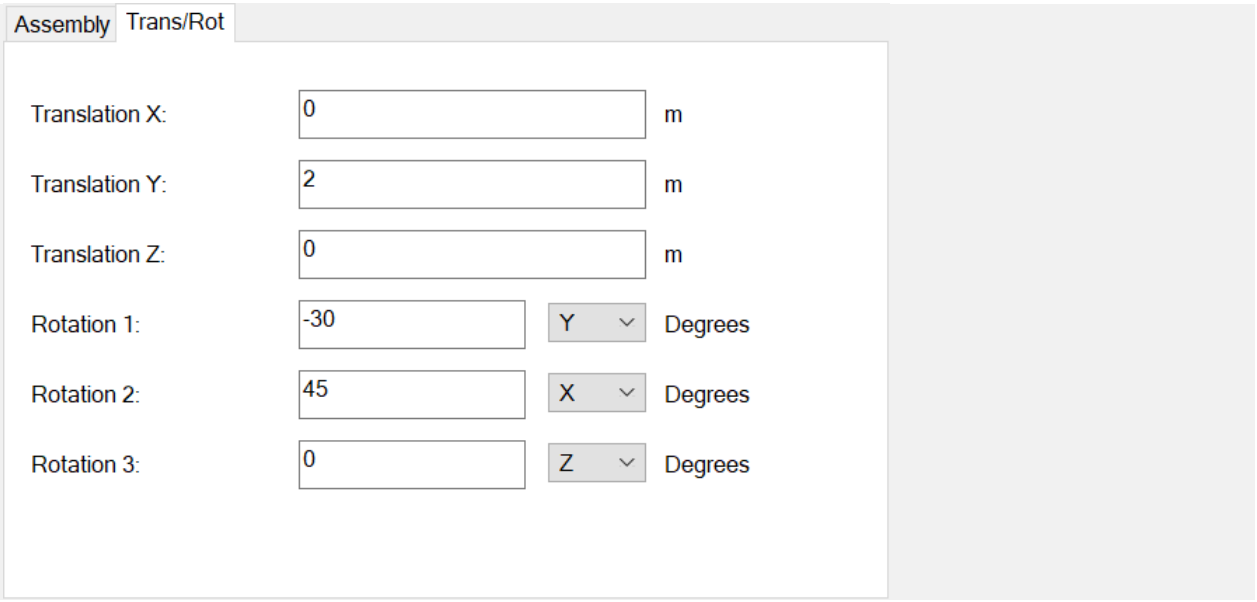
Here is a program that demonstrates creating FD entities and positioning them using BaseTrans and LocalTrans:

```
using OpenTD;
namespace OpenTDGettingStarted
{
    class PositionFiniteDifference
    {
        public static void Main(string[] args)
        {
            var td = new ThermalDesktop();
            td.Connect();

            // create a rectangle and position it using BaseTrans,
            // which is equivalent to moving and rotating it using
            // AutoCAD commands
            var rect = td.CreateRectangle();
            rect.TopStartSubmodel = "plate";
            rect.TopStartId = 1;
            rect.XMax = 1.1;
            rect.YMax = 2.1;
            rect.BreakdownU.Num = 10;
            rect.BreakdownV.Num = 20;
            rect.BaseTrans.SetToRotX(30);
            rect.BaseTrans.SetOrigin(new Point3d(0, 0, 1));
            rect.Update();

            // create an FD solid brick and position it using LocalTrans,
            // which is equivalent to moving and rotating it using
            // the TD Trans/Rot tab
            var fdBrick = td.CreateSolidBrick();
            fdBrick.StartSubmodel = "brick";
            fdBrick.StartId = 1;
            fdBrick.XMax = 0.19;
            fdBrick.YMax = 0.31;
            fdBrick.ZMax = 0.50;
            fdBrick.BreakdownU.Num = 2;
```

```
            fdBrick.BreakdownV.Num = 3;
            fdBrick.BreakdownW.Num = 5;
            fdBrick.LocalTrans.Tx = 0.5;
            fdBrick.LocalTrans.Ty = 1.0;
            fdBrick.LocalTrans.Tz = 0.5;
            fdBrick.LocalTrans.Axis1 = 2; // rotate about z
            fdBrick.LocalTrans.Rot1 = 30; // rotate 30 deg about z
            fdBrick.Update();

            // control view
            td.SetVisualStyle(VisualStyles.THERMAL_PP);
            td.RestoreIsoView(IsoViews.NE);
        }
    }
}
```

2.5. Additional Information on Positioning Entities using BaseTrans and LocalTrans

As discussed in Section 2.4, many OpenTD classes contain the members BaseTrans and LocalTrans, which both can be used to position entities. BaseTrans is equivalent to using AutoCAD commands like MOVE or ROTATE3D to position the entity, while LocalTrans is equivalent to using the Trans/Rot tab that can be found when editing an entity:



These particular Trans/Rot parameters would result in the following transformations:

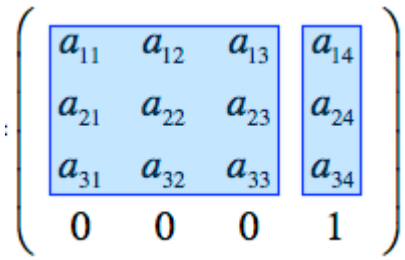
1. Translate 2 m along the entity's base Y axis
2. Rotate -30 deg about the entity's Y axis

3. Rotate 45 deg about the entity's new X axis, that is, rotations are intrinsic (they are about the entity's current axes, not some fixed axes).

LocalTrans is an instance of the *Transformation* class. To use it to perform the operations listed above, you'd use something like this:

```
a.LocalTrans.Ty = 2;
a.LocalTrans.Axis1 = 1; // Y
a.LocalTrans.Rot1 = -30;
a.LocalTrans.Axis2 = 0; // X
a.LocalTrans.Rot2 = 45;
a.Update();
```

BaseTrans is an instance of the *Matrix3d* class, that is, it is a 4x4 matrix representing a geometric transformation:



The upper-left 3x3 submatrix represents a rotation. (In general it could represent many other operations, but when positioning entities using OpenTD, you should stick to rotations.) The 3x1 column vector on the right represents a translation. The bottom row should always be 0 0 0 1.

OpenTD positions entities at the WCS origin, then transforms them by BaseTrans, followed by LocalTrans. Some of the methods available to BaseTrans are frequently misunderstood. There is a family of methods that sets the rotation matrix:

- SetToRotation
- SetToRotX
- SetToRotY
- SetToRotZ

These cannot be used sequentially to perform sequential rotations. Each one clears the rotation matrix and sets it to a single rotation. For example, SetToRotX(45) creates a rotation matrix that rotates 45 deg about the X axis.

To perform sequential rotations, use matrix multiplication. For example, to perform a 10 deg rotation about an entity's Z axis, followed by a 50 deg rotation about its new X axis (that is, intrinsic rotations), you could use something like:

```
var A = new Matrix3d().SetToRotZ(10);
var B = new Matrix3d().SetToRotX(50);
a.BaseTrans = A * B;
a.Update();
```

For intrinsic rotations, multiply matrices in order from left to right.

Translations can also be included using matrix multiplication. For example, to perform a 10 deg rotation about an entity's Z axis, followed by a 2 m translation along its X axis, followed by a 50 deg rotation about its X axis, you could use something like:

```
var A = new Matrix3d().SetToRotZ(10);
var T = new Matrix3d();
T.SetOrigin(new Point3d(2, 0, 0));
var B = new Matrix3d().SetToRotX(50);
a.BaseTrans = A * T * B;
a.Update();
```

2.6. Create Finite Elements

In TD, finite elements can be created directly by attaching them to existing nodes, but the preferred approach is to use an FE Mesh Importer. OpenTD allows you to use either method.

Creating a finite element mesh using the *FEMeshImporter* class is demonstrated in the following program:

```
using System;
using OpenTD;
namespace OpenTDGettingStarted
{
    class CreateFiniteElements
    {
        public static void Main(string[] args)
        {
            var td = new ThermalDesktop();
            td.Connect();
```

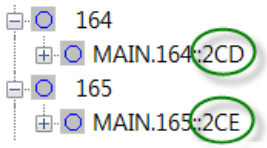
```
// In TD and OpenTD, you can create a FEM with nodes and elements
// directly, but the preferred approach is to use an FE Mesh Importer.

// We'll demonstrate how to use a mesh importer. We'll start
// by creating an empty one:
bool useUCS = false;
var meshImporter = td.CreateFEMeshImporter("a mesh importer", useUCS);

// We're going to call the FEMeshImporter.SetMesh command, but
// first we'll need to construct an FEMesh to pass to it. We'll
// use linear quads, but a full complement of linear and quadratic
// surface and solid element types are available.
// The FEMesh object is a lightweight description of the mesh, with
// lightweight nodes and elements that are only used as input to
// the SetMesh command.
var feMesh = new OpenTD.RadCAD.FEModel.FEMesh();
int uDiv = 10;
int vDiv = 10;
double height = 0.5;
double xPeriods = 2.0;
double yPeriods = 1.0;
double xLen = 5.0;
double yLen = 3.0;
int id = 0;
int elemId = 0;
for (int j = 0; j < vDiv + 1; ++j)
{
    double y = j * yLen / vDiv;
    for (int i = 0; i < uDiv + 1; ++i)
    {
        double x = i * xLen / uDiv;
        double z = height *
            Math.Cos(x / xLen * xPeriods * 2.0 * Math.PI) *
            Math.Cos(y / yLen * yPeriods * 2.0 * Math.PI);
        // lightweight node description:
        var node = new OpenTD.RadCAD.FEModel.Node();
        node.x = x;
        node.y = y;
        node.z = z;
        node.Nx = 0.0;
        node.Ny = 0.0;
        node.Nz = 1.0;
        node.id = ++id;
        feMesh.nodes.Add(node);

        if (i < uDiv && j < vDiv)
        {
            // lightweight surface description:
            var face = new OpenTD.RadCAD.FEModel.SurfaceElement();
            face.id = ++elemId;
            face.order = 1;
            face.numNodes = 4;
            int baseIndex = j * (uDiv + 1) + i + 1;
            face.nodeIds.Add(baseIndex);
            face.nodeIds.Add(baseIndex + 1);
            face.nodeIds.Add(baseIndex + 1 + uDiv + 1);
            face.nodeIds.Add(baseIndex + uDiv + 1);
            feMesh.surfaceElements.Add(face);
```

```
                    }
                }
            }

            // Okay, now we can call SetMesh:
            meshImporter.SetMesh(feMesh);

            td.SetVisualStyle(VisualStyles.THERMAL_PP);
            td.RestoreIsoView(IsoViews.SE);
            td.ZoomExtents();
        }
    }
}
```

If you create a mesh using the FEMeshImporter class (the preferred approach), it is important to note that the *FEMesh* definition provided during creation is only used at that time. After creation, the client-side FEMesh object is not connected to the TD model and if you want to edit the mesh, you will have to get the editable objects to modify. This is discussed in Section 3.2.

2.7. Work with Connections, Handles, Markers, and Domains

To connect objects in TD, you will use a Connection. A Connection contains a handle and a marker. As discussed in Section 2.2 a handle is a string that TD uses to uniquely identify each object in a drawing. You have probably seen them listed in the Model Browser:



In addition to Connections, OpenTD uses handles to find objects when, for example, you call the Update() method on an object. Internally, OpenTD keeps track of which dwg contains the object, and finds it in the dwg using the Handle property of the object.

The other part of a Connection is a marker. This is an integer that specifies how the associated object is connected. For example, marker = 1 connects to the XMIN surface of an FD brick. Marker = 42 connects to XMAX, YMAX, and ZMAX. And Marker = 63 connects to all six surfaces of a brick. You might be wondering if those are random numbers! Converted to binary they make more sense:

Decimal	Binary	Applied Surfaces					
1	0b000001						XMIN
42	0b101010	ZMAX		YMAX		XMAX	
63	0b111111	ZMAX	ZMIN	YMAX	YMIN	XMAX	XMIN

There is a special value for markers: if Marker is set to -999, then the Connection connects to a TD domain. In this special case, the Handle member no longer specifies the handle of a single TD object, but rather the name of the domain.

The following program demonstrates these concepts:

```
using System.Collections.Generic;
using OpenTD;
namespace OpenTDGettingStarted
{
    class WorkWithConnections
    {
        public static void Main(string[] args)
        {
            var td = new ThermalDesktop();
            td.Connect();

            // create a rectangle:
            var brick = td.CreateSolidBrick();

            // create a heatload with default connection:
            // equiv. to td.CreateHeatLoad(new Connection(brick.Handle, 1));
            var heatLoad = td.CreateHeatLoad(brick);
            heatLoad.AppliedType = RcHeatLoadData.AppliedTypeBoundaryConds.SURFACE;
            heatLoad.Update();

            // with marker = 1 = 0b000001, heatload applied to XMIN
            // let's apply it to XMIN and YMIN:
            heatLoad.ApplyConnections[0].Marker = 0b000101;
            heatLoad.Name = "q applied to brick";
            heatLoad.Update();

            // create a rectangle:
            var rect = td.CreateRectangle();
            rect.BaseTrans.SetToRotX(90);
            rect.BaseTrans.SetOrigin(new Point3d(0, 0, 2));
            rect.XMax = 2;
            rect.YMax = 3;
            rect.BreakdownU.Num = 10;
            rect.BreakdownV.Num = 15;
```

```
            rect.Update();

            // create a domain that includes some of the rect nodes:
            var domainConnections = new List<Connection>();
            foreach (Node n in td.GetNodes())
            {
                if (rect.AttachedNodeHandles.Contains(n.Handle))
                {
                    if (n.Origin.Z < 4 && n.Origin.X < 1)
                        domainConnections.Add(new Connection(n));
                }
            }
            td.GetDomainManager().CreateDomain
                ("HEATED", DomainType.NODESET, domainConnections);

            // apply a heat load to the domain:
            var rectHeatLoad = td.CreateHeatLoad(
                new Connection("HEATED", -999));
            rectHeatLoad.Name = "q applied to rectangle domain";
            rectHeatLoad.Update();

            td.SetVisualStyle(VisualStyles.THERMAL);
            td.RestoreIsoView(IsoViews.SW);
            td.ZoomExtents();
        }
    }
}
```

2.8. Work with Units, Symbols, and Expressions

OpenTD offers full support for units, symbols, and expressions. The most important concept to understand is that – except for a few exceptions discussed below – all dimensional values in OpenTD are expressed in the units defined in a thread static variable called *Units.WorkingUnits*, which is completely independent of the drawing units set in any connected TD instance.

You can use expressions in OpenTD anywhere you can use them in the GUI. Look for members named "SomethingExp" to set the expression corresponding to the member "Something". Just like in the GUI, expressions have their own unit system, independent of the dwg units and the WorkingUnits. This is the main exception to the rule that all dimensional values in OpenTD are expressed in Units.WorkingUnits.[1] Any other exceptions will be class members with fixed units in their names.

The following program shows how to use WorkingUnits, symbols, and expressions:

[1] The other exceptions are rare properties that must be entered in a specific unit, for example km for *PlanetParameters.radiusKm*. In all cases these will be indicated by the unit name in the name of the property.

```
using System;
using System.Collections.Generic;
using System.Linq; // for the Select method, below
using OpenTD;
namespace OpenTDGettingStarted
{
    class WorkWithUnitsSymbolsExpressions
    {
        public static void Main(string[] args)
        {
            var td = new ThermalDesktop();
            td.Connect();

            // To set the units used in the GUI, use SetDwgUnits.
            // This is equivalent to setting Preferences->Units.
            // Here we'll set the dwg units to English with inches
            // instead of feet for model length:
            var dwgUnits = new UnitsData();
            dwgUnits.SetToEng();
            dwgUnits.modelLength = UnitsData.ModelLength.INCH;
            td.SetDwgUnits(dwgUnits);

            // OpenTD uses its own unit system called WorkingUnits
            // to control input and output of all dimensional
            // values. In this example we'll set WorkingUnits to SI,
            // then set and get the density of a material in kg/m3.
            // Then we'll set WorkingUnits.modelLength to cm and get
            // the same property, showing that it will now return as
            // kg/cm3. Note that since the dwg units were set to inches
            // and lbm, the value shown in the GUI is 0.289018 lbm/in^3:
            // it's completely independent of the WorkingUnits.
            string materialName = "steel";
            if (td.GetThermoPropss().Select(x => x.Name).Contains(materialName))
                td.DeleteThermoProps(materialName);
            var material = td.CreateThermoProps(materialName);
            Units.WorkingUnits.SetToSI(); // this is the default anyway
            material.Density = 8000; // kg/m3
            material.Update();
            Console.WriteLine(material.Density); // "8000"
            Console.WriteLine(material.Density.ToString()); // "8000 kg/m^3"
            Units.WorkingUnits.modelLength = UnitsData.ModelLength.CM;
            Console.WriteLine(material.Density); // "0.008"
            Console.WriteLine(material.Density.ToString()); // "0.008 kg/cm^3"

            // You can set both the dwg units and WorkingUnits with the
            // SetUnits method. Also, you don't always have to create
            // a new UnitsData. The Units.SI and Units.Eng UnitsData
            // objects are convenient static readonly objects that
            // correspond to standard SI and English units systems.
            td.SetUnits(Units.SI);

            // Create symbols using the CreateSymbol method. Here we'll
            // create a symbol representing a heat load value in Btu/hr:
            string symbolName = "heatload";
            var heatload = td.CreateSymbol(symbolName, "34.12 * 2");
            heatload.Description = "heat load in Btu/hr";
            heatload.Update();
```

```
            // Get evaluated symbol values using GetSymbolValue. These
            // represent the basic symbol values, unmodified by Case Sets
            // or other means.
            var heatloadValue = td.GetSymbolValue(symbolName);
            Console.WriteLine($"{symbolName} value = {heatloadValue}");

            // You can use expressions in OpenTD anywhere you can use
            // them in the GUI. Here we'll create a heatload, set its
            // value expression ("ValueExp") to our symbol created above,
            // then set the units of the expression to BTU/hr to match
            // the symbol:
            var n = td.CreateNode();
            var q = td.CreateHeatLoad(new List<Connection> { n });
            q.ValueExp.Value = symbolName;
            q.ValueExp.units.energy = UnitsData.Energy.BTU;
            q.ValueExp.units.time = UnitsData.Time.HR;
            q.Update();

            td.ZoomExtents();
        }
    }
}
```

2.9. Create Thermophysical Properties using Bivariate Arrays

The following program demonstrates how to create or open a thermophysical property database, and how to create materials in it. One of the material definitions uses bivariate arrays, which are used in many places within SINDA/FLUINT, TD, and OpenTD.

```
using System.Collections.Generic;
using OpenTD;
namespace OpenTDGettingStarted
{
    class CreateThermophysicalProps
    {
        public static void Main(string[] args)
        {
            var td = new ThermalDesktop();
            td.Connect();

            // Let's make sure we're working with an empty database.
            // Note: relative pathnames in OpenTD are relative to the current
            // working directory, which usually starts at the location of your
            // exe file.
            string dbPath = "TemporaryThermoPropDatabase.tdp";
            System.IO.File.Delete(dbPath);
            td.OpenThermoPropDB(dbPath);

            // We'll create a thermophysical property representing Al 6061-T6,
            // with data taken from the Spacecraft Thermal Control Handbook.
            // The handbook uses the following units:
            // density: kg/cm3
            // k: W/(cm.degC)
            // Cp: W-hr/(kg.degC)
```

```
// We'll set WorkingUnits to SI with cm before setting density and k.
// For Cp, we'll need to set energy units to W-hr.
// Since we're working with inconsistent units, we'll save and restore
// whatever working unit system was in use before now.
Units.SaveWorkingUnits();
Units.WorkingUnits.SetToSI();
Units.WorkingUnits.modelLength = UnitsData.ModelLength.CM;
var Al6061 = td.CreateThermoProps("Al6061-T6");
Al6061.Comment = "Al 6061-T6 from Spacecraft Thermal Control Handbook\n"
               + "The Aerospace Corp., 2002";
Al6061.Density = 0.00277;
Al6061.Conductivity = 1.679;
Units.WorkingUnits.energy = UnitsData.Energy.WATT_HOUR;
Al6061.SpecificHeat = 0.267;
Al6061.Update();
Units.RestoreWorkingUnits();

// To rename a thermophysical property, you need to use Rename,
// since thermophysical properties are stored in TD using names as
// identifiers, unlike other entities that use AutoCAD handles.
Al6061.Rename("Aluminum 6061-T6");

// What if you've got a material with anisotropic,
// temperature- and pressure-dependent conductivity?
// For example, here's the conductivity of "Material A":

// conductivity in x and y directions:
// 100 K: 21 W/(m.K)
// 200 K: 25 W/(m.K)
// 300 K: 27 W/(m.K)

// conductivity in z-dir: (W/(m.K))
//          150 K  250 K  350 K
// 50 kPa     3      5      8
// 100 kPa    6      9     10

// Here's how to create a material with temperature-
// and pressure-dependent conductivity.
Units.WorkingUnits.SetToSI();
var materialA = td.CreateThermoProps("Material A");
materialA.Comment = "Aniso k demo";
materialA.Anisotropic = 1;
materialA.VarConductivity = 1;
materialA.VarConductivityY = 1;
var kxyTemp = new List<double> { 100, 200, 300, };
var kxyValue = new List<double> { 21, 25, 27, };
materialA.ConductivityTemp = kxyTemp;
materialA.ConductivityValue = kxyValue;
materialA.ConductivityYTemp = kxyTemp;
materialA.ConductivityYValue = kxyValue;
materialA.VarCondTempPresZ = 1;
materialA.bivarTemperatureZ = new List<double> { 150, 250, 350 };
materialA.bivarPressureZ = new List<double> { 50, 100 };
materialA.bivarConductivityZ = new List<double>
{
    3, 5, 8,
    6, 9, 10,
};
```

```
            materialA.Update();
        }
    }
}
```

After running the program, the kz conductivity for Material A will show that the three lists of doubles we input are now expressed as a bivariate array:

```
Enter values of Temp [K],  on the first line
Enter first value on additional lines as pressure[Pa]
followed by values of conductivity[W/m/K]
```

	150	250	350
50000	3	5	8
100000	6	9	10

The dwg units default to SI so the pressures are shown in Pa, even though we set the OpenTD WorkingUnits to kPa. Remember that WorkingUnits are independent of dwg units (see Section 2.8).

2.10. Create Optical Properties

If you have a RadCAD license, you can create optical properties with OpenTD. The following program creates a simple optical property, and a more complicated wavelength-dependent property:

```
using System.Collections.Generic;
using OpenTD;
namespace OpenTDGettingStarted
{
    class CreateOpticalProperties
    {
        public static void Main(string[] args)
        {
            var td = new ThermalDesktop();
            td.Connect();

            // Let's make sure we're working with an empty optical database.
            // Note: relative pathnames in OpenTD are relative to the current
            // working directory, which usually starts at the location of your
            // exe file.
            string dbPath = "TemporaryOpticalPropDatabase.rco";
            System.IO.File.Delete(dbPath);
            td.OpenOpticalPropDB(dbPath);

            // create simple optical property
            var black = td.CreateOpticalProps("black");
            black.Comment = "ideal black surface";
```

```
            black.Alph = 1;
            black.Emis = 1;
            black.Update();

            // create wavelength-dependent optical property
            var catalac = td.CreateOpticalProps("Cat-A-Lac Black");
            catalac.Comment = "Cat-A-Lac Black from TwoPlates.dwg";
            catalac.UseWaveLengthDepProps = 1;
            catalac.UseVarWaveLengthEmiss = 1;
            // wavelength always in micrometers, regardless of WorkingUnits:
            catalac.emissVarWaveLengthum = new List<double> {
                0.100, 8.000, 10.000, 13.000, 19.000, 20.000,
                30.000, 40.000, 60.000, 110.000, 1000.000,
            };
            catalac.emissVarWaveLengthValue = new List<double> {
                 0.92, 0.92, 0.85, 0.91, 0.94, 0.82,
                 0.95, 0.95, 0.78, 0.6, 0.1,
            };
            catalac.Update();
        }
    }
}
```

2.11. Create Fluid Entities

With a FloCAD license, you can use OpenTD to work with FloCAD entities, using methods like those we've already discussed for thermal entities. Here is a program that demonstrates working with FloCAD:

```
using System.Collections.Generic;
using OpenTD;
using OpenTD.FloCAD;
namespace OpenTDGettingStarted
{
    class CreateFluidEntities
    {
        public static void Main(string[] args)
        {
            var td = new ThermalDesktop();
            td.Connect();

            // create a fluid submodel
            var primary = td.CreateFluidSubmodel("PRIMARY");
            primary.ClearFluidLists();
            primary.FluidIds.Add(6070);
            primary.FluidLetters.Add(0); // 0 = A, 1 = B, etc.
            primary.FluidFilenames.Add(""); // will be set to "Water", b/c ID 6070
            primary.Update();

            // create some lumps
            var lumps = new List<Lump>();
            for (int i = 0; i < 10; ++i)
            {
                var lump = td.CreateLump();
```

```
                lump.Submodel = "PRIMARY";
                lump.Origin = new Point3d(0.1 * i, 1, 0);
                lump.Volume = 1e-4;
                lump.Update();
                lumps.Add(lump);
            }

            // make end lumps into plenums
            lumps[0].LumpType = RcLumpData.LumpTypes.PLENUM;
            lumps[0].InitialPres = 2e5;
            lumps[0].Update();
            lumps[lumps.Count - 1].LumpType = RcLumpData.LumpTypes.PLENUM;
            lumps[lumps.Count - 1].InitialPres = 1e5;
            lumps[lumps.Count - 1].Update();

            // connect the lumps with stubes
            for (int i = 0; i < lumps.Count - 1; ++i)
            {
                var stube = td.CreatePath(lumps[i], lumps[i + 1]);
                stube.FlowArea = 0.003;
                stube.Update();
            }

            // control view
            td.SetVisualStyle(VisualStyles.THERMAL_PP);
            td.RestoreIsoView(IsoViews.SW);
            td.ZoomExtents();
        }
    }
}
```

3. Modifying TD Models

So far, you have seen how to create new entities in TD, but OpenTD can also be used to query existing models and modify their contents. For most entity types, there is a ThermalDesktop method to get all items of that type in the model, and a method to get a specific item. For example, *GetNodes()* returns a list of all Nodes, and *GetNode(string handle)* returns a specific node if you know its AutoCAD handle. (See Section 2.7.) It is often convenient to use the *LINQ Single* or *Where* methods to search a returned list of entities to find the items that meet some criteria. For example, to find all the arithmetic nodes on layer "sheet", you could do something like this:

```
var arithmeticNodes = td.GetNodes().Where
   (x => x.NodeType == RcNodeData.NodeTypes.ARITHMETIC
         && x.Layer == "sheet");
```

The variable "arithmeticNodes" would now be an *IEnumerable<Node>* containing all of the arithmetic nodes on layer "sheet".[2]

Another useful LINQ technique is to use the *Select* method to extract a related list from an input list. For example, the following line creates an IEnumerable<string> containing the handles of all of the nodes returned by GetNodes():

```
var nodeHandles = td.GetNodes().Select(x => x.Handle);
```

To delete any item with an AutoCAD handle, use the *ThermalDesktop.DeleteEntity* method. For items without handles, there are specialized delete methods such as *ThermalDesktop.DeleteSymbol*, which accepts the name of the symbol to delete.

The following examples demonstrate querying, modifying and deleting entities in a model.

3.1. Query and Edit a Model

Here is a program that demonstrates how to query a model and make simple edits. For the purposes of this program, we will first create a model and then query it, but the model could have been created by any means, including the GUI. The query techniques are the same.

```
using System;
using System.Collections.Generic;
using System.Linq;
using OpenTD;
namespace OpenTDGettingStarted
{
    class QueryModel
    {
        public static void Main(string[] args)
        {
            var td = new ThermalDesktop();
            td.Connect();

            Console.WriteLine("Creating a simple model...");
            var barNodes = new List<Node>();
            for (int i = 0; i < 10; ++i)
```

[2] You can use foreach to iterate over the elements of an IEnumerable, just like a List. Or you can convert the IEnumerable to a List using the ToList() method.

```
{
    var n = td.CreateNode();
    n.Submodel = "bar";
    n.Id = i + 1;
    n.Origin = new Point3d(0.01 * i, 1, 0);
    n.Update();
    barNodes.Add(n);
}
for (int i = 0; i < barNodes.Count - 1; ++i)
{
    var c = td.CreateConductor(barNodes[i], barNodes[i + 1]);
    c.Submodel = "bar";
    c.Id = 100* (i + 1);
    c.Update();
}

var roomAir = td.CreateNode();
roomAir.Submodel = "room";
roomAir.NodeType = RcNodeData.NodeTypes.BOUNDARY;
roomAir.Origin = new Point3d(0.055, 1.1, 0);
roomAir.InitialTemp = 300;
roomAir.Update();

var barConnections = new List<Connection>();
foreach (Node n in barNodes)
    barConnections.Add(new Connection(n));
var convection = td.CreateConductor(roomAir, barConnections);
convection.Value = 1;
convection.Submodel = "room";
convection.Id = 33;
convection.Update();

td.ZoomExtents();

Console.WriteLine("Get all nodes in model...");
var allNodes = td.GetNodes();
foreach (Node n in allNodes)
    Console.WriteLine("  " + n);

Console.WriteLine("Edit node BAR.3...");
try
{
    var bar3 = allNodes.Single(x => x.Submodel == "BAR"
                                 && x.Id == 3);
    Console.WriteLine("  Found it. Editing...");
    bar3.Comment = "This node was edited by OpenTD.";
    bar3.Update();
}
catch (Exception ex)
{
    Console.WriteLine("Problem getting or editing BAR.3: "
        + ex.Message);
}

Console.WriteLine("Find all boundary nodes...");
var boundaryNodes = allNodes.Where
    (x => x.NodeType == RcNodeData.NodeTypes.BOUNDARY);
foreach (Node n in boundaryNodes)
```

```
                Console.WriteLine("  " + n);

            Console.WriteLine("Get the nodes connected to each conductor...");
            var nodesInConductor = new Dictionary<Conductor, List<Node>>();
            foreach (Conductor c in td.GetConductors())
            {
                try
                {
                    var nodes = new List<Node>();
                    foreach (Connection conn in c.From)
                        nodes.Add(allNodes.Single(n => n.Handle == conn.Handle));
                    foreach (Connection conn in c.To)
                        nodes.Add(allNodes.Single(n => n.Handle == conn.Handle));
                    nodesInConductor.Add(c, nodes);
                }
                catch (Exception ex)
                {
                    Console.WriteLine(
                        "Problem getting nodes for {0}: {1}", c, ex.Message);
                }
            }
            foreach (Conductor c in nodesInConductor.Keys)
            {
                Console.WriteLine("  " + c + " is connected to:");
                foreach (Node n in nodesInConductor[c])
                    Console.WriteLine("    " + n);
            }

            Console.WriteLine("Try to get a symbol that doesn't exist...");
            var thicknessSymbol
                = td.GetSymbols().Where(x => x.Name == "thickness");
            int count = thicknessSymbol.Count();
            if (count == 0)
                Console.WriteLine("  There was no symbol named 'thickness'.");
            else
                Console.WriteLine("  Found {0} symbol(s) named 'thickness'.", count);
        }
    }
}
```

3.2. Query and Edit Finite Elements

As mentioned previously in Section 2.5, if you want to edit a mesh created by an FEMeshImporter, you will first have to get the objects representing the actual nodes and elements. Editing the FEMeshImporter directly only offers limited functionality. This is demonstrated in the following program:

```
using System;
using System.Linq;
using OpenTD;
using OpenTD.RadCAD.FEM;
namespace OpenTDGettingStarted
{
    class QueryAndEditFiniteElements
```

```
{
    public static void Main(string[] args)
    {
        var td = new ThermalDesktop();
        td.Connect();

        // First we'll create a mesh using an FEMeshImporter:
        var meshImporter = td.CreateFEMeshImporter("a mesh importer", false);
        var feMesh = new OpenTD.RadCAD.FEModel.FEMesh();
        int uDiv = 3;
        int vDiv = 3;
        double height = 0.5;
        double xPeriods = 2.0;
        double yPeriods = 1.0;
        double xLen = 5.0;
        double yLen = 3.0;
        int id = 0;
        int elemId = 0;
        for (int j = 0; j < vDiv + 1; ++j)
        {
            double y = j * yLen / vDiv;
            for (int i = 0; i < uDiv + 1; ++i)
            {
                double x = i * xLen / uDiv;
                double z = height *
                    Math.Cos(x / xLen * xPeriods * 2.0 * Math.PI) *
                    Math.Cos(y / yLen * yPeriods * 2.0 * Math.PI);
                var node = new OpenTD.RadCAD.FEModel.Node();
                node.x = x;
                node.y = y;
                node.z = z;
                node.Nx = 0.0;
                node.Ny = 0.0;
                node.Nz = 1.0;
                node.id = ++id;
                feMesh.nodes.Add(node);

                if (i < uDiv && j < vDiv)
                {
                    var face = new OpenTD.RadCAD.FEModel.SurfaceElement();
                    face.id = ++elemId;
                    face.order = 1;
                    face.numNodes = 4;
                    int baseIndex = j * (uDiv + 1) + i + 1;
                    face.nodeIds.Add(baseIndex);
                    face.nodeIds.Add(baseIndex + 1);
                    face.nodeIds.Add(baseIndex + 1 + uDiv + 1);
                    face.nodeIds.Add(baseIndex + uDiv + 1);
                    feMesh.surfaceElements.Add(face);
                }
            }
        }
        meshImporter.SetMesh(feMesh);

        // As mentioned previously, the FEMesh we passed to SetMesh
        // is a lightweight description of the mesh, suitable for
        // initial creation only. To work with the elements
        // it created, we need to get them from TD:
```

```
            var quads = td.GetLinearQuads();

            // Let's edit all of the elements and their nodes:
            string submodel = "new_submodel";
            var allNodes = td.GetNodes();
            foreach (LinearQuad q in quads)
            {
                q.CondSubmodel = submodel;
                q.TopThickness = 0.01;
                q.Update();
                var quadNodes
                    = allNodes.Where(n => q.AttachedNodeHandles.Contains(n.Handle));
                foreach (Node n in quadNodes)
                {
                    n.Submodel = submodel;
                    n.Update();
                }
            }

            td.SetVisualStyle(VisualStyles.THERMAL_PP);
            td.RestoreIsoView(IsoViews.SE);
            td.ZoomExtents();
        }
    }
}
```

4. Interacting with End Users

4.1. Control the View

There are several techniques you can use to control how the model is displayed in the main Thermal Desktop window:

- Use the ThermalDesktop.VisibilityManager to get and set item and item label visibility.
- Use an object's *Layer* member to change what layer it is on, perhaps moving it to or from a frozen layer.
- Use ThermalDesktop.*GetLayers* or .*GetLayer* to get an existing layer, then change its color, frozen status, and so on.
- Use an object's *ColorIndex* member to directly change its color.

- Use ThermalDesktop.*RestoreIsoView* to set the view to one of the standard isometric views.
- Use ThermalDesktop.*RestoreOrthoView* to set the view to one of the standard orthographic views.
- Use ThermalDesktop.*RestoreView* to set the view to a pre-defined, custom named view. These can also contain layer settings. Create named views in the GUI using the AutoCAD "view" command.
- Use ThermalDesktop.*SetView* and either an *IsoViews* enum, an *OrthoViews* enum, or a string representing a named view to set the view.
- Use ThermalDesktop.*SetVisualStyle* to select wireframe, hidden, shaded, and more.
- Use ThermalDesktop.*ResetGraphics* to redraw everything in the main window.
- Display contour plots using the ThermalDesktop.*DatasetManager*. See Section 7.2.1.

4.2. Capture Graphics Area

Once you have set up the main TD view using the techniques discussed above, you may wish to capture the graphics area as a bitmap for use elsewhere. To do this, use the ThermalDesktop.*CaptureGraphicsArea* method, with the following signature:

```
System.Drawing.Bitmap ThermalDesktop.CaptureGraphicsArea()
```

The Bitmap object it returns can be saved to a file, or further processed within your program.

Known issue: if AutoCAD is in fast-shaded mode then this command will return an all-black image if the visual style is set to SHADED, SHADED_W_EDGES, or WIRE.

4.3. Working with the Selection Set

OpenTD allows you to ask the end user to select entities on screen, just like a TD command. It also allows you to do the opposite: set the selection set programmatically. These two operations are performed with the ThermalDesktop.*GetSelection* and *SetSelection* methods, respectively. There is also a *ClearSelection* method.

GetSelection returns a list of *EntityDescriptors*, each containing a Connection for a selected entity and a string called *RawType* from AutoCAD describing what it is. For example, if the user selected a node, the RawType would be “RcNode”.

EntityDescriptors are also returned by the ThermalDesktop.*GetEntityType* and *GetEntityTypes* methods, which can be used to determine entity types from handles.

The following program demonstrates working with selection sets. It also uses the LINQ *query syntax* (for example: ‘from node in td.GetNodes() where node.Submodel == “MAIN”…’), which is a declarative syntax that can be easier to read than the *method syntax* we have used previously.

```
using System;
using System.Linq;
using OpenTD;
namespace OpenTDGettingStarted
{
    class Selections
    {
        public static void Main(string[] args)
        {
            var td = new ThermalDesktop();
            td.Connect();

            // Create some randomly-placed nodes and lumps:
            var rand = new Random();
            const int numberOfItems = 200;
            for (int i = 0; i < numberOfItems / 2; ++i)
            {
                td.CreateNode(new Point3d(
                    rand.NextDouble(), rand.NextDouble(), rand.NextDouble()));
                var lump = td.CreateLump();
                lump.Origin = new Point3d(
                    rand.NextDouble(), rand.NextDouble(), rand.NextDouble());
                lump.Update();
            }
            td.SetView(IsoViews.SE);

            Console.WriteLine("Using SetSelection to select entities for which" +
                " X < 0.5, Y < 0.5, and Z > 0.5");
            var NodeHandles = from node in td.GetNodes()
                              where node.Origin.X < 0.5
                              && node.Origin.Y < 0.5
                              && node.Origin.Z > 0.5
                              select node.Handle;
            var LumpHandles = from lump in td.GetLumps()
                              where lump.Origin.X < 0.5
                              && lump.Origin.Y < 0.5
                              && lump.Origin.Z > 0.5
                              select lump.Handle;
            var Region1Handles = NodeHandles.Concat(LumpHandles);
            td.SetSelection(Region1Handles);
```

```
            Console.WriteLine($"Selected {NodeHandles.Count()} nodes" +
                $" and {LumpHandles.Count()} lumps. You may" +
                $" have to move your mouse into the AutoCAD window" +
                $" to see the selection.");
            Console.WriteLine("Press [ENTER] to continue...");
            Console.ReadLine();

            Console.WriteLine("Now asking user to select entites.");
            td.ClearSelection();
            var selectedEntities = td.GetSelection();
            var selectedNodeHandles = from entity in selectedEntities
                                      where entity.RawType == "RcNode"
                                      select entity.Connection.Handle;
            var selectedLumpHandles = from entity in selectedEntities
                                      where entity.RawType == "RcLump"
                                      select entity.Connection.Handle;
            Console.WriteLine($"User selected {selectedNodeHandles.Count()}" +
                $" nodes and {selectedLumpHandles.Count()} lumps.");
        }
    }
}
```

4.4. Using Loggers

Using log files to record program operations and errors can be a helpful technique, especially when debugging problems. OpenTD offers access to a powerful logging framework, the same one used by us within TD.

To use the logging framework, you will create an OpenTD.*Logging.Logger* object, then call various methods on it to send messages to log files. The messages have one of the following levels assigned to them, from most to least important: Error, Warning, Information, and Verbose.

Loggers are created using the *LoggerFactory* class, usually as a static member of a class. For example, here is the one we use internally for the ThermalDesktop class:

```
private static readonly Logger log =
    LoggerFactory.GetLogger(typeof(ThermalDesktop).ToString());
```

The LoggerFactory.*GetLogger* method accepts a string representing the name of the logger. This name will be used to route messages from the logger to files. Conventionally, this is the full name of the class that it is defined in, to make it obvious where messages originated. In the above line, we used “typeof(ThermalDesktop).ToString()” instead of the literal string “OpenTD.ThermalDesktop” to ensure that the name is correct with no typos. Did you notice the typo in that literal string?

After calling GetLogger, your log is ready to use. Interestingly, you don't define a log file location within your program. Instead, this is controlled at runtime using the configuration file located at %localappdata%\ThermalDesktop\OpenTDLogConfig.xml. This makes logging more flexible. During debugging on an end-user's computer, you can adjust this configuration file to concentrate on the problem areas in your program.

This is a recent configuration file installed with TD[3]:

```
<?xml version="1.0" encoding="utf-8"?>
<CRLogging>
  <!-- Use this file to configure where OpenTD will send logging messages.
       To make OpenTD use it, copy it to %localappdata%\ThermalDesktop
       and restart OpenTD. -->
  <Loggers>
    <!-- Loggers dispatch messages to their listeners. Logger names correspond to names used
         within OpenTD to categorize messages. Use the RootLogger (below) to configure all loggers
         not listed in this Logger section. Loggers only dispatch messages at and above a
         specified level. -->
    <!-- Levels: Error, Warning, Information, Verbose -->
    <Logger name="OpenTDv61" level="Information" listeners="Log, ErrorLog"/>
    <Logger name="OpenTDv62" level="Information" listeners="Log, ErrorLog"/>
    <Logger name="OpenTDv232" level="Information" listeners="Log, ErrorLog"/>
    <Logger name="OpenTDv232Demos" level="Verbose" listeners="OpenTDv232Demos"/>
    <Logger name="OpenTDMixedInterface" level="Information" listeners="Log, ErrorLog"/>
  </Loggers>
  <!-- The root logger configures all loggers not listed by name in the Loggers section. -->
  <!-- Levels: Error, Warning, Information, Verbose -->
  <RootLogger name="Root" level="Information" listeners="Log, ErrorLog"/>
  <Listeners>
    <!-- Listeners listen for messages from loggers and direct them to a destination,
         typically a file. They only listen for messages at and above a specified level. -->
    <!-- Levels: Error, Warning, Information, Verbose -->
    <!-- Note regarding filenames: the directories above the filename must already exist when
         OpenTD starts, otherwise, the listener will not be used. -->
    <Listener name="Log" type="File" level="Verbose"
              filename="%localappdata%\ThermalDesktop\log\OpenTD.Client.log"/>
    <Listener name="ErrorLog" type="File" level="Error"
              filename="%localappdata%\ThermalDesktop\log\OpenTD.Client.Error.log"/>
    <Listener name="OpenTDv232Demos" type="File" level="Verbose"
              filename="%localappdata%\ThermalDesktop\log\OpenTDv232Demos.log"/>
  </Listeners>
</CRLogging>
```

Two types of objects are defined in the configuration file: loggers and listeners. Loggers emit messages, while listeners route them to destinations. In each case, they only act on messages that are at or above the level they are defined at.

[3] Subject to change. You may find this file in the TD installation directory. If you'd like to use it, copy it to the %localappdata%\ThermalDesktop directory.

You'll note that in the default configuration file, we have defined several loggers for various versions of OpenTD. For example:

```
<Logger name="OpenTD" level="Information" listeners="Log, ErrorLog"/>
```

This means that any logger in an OpenTD client program whose name starts with OpenTD will activate this logger, as long as it is sending a message at the Information level or above. It will send its messages to the listeners called *Log* and *ErrorLog*.

There is also a *RootLogger* defined. This is activated by any loggers whose names do not match the named loggers defined in the <Loggers> block.

There are three listeners defined, corresponding to three output files. Listener Log will write any-level messages to %localappdata%\ThermalDesktop\log\OpenTD.Client.log, while listener ErrorLog will only write Error-level messages to OpenTD.Client.Error.log. And the OpenTDDemos listener will write any-level messages to OpenTDDemos.log.

Using the default configuration file, let's look at some examples of how messages are routed to files.

4.4.1. Example 1

The Logger called "OpenTD.ThermalDesktop" sends an Error-level message:

```
log.LogError("Component A is not working");
```

Logger "OpenTD" defined in the configuration file recognizes this logger because its name starts with "OpenTD", and it routes messages at Information-level and above, so it routes this message to listeners Log and ErrorLog.

Listener Log writes messages at any level, so it writes this message to file OpenTD.Client.log.

Listener ErrorLog writes messages at Error-level, so it writes this message to OpenTD.Client.Error.log.

4.4.2. Example 2

The Logger called “OpenTD.ThermalDesktop” sends a Verbose-level message:

```
log.LogVerbose(“Component A is starting”);
```

Logger “OpenTD” defined in the configuration file recognizes this logger because its name starts with “OpenTD”, but it only routes messages at Information-level and above, so it does not route this message.

4.4.3. Example 3

A logger in a client program is defined with the name “SatelliteBuilder.BusBuilder”:

```
private static readonly Logger log3 =
    LoggerFactory.GetLogger(“SatelliteBuilder.BusBuilder”);
```

It sends an Information-level message:

```
log3.LogInfo(“Applied all optical properties to bus”);
```

No named loggers in the configuration file recognize this logger name, so the RootLogger takes it. Since the RootLogger routes Information-level messages and above, it routes this message to listeners Log and ErrorLog.

Listener Log writes messages at any level, so it writes this message to file OpenTD.Client.log.

Listener ErrorLog writes messages at Error-level only, so it does not write this message.

4.5. Miscellaneous End-User Interaction Techniques

- ThermalDesktop.*ProgressBar* can be used to show a progress bar and message in the TD window during long calculations.
- ThermalDesktop.*Print* prints a message to the AutoCAD console.
- The User Coordinate System (UCS) can be queried or set using the ThermalDesktop.*UCS* property.

- Model Browser visibility can be controlled using the ThermalDesktop.*ShowModelBrowser* and *HideModelBrowser* methods.
- Get a System.Windows.Forms.NativeWindow representing the main AutoCAD window using the ThermalDesktop.*GetMainWindow()* method. NativeWindow implements the IWin32Window interface, so this object can be used anywhere your application requires an IWin32Window that represents the main AutoCAD window.

5. Working with Case Sets

OpenTD gives you full control over case sets. You can create, query, modify, run, or delete them. The following sections show how.

5.1. Create and Run a Case

OpenTD can be used to interact with the Case Set Manager to create and run cases as shown in the following program:

```
using System.Collections.Generic;
using System.IO;
using OpenTD;
namespace OpenTDGettingStarted
{
    class CreateAndRunCase
    {
        public static void Main(string[] args)
        {
            // you may wish to change the location of the working dir:
            string workingDir = @"c:\temp\OpenTDCreateAndRunCase";
            if (Directory.Exists(workingDir))
                Directory.Delete(workingDir, true);
            Directory.CreateDirectory(workingDir);

            var td = new ThermalDesktop();
            td.ConnectConfig.StartDirectory = workingDir;
            td.Connect();

            // *** Create a simple model of a heated bar ***
            var barNodes = new List<Node>();
            for (int i = 0; i < 10; ++i)
            {
                var n = td.CreateNode();
```

```
        n.Submodel = "bar";
        n.Id = i + 1;
        n.MassVol = 10;
        n.Origin = new Point3d(0.01 * i, 1, 0);
        n.InitialTemp = 300;
        n.Update();
        barNodes.Add(n);
    }
    for (int i = 0; i < barNodes.Count - 1; ++i)
    {
        var c = td.CreateConductor(barNodes[i], barNodes[i + 1]);
        c.Submodel = "bar";
        c.Id = i + 1;
        c.Value = 0.1;
        c.Update();
    }

    var roomAir = td.CreateNode();
    roomAir.Submodel = "room";
    roomAir.NodeType = RcNodeData.NodeTypes.BOUNDARY;
    roomAir.Origin = new Point3d(0.055, 1.1, 0);
    roomAir.InitialTemp = 300;
    roomAir.Update();

    var barConnections = new List<Connection>();
    foreach (Node n in barNodes)
        barConnections.Add(new Connection(n));
    var convection = td.CreateConductor(roomAir, barConnections);
    convection.Value = 1;
    convection.Submodel = "room";
    convection.Update();

    var qTorch = td.CreateSymbol("qTorch", "80");
    qTorch.OutputAsRegister = true;
    qTorch.Update();

    var torch = td.CreateHeatLoad(new Connection(barNodes[0]));
    torch.ValueExp.Value = qTorch.Name;
    torch.Submodel = "torch";
    torch.Update();

    td.ZoomExtents();
    // *** End simple model creation ***

    // Create a transient case and run it:
    var nominal = td.CreateCaseSet
        ("transient with nominal torch", "", "torchNom");
    nominal.SteadyState = 0;
    nominal.Transient = 1;
    nominal.SindaControl.timend = 600;
    nominal.Update();
    nominal.Run();

    // Create a cold case by overriding a symbol, and run it:
    var cold = td.CreateCaseSet
        ("transient with cold torch", "", "torchCold");
    cold.SteadyState = 0;
    cold.Transient = 1;
```

```
            cold.SindaControl.timend = 1200;
            cold.SymbolNames.Add(qTorch.Name);
            cold.SymbolValues.Add("50");
            cold.SymbolComments.Add("cold torch heat input");
            cold.SaveAll = 1;
            cold.Update();
            cold.Run();
        }
    }
}
```

Solution files, including “torchNom.sav” and “torchCold.sav”, will be written to the working directory, which we created and then set with ThermalDesktop*.ConnectConfig.StartDirectory*. (See Section 9.1 for more information about controlling how OpenTD connects to TD.) Otherwise, the working directory would have probably been the directory containing your exe. You can also use the ThermalDesktop.*SaveAs* method before running to save the dwg and set the working directory.

5.2. Create an Orbit and Apply it to a Case Set

Orbits are created using the ThermalDesktop.*CreateOrbit* method. Once created, they are applied to case sets by adding a new item to the *CaseSet.RadiationTasks* list. This can also be used to add other types of radiation tasks, such as RADK calculations, as shown in the following program:

```
using OpenTD;
using OpenTD.RadCAD;
namespace OpenTDGettingStarted
{
    class CreateOrbit
    {
        public static void Main(string[] args)
        {
            var td = new ThermalDesktop();
            td.Connect();

            // create an orbit
            var beta30 = td.CreateOrbit("beta30");
            beta30.OrbitType = Orbit.OrbitTypes.BASIC;
            beta30.Planet = Orbit.Planets.EARTH;
            beta30.OrbitData.Inclination = 30;
            beta30.Update();

            // add orbit to a new case set
            var beta30Case = td.CreateCaseSet("beta30 case");
            // first add a RADK task:
            beta30Case.RadiationTasks.Add(
                new RadiationTaskData()
                {
```

```
                    TypeCalc = RadiationTaskData.calcType.RADK,
                    AnalGroup = "BASE",
                });
            // now add the orbital heat rates task:
            beta30Case.RadiationTasks.Add(
                new RadiationTaskData()
                {
                    TypeCalc = RadiationTaskData.calcType.HEATRATE,
                    OrbitName = "beta30",
                    AnalGroup = "BASE",
                });
            beta30Case.Update();
        }
    }
}
```

5.1. Run in Batch Mode

When you call CaseSet.Run(), it is the same as selecting and running a single case in the GUI. If you would like to select multiple cases and run, either in Demand or Batch mode, you can use ThermalDesktop.*CaseSetManager*. It provides all the functionality of the Case Set Manager, including the ability to adjust Manager-level options, using the CaseSetManager.*Options* member.

To run multiple cases – just like selecting them all in the GUI and running – use the following CaseSetManager.Run overload:

```
void Run(IEnumerable<int> caseIndices)
```

In other words, you provide a List<int> or other IEnumerable<int> containing the indices of the cases you would like to run. These indices correspond to the list indices returned by CaseSetManager.*GetCaseSets()* or the equivalent ThermalDesktop.*GetCaseSets()* method.

6. Communicating with SINDA/FLUINT

OpenTD can be used to communicate with and control running SINDA/FLUINT (S/F) solutions using the classes in the OpenTD.*CoSolver* namespace:

- *SF_Launcher* is used to load and run a S/F model directly from an input file, such as an *inp* file created by the TD Case Set Manager. Once launched, the solution proceeds normally with no interaction from the SF_Launcher object.

- *SF_CoSolver* is like an SF_Launcher in that it launches a S/F model from an input file, but once launched it attempts to connect to and control it.

- *TDSF_CoSolver* launches a S/F model from within the TD Case Set Manager, then attempts to connect to and control it.

There is a demo of CoSolver usage available in the OpenTD demos package. (See Section 10.)

7. Reading Results

OpenTD offers two ways to work with solution results:

1. You can work directly with save files, CSR's and other data files. You can create XY plots and tables, or you can simply get the data into memory for further manipulation.

2. You can create and modify contour plots and other post-processing output within a TD instance.

The following examples will show how to use both methods.

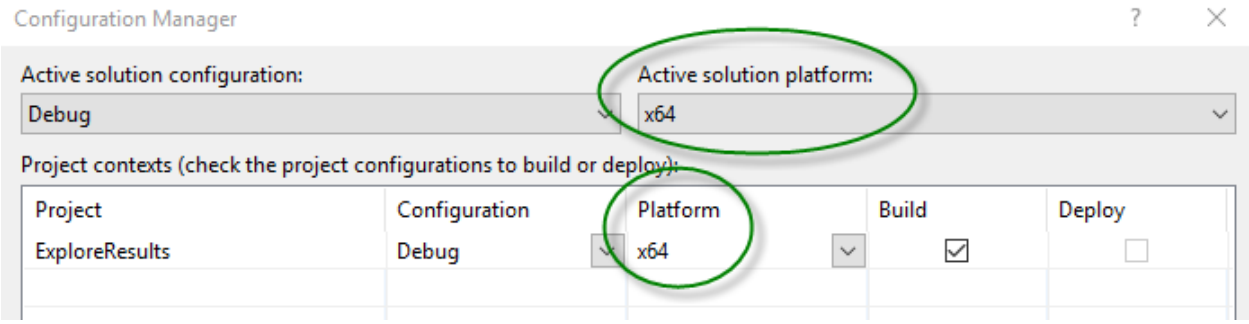
7.1. Work Directly with Results using OpenTD.Results

The classes for working directly with save files, CSR's or other solution results are found within the *OpenTD.Results* namespace, packaged within the OpenTD.Results.dll assembly. The following section describes how to set up a .NET project to access this assembly.

7.1.1. Before using OpenTD.Results

To use OpenTD to explore results directly (rather than via an instance of TD), you will need to add a reference to the OpenTD.Results.dll assembly. You can find it in the GAC in the 64-bit directory. (Try looking under C:\Windows\Microsoft.NET\assembly\GAC_64\OpenTD.Results.) You will also probably need to add a reference to OpenTD.dll, as usual. (See Section 2.1.)

Since OpenTD.Results is a 64-bit assembly, your project will also have to be 64-bit. In Visual Studio, use the Configuration Manager to create a 64-bit solution and a 64-bit project. When set up correctly, your Configuration Manager dialog should look something like this:



7.1.2. The Basics

OpenTD offers powerful tools for manipulating solution results, but it is a lot to learn all at once. This section shows how to simply get some data out of a dataset (a save file, CSR, or saveX file.)

First, reference the OpenTD.Results dll as discussed above. Next, add a using statement for convenience:

```
using OpenTD.Results.Dataset;
```

Now you can connect to a dataset:

```
var myData = DatasetFactory.Load(@"path\to\dataset");
```

To get, for example, all temperatures from a couple of nodes, pass their SINDA/FLUINT names to the *IDataset.GetData* method and call *GetValues(UnitsData u)* on the result:

```
var someNodeTs = myData.GetData("WALL.T10", "WALL.T11").GetValues(Units.SI);
```

This will return a list containing two inner lists of node temperatures in SI units, one for each node. Note that if you request data for just one entity, it will still be returned as a list of lists. For example, the following returns a list containing a single list of flowrates for path "FLOW.60":

```
var onePathFRs = myData.GetData("FLOW.FR60").GetValues(Units.SI);
```

To get the data in other units, you can pass an appropriate *UnitsData* object to GetValues, for example:

```
var someNodeTsEng = myData.GetData("WALL.T10", "WALL.T11").GetValues(Units.Eng);
```

GetData will accept arbitrary-length lists of SINDA/FLUINT names of any type. They can even include registers:

```
var disparateData = myData.GetData(
    "WALL.T10", "FLOW.PL100", "VALVE_POS", "ROOM.T1").GetValues(Units.SI);
```

In this case, GetData will assume that "VALVE_POS" is a dimensionless register and return the raw values from the save/CSR. This behavior can be modified, as discussed in Section 7.1.4.

You can construct an array of strings to pass to GetData. For example, to get temperatures for nodes "MAIN.1" through "MAIN.100", inclusive:

```
string[] someNames = new string[100];
for (int i = 0; i < 100; ++i)
    someNames[i] = "MAIN.T" + (i + 1);
var someTs = myData.GetData(someNames).GetValues(Units.SI);
```

Please note that there are special classes that help to construct lists of entities representing SINDA/FLUINT and TD groups such as submodels and domains. These are introduced in Section 7.1.3 and discussed in detail in Section 7.1.4.

In the examples above, we have extracted 'T', 'PL', and 'FR' data. Please examine the values of the StandardDataSubtypes enum to see all strings recognized by GetData. It also recognizes names that require fluid constituent letters, such as 'GTW'. StandardDataSubtypes are discussed in Sections 7.1.4 and 7.1.5.

To get TIMEN from a dataset, use:

```
var times = myData.GetTimes().GetValues();
```

This returns a list of times in the current WorkingUnits.

7.1.3. Advanced Results Manipulation and XY Plots

The previous section showed how to easily get data out of save files or CSR's. In this and the following sections we will show how to manipulate that data, work with groups and multiple datasets, combine data in arbitrarily complex ways, and create simple XY plots.

Compared to previous methods for extracting and plotting SINDA/FLUINT data, OpenTD offers several improvements:

- The *IDataset* interface and abstract *Dataset* class provide a common syntax for extracting data from save files, CSR's, and other sources. There are several implementations of Dataset that you can use:
  - *SaveFile*
  - *CSR*
  - *SpreadsheetDataFile*
  - *TextTransientFile*
- Data is returned from Datasets via *DataArrays*. In addition to containing the data (T's, Q's, and more), DataArrays know their physical dimension, their units, and contain a reference back to their source Dataset.
- If data is missing from a record, it is expressed in DataArrays as NaN. This happens, for example, when register values are only written to a transient dataset for the beginning and end records.
- *ItemIdentifiers* are used to identify items, such as "MAIN.1" or "FLOW.100".
- *ItemIdentifierCollections* can be constructed based on submodels, domains, or arbitrary lists.

- The *DataSubtype* class describes things like T, Q, TL, GTW, etc. Standard DataSubtypes can be created using the *StandardDataSubtypes* enum or the *FullStandardDataSubtype* struct. (The FullStandardDataSubtype struct is required to describe subtypes that contain a fluid constituent, like GTW.) Custom DataSubtypes can also be created.
- *DataItemIdentifiers* combine ItemIdentifiers and DataSubtypes. They are used to identify data from items, such as "MAIN.T1", "FLOW.TL100", or "FLOW.GTW100".
- *DataItemIdentifierCollections* can be constructed by combining ItemIdentifierCollections with DataSubtypes, or from arbitrary lists.
- *DerivedDatasets* are Datasets that contain references to multiple Datasets and operate on them to return data. Because DerivedDatasets are also Datasets, they can be used anywhere Datasets can be used -- even as input to other DerivedDatasets -- so you can chain them together in arbitrarily complex ways. You can extend the DerivedDataset class to create your own, or use the following pre-defined DerivedDatasets:
  - *ConcatenatedDataset* takes a list of input Datasets and stitches their data together.
  - *DatasetSlice* takes a single Dataset and only returns data between given start and end times.
- *DerivedDataArrays* are like DerivedDatasets but for DataArrays. They contain a list of input DataArrays and operate on them to return data. Just like DerivedDatasets, DerivedDataArrays can be chained together in arbitrarily complex ways, and you can create your own by extending the DerivedDataArray class or use the following pre-defined DerivedDataArrays:
  - *AverageDataArray* returns the arithmetic mean of its input arrays at each record.
  - *ForumlaDataArray* operates on each record of its input arrays using a provided formula, in the units provided.
  - *MaxDataArray* returns the maximum value of its input arrays at each record.
  - *MinDataArray* returns the minimum value of its input arrays at each record.
  - *SelectMaxDataArray* returns the input array with the maximum value.
  - *SelectMinDataArray* returns the input array with the minimum value.
  - *SumDataArray* returns the sum of its input arrays at each record.
  - *WeightedAverageDataArray* returns a weighted average of its input arrays at each record. The weighting factors can be fixed or change with each record. There are helpful constructors to make it easy to, for example, determine mCp-weighted temperatures.
- Plots are intelligent:
  - They automatically put series with different dimensions on different axes.
  - They automatically name themselves and series.

- They automatically label axes with dimensions and units. (They default to the WorkingUnits at the time they are first created.)
- NaN's are displayed as a discontinuity in the series. Series with discontinuities (or steady-state series with only one record) are automatically displayed with markers and lines.
- All the above can be customized.

In this program, we will make use of the "torchNom.sav" and "torchCold.sav" files created in Section 5.1, so please run that program and locate the two save files.

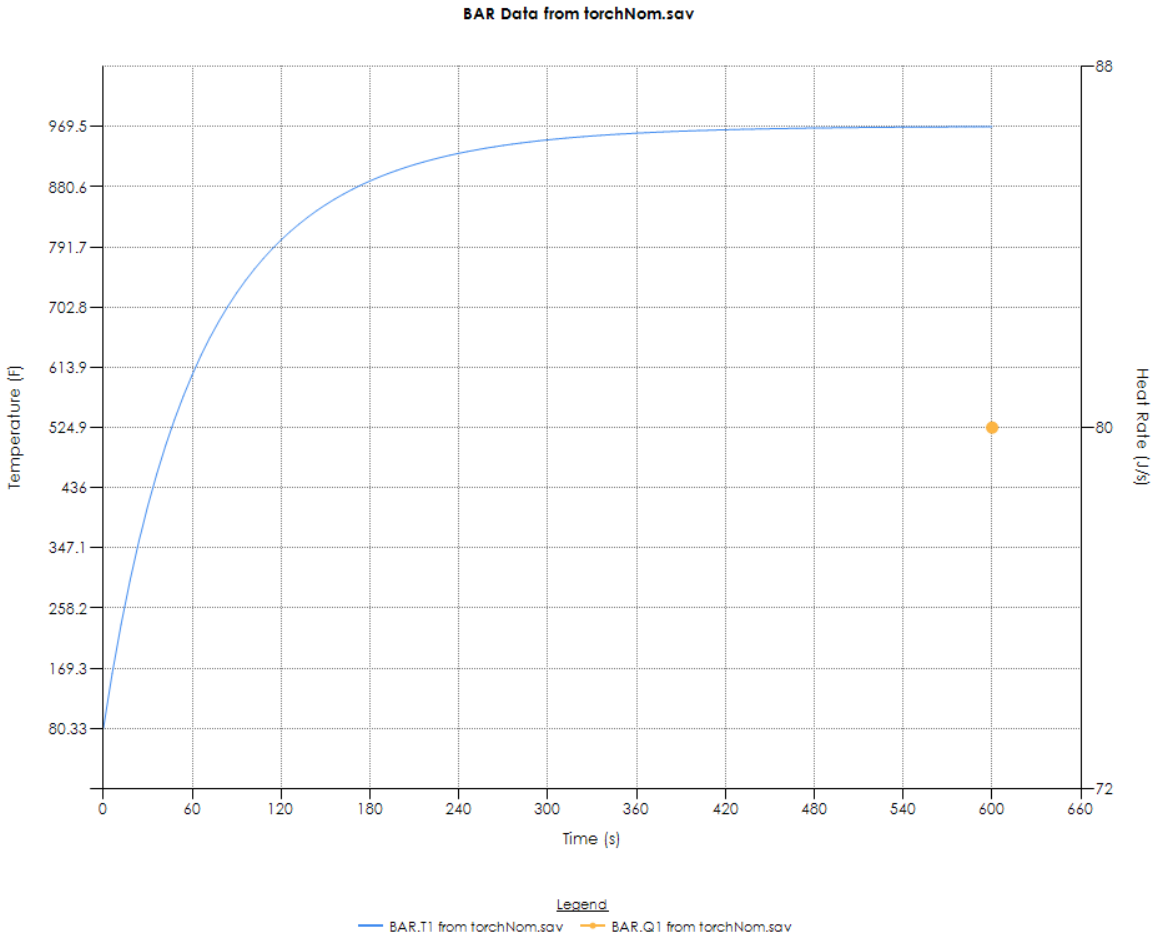
Once Visual Studio is set up with a 64-bit solution and project (Section 7.1.1 above), and you have added references to OpenTD.dll and OpenTD.Results.dll, you will be ready to try the following program.

```
using System;
using OpenTD;
using OpenTD.Results.Dataset;
using OpenTD.Results.Plot;
namespace OpenTDGettingStarted
{
    class ExploreAndPlotResults
    {
        static void Main(string[] args)
        {
            // A Dataset is an abstract class representing solution
            // results. You can create a Dataset from a save file,
            // a CSR, a text file, or from a combination of other
            // Datasets. Let's open the torch save files created
            // in the "Create and Run a Case" example.
            // You may have to change the paths to your copies of
            // torchNom.sav and torchCold.sav.
            var torchNom = DatasetFactory.Load(
                @"c:\temp\OpenTDCreateAndRunCase\torchNom.sav");
            var torchCold = DatasetFactory.Load(
                @"c:\temp\OpenTDCreateAndRunCase\torchCold.sav");

            // Data is returned as DataArrays or DataArrayCollections,
            // which know their physical dimension, their source, and
            // how to return their values in any unit system.
            DataArrayCollection someTemps = torchNom.GetData("BAR.T1", "BAR.T2");

            // DataArray.GetValues returns data in current WorkingUnits
            Units.WorkingUnits.SetToSI();
            Console.WriteLine(
                "BAR.T1 at first time in K: "
                + someTemps[0].GetValues()[0].ToString());

            Units.WorkingUnits.temp = UnitsData.Temp.F;
            Console.WriteLine(
                "BAR.T1 at first time in deg F: "
                + someTemps[0].GetValues()[0].ToString());
```

```
// You can plot data using a SimplePlot:
var plotSomeTemps = new SimplePlot();
plotSomeTemps.AddSeries(someTemps);
plotSomeTemps.Show();
// Note that AddSeries accepted a DataArrayCollection. Since
// each DataArray has a reference to its Dataset, AddSeries is
// able to go and get the time array for X data.

// Let's make a plot with different types of data:
var TandQ = torchNom.GetData("BAR.T1", "BAR.Q1");
var TandQPlot = new SimplePlot();
TandQPlot.AddSeries(TandQ);
TandQPlot.Show();
// Note that since we only saved Q data to the sav
// file for the final record, it is shown as a point.

// Let's plot temperatures for all BAR nodes from both datasets.
// To do this, we're going to use an ItemIdentifierCollection to
// identify all of the BAR nodes.
var barNodes
    = new ItemIdentifierCollection(DataTypes.NODE, "BAR", torchNom);
var barTsNom = torchNom.GetData(barNodes, StandardDataSubtypes.T);
var barTsCold = torchCold.GetData(barNodes, StandardDataSubtypes.T);
var allBarNodesPlot = new SimplePlot();
allBarNodesPlot.AddSeries(barTsNom);
allBarNodesPlot.AddSeries(barTsCold);
allBarNodesPlot.Show();

// That was a lot of series on one plot. Let's plot the average
// bar temps from each case instead. For this we'll use an
// AverageDataArray, which is a type of DerivedDataArray.
// DerivedDataArrays take a collection of DataArrays and
// operate on it to produce a single DataArray.
var avgTNom = new AverageDataArray(barTsNom);
var avgTCold = new AverageDataArray(barTsCold);
var avgPlot = new SimplePlot("Average bar T's");
avgPlot.AddSeries(avgTNom);
avgPlot.AddSeries(avgTCold);
avgPlot.Show();

// There are other types of DerivedDataArrays. Here we'll use
// a SelectMaxDataArray to find the hottest node across both
// datasets:
var allTs = new DataArrayCollection();
allTs.AddRange(barTsNom);
allTs.AddRange(barTsCold);
var maxT = new SelectMaxDataArray(allTs);
var maxPlot = new SimplePlot("Hottest Node");
maxPlot.AddSeries(maxT);
maxPlot.AutoHideLegend = false;
maxPlot.Show();
// We set AutoHideLegend to false because SimplePlot will
// normally hide the legend if there's only one series,
// and use the series name as the title. Since we customized
// the title, that wouldn't work here.

// What if we're only interested in the first 400 s of the
```

```
            // cold case? We can use a DatasetSlice, which is a type of
            // DerivedDataset. DerivedDatasets are to Datasets what
            // DerivedDataArrays are to DataArrays.
            Units.WorkingUnits.time = UnitsData.Time.SEC;
            var coldCaseSlice = new DatasetSlice(torchCold, 0, 400);
            var bar2TCold400 = coldCaseSlice.GetData("BAR.T2");
            var slicePlot = new SimplePlot();
            slicePlot.AddSeries(bar2TCold400);
            slicePlot.Show();
            // Note that if there isn't a record at 400 s, DatasetSlice
            // will choose the next record.
        }
    }
}
```

If it runs correctly, you should see several *SimplePlot*-generated dialogs like the following. Instead of displaying them to the screen with SimplePlot.*Show()*, we could have used .*GetAsImage* or .*GetAsTable* to return objects that could be further manipulated in our program, or .*SaveAsImage* or .*SaveAsTable* to write image or csv files containing the data.



7.1.4. Work with Groups in Results Data

The previous example provided an overview of using OpenTD to read and plot SINDA/FLUINT results. In the following example we will discuss ItemIdentifiers and DataItemIdentifiers, which allow you to specify groups of items, whether by submodel, domain, or arbitrary list.

Once again, we are using the OpenTD.Results dll, so create a 64-bit project as discussed in Section 7.1.

```
using System;
using System.IO;
using System.Collections.Generic;
using OpenTD;
using OpenTD.Results.Dataset;
using OpenTD.Results.Plot;

namespace OpenTDGettingStarted
{
    class WorkingWithGroups
    {
        public static void Main(string[] args)
        {
            // Let's open one of the torch save files from the
            // "Create and Run a Case" example.
            // You may need to change the workingDir string to the
            // dir containing torchNom.sav.
            string workingDir = @"c:\temp\OpenTDCreateAndRunCase";
            var data = DatasetFactory.Load(Path.Combine(workingDir, "torchNom.sav"));

            #region ItemIdentifiers
            // Use ItemIdentifiers to identify SINDA/FLUINT entities
            // like WALL.100, WATER.45, or MY_REGISTER.

            // You can create a collection of ItemIdentifiers from a submodel in a
            // Dataset:
            var allbarNodeNames
                = new ItemIdentifierCollection(DataTypes.NODE, "BAR", data);
            Console.WriteLine("\nItemIdentifierCollection: BAR submodel:");
            foreach (var item in allbarNodeNames)
                Console.WriteLine("{0} => Submodel = '{1}', Id = {2}",
                    item, item.Submodel, item.Id);

            // You can create a collection of ItemIdentifiers representing the
            // nodes in a domain, but you need to supply a ThermalDesktop instance
            // to define the domain: (this example is commented out to avoid loading
            // a TD instance.)
            // var allSomeDomainNodes
            //    = new ItemIdentifierCollection("SOME_DOMAIN", aTdInstance);

            // You can create arbitrary collections of ItemIdentifiers by supplying
            // a list of SINDA names:
            var arbitraryItems = new ItemIdentifierCollection(
                "MAIN.1",
```

```
    "FLOW.1O",
    "MY_REGISTER");
Console.WriteLine("\nItemIdentifiers: arbitrary list:");
foreach (var item in arbitraryItems)
    Console.WriteLine("{0} => Sub = '{1}', Id = {2}, RegName = '{3}'",
        item, item.Submodel, item.Id, item.RegisterName);
#endregion

#region DataSubtypes
// Use DataSubtypes to identify types of SINDA/FLUINT data,
// like T, TL, GTW, etc.

// You can create a new DataSubtype with the StandardDataSubtypes enum:
var SubtypeT = new DataSubtype(StandardDataSubtypes.T);

// For multispecies data, you can provide a StandardDataSubtypes
// and a FluidConstituents enum...
var SubtypeGTW = new DataSubtype(
    StandardDataSubtypes.GT, FluidConstituents.W);

// ...or you can provide a FullStandardDataSubtypes struct,
// which just contains a StandardDataSubtypes/FluidConstituents pair:
var SubtypeGTW_alternate
    = new DataSubtype(new FullStandardDataSubtype(
        StandardDataSubtypes.GT, FluidConstituents.W));

// You can also create your own DataSubtype, maybe for custom data
// you're reading out of a text file:
var SubtypeBoundaryLayerThickness = new DataSubtype(
        dimension: UnitsData.UnitsType.MODEL_LENGTH,
        isDimensionalOnlyIfPositive: false,
        description: "Boundary Layer Thickness for Phase B",
        baseSindaDesignator: null,
        dataType: DataTypes.PATH);

// Through the magic of implicit casts, you can use a
// StandardDataSubtypes or FullStandardDataSubtype anywhere
// a DataSubtype is expected:
var listOfDataSubtypes = new List<DataSubtype>() {
    StandardDataSubtypes.T,
    new FullStandardDataSubtype(
        StandardDataSubtypes.GT, FluidConstituents.W) };
Console.WriteLine("\nDataSubtypes: implicit casts to DataSubtype:");
foreach (var item in listOfDataSubtypes)
    Console.WriteLine(item);
#endregion

#region Get data with DataSubtypes and ItemIdentifiers
// Datasets can return data for multiple items if provided an
// ItemIdentifierCollection and a DataSubtype:
var allBarNodeTs = data.GetData(allbarNodeNames, StandardDataSubtypes.T);
Console.WriteLine("\nGet data with DataSubtypes and ItemIdentifiers:\n" +
    "DataArrayCollection.Name: {0}\n.Count: {1}\n.DataItemCount: {2}",
    allBarNodeTs.Name, allBarNodeTs.Count, allBarNodeTs.DataItemCount);
#endregion

#region DataItemIdentifiers
// Use DataItemIdentifiers to identify SINDA/FLUINT entities like
```

```
// WALL.T100, WATER.TL45, or "MY_REGISTER temperature".
// They combine ItemIdentifiers and DataSubtypes.

// You can create DataItemIdentifiers from ItemIdentifiers and a single
//DataSubtype:
var allBar_T_Names
    = new DataItemIdentifierCollection(
        allbarNodeNames, StandardDataSubtypes.T);
Console.WriteLine("\nDataItemIdentifiers: all BAR T names:");
foreach (var item in allBar_T_Names)
    Console.WriteLine(item);

// Or you can create a DataItemIdentifierCollection
// by parsing a list of SINDA/FLUINT names:
var parsedDataItems = new DataItemIdentifierCollection(
    "MAIN.T100",
    "FLOW.FR40",
    "FLOW.GTW11",
    "A_REGISTER");
Console.WriteLine("\nDataItemIdentifiers: parsed SINDA/FLUINT names:");
foreach (var item in parsedDataItems)
    Console.WriteLine(item);

// One issue with parsing SINDA/FLUINT names: it will assume any
// registers are dimensionless, so if you'd like to assign a
// dimension to a register, you should explicitly create the
// DataItemIdentifier for it:
var A_REGISTER_flowrate
    = new DataItemIdentifier(
        new ItemIdentifier("A_REGISTER"), StandardDataSubtypes.FR);

// If the register stores data in a unit system other than that
// of the Dataset, you can specify that too:
var A_REGISTER_flowrate_Eng
    = new DataItemIdentifier(
        new ItemIdentifier("A_REGISTER"), StandardDataSubtypes.FR,
        Units.Eng);

// You can combine ItemIdentifiers and a list of DataSubtypes,
// creating all combinations of DataItemIdentifiers:
var TandQ = new List<DataSubtype>()
{
    StandardDataSubtypes.T,
    StandardDataSubtypes.Q
};
var allBar_TandQ_Names
    = new DataItemIdentifierCollection(allbarNodeNames, TandQ);
Console.WriteLine("\nDataItemIdentifiers: " +
    "combine ItemIdentifiers and DataSubtypes:");
foreach (var item in allBar_TandQ_Names)
    Console.WriteLine(item);
#endregion

#region Get data with DataItemIdentifiers
// Datasets can return data for multiple items if provided a
// DataItemIdentifierCollection:
var allBar_TandQ = data.GetData(allBar_TandQ_Names);
```

```
            // And you can plot all the series in a DataArrayCollection
            // with a single SimplePlot.AddSeries method:
            {
                var plot = new SimplePlot();
                plot.Name
                    = "Get data with DataItemIdentifiers: plot DataArrayCollection";
                plot.AddSeries(allBar_TandQ);
                plot.Show();
            }
            // Since the DataArrays in allBar_TandQ know their SourceDataset,
            // AddSeries uses it to get time for the x data of each series.

            // Skip some steps and let GetData parse a list of SINDA/FLUINT names.
            // One issue with this: it will assume any registers are dimensionless,
            // so if you'd like to assign a dimension to a register, you should
            // explicitly create the DataItemIdentifier for it.
            var someData = data.GetData("BAR.T1", "BAR.Q1", "BAR.T3");
            {
                var plot = new SimplePlot();
                plot.Name = "Get data with DataItemIdentifiers: parsed names";
                plot.AddSeries(someData);
                plot.Show();
            }
            #endregion
        }
    }
}
```

7.1.5. A Note on DataSubtypes

As shown in the preceding sections, the DataSubtype class is used to describe types of data found in solution results such as “node temperature T” or “lump pressure PL”. There are several methods that require a DataSubtype as a parameter. When using those methods, you can create your own DataSubtype, but it is usually easier to allow OpenTD to do it for you by making use of the StandardDataSubtypes enum and/or the FullStandardDataSubtype struct.

Through the magic of implicit casting (see Section 9.4) any method that accepts a DataSubtype will accept either a StandardDataSubtypes or FullStandardDataSubtype instead. For example, one of the GetData overloads has the following signature:

```
DataArrayCollection GetData(
    ItemIdentifierCollection itemIds, DataSubtype subtype, UnitsData units = null)
```

It expects a list of entity names (the ItemIdentifierCollection), the type of data to get (the DataSubtype), and you can optionally specify the units for any registers in the ItemIdentifierCollection. To get node temperatures for all the nodes in the “PANEL” domain from TD instance “td”, you could do this:

```
var panelNodes = new ItemIdentifierCollection("PANEL", td);
var panelNodeTs = myData.GetData(panelNodes, StandardDataSubtypes.T);
```

We passed "StandardDataSubtypes.T" even though the method expected a DataSubtype. If you are using a language that does not support .NET implicit casting, you can explicitly construct the DataSubtype instead:

```
var panelNodes = new ItemIdentifierCollection("PANEL", td);
var panelNodeTs = myData.GetData(panelNodes, new DataSubtype(StandardDataSubtypes.T));
```

Implicit casting to a DataSubtype also works for FullStandardDataSubtypes, which are structs that combine StandardDataSubtypes with a fluid constituent.

7.1.6. Get Model Topology from Solution Results

As demonstrated in Section 3, you can query a dwg file to determine the topology of a model. For example, you can use ThermalDesktop.*GetConductors()* to get all of the conductors, then for each you can use the *From* and *To* members and GetNode to find the attached nodes.

That approach is suitable when you are working with the dwg file, but if you are just using solution results it is often more convenient to read topology data directly without involving TD. Also, model topology can change during the run due to BUILD statements or other techniques, so reverse-engineering the actual topology at a given record from the dwg file is error-prone.

Fortunately, OpenTD allows you to read the actual as-solved model topology at each record using *PCS files* automatically created by SINDA/FLUINT. You may have seen these files before in your solution directory, often named "MyCase.savPCS" or similar.

The following program will read model topology from a PCS file and print out all the conductors and the nodes they connect. It uses "torchNom.sav" and "torchNom.savPCS", created in Section 5.1, so please run that program and locate the two files.

```
using System;
using System.Collections.Generic;
using System.IO;
using OpenTD.Results.Dataset;
using OpenTD.Results.Dataset.Topology;
namespace OpenTDGettingStarted
{
    class GettingModelTopology
```

```
    {
        public static void Main(string[] args)
        {
            // Let's open one of the torch save files from the
            // "Create and Run a Case" example.
            // You may need to change the workingDir string to the
            // dir containing torchNom.sav.
            string workingDir = @"c:\temp\OpenTDCreateAndRunCase";
            var data = DatasetFactory.Load(Path.Combine(workingDir, "torchNom.sav"));

            // Now we'll read the PCS file, which contains topology and other
            // extra information. Topology can change with each record, so we
            // have to specify which record. We'll just use the first one:
            List<long> recordNums = data.GetRecordNumbers();
            string pcsPath = Path.Combine(workingDir, "torchNom.savPCS");
            IDatasetTopology topology
                = DatasetTopology.Load(data, recordNums[0], pcsPath);

            // Find all conductors and their attached nodes:
            foreach (IConductorInfo cond in topology.Conductors)
            {
                Console.WriteLine("{0} Conductor {1}:",
                    cond.IsRad ? "Radiation" : "Linear", cond.SindaName);
                Console.WriteLine("  From {0} Node {1}",
                    cond.FromNode.NodeType, cond.FromNode.SindaName);
                Console.WriteLine("  To {0} Node {1}",
                    cond.ToNode.NodeType, cond.ToNode.SindaName);
            }
        }
    }
}
```

You may notice that each of the conductors is listed twice, once from node A to B, and once from B to A. This is an indication that the conductor is a normal two-way conductor. One-way conductors will only show up once.

7.1.7. Calculate Heat Rates between Groups

Model topology along with solution results can be used to determine heat rates between groups of entities. This is done using an *IBrowser*.

The following program uses the solution results from Section 5.1. Make sure to run that program first before trying this one. As you may recall, that program creates a model of a bar heated on one end, convecting to a room along its length. We will be using results from a transient case, starting with a cold bar.

We will create two IBrowsers, one early in the solution and one at the final record, using them to find the total heat rate from the bar to the room at the two times. We will also use a

SumDataArray (Section 7.1.3) to plot the constant total external heat into the bar over the whole solution. We expect the heat rate from the bar to the room to approach the external heat rate as the system approaches steady-state.

```
using System;
using System.Collections.Generic;
using System.IO;
using OpenTD.Results.Dataset;
using OpenTD.Results.Dataset.Topology;
using OpenTD.Results.Plot;
namespace OpenTDGettingStarted
{
    class CalculatingHeatRates
    {
        public static void Main(string[] args)
        {
            // Open torchCold.sav and torchCold.savPCS, getting topology at
            // early record: (using torchCold because we set SaveAll = 1 to
            // make sure all data req'd for Browser available)
            string workingDir = @"c:\temp\OpenTDCreateAndRunCase";
            var data = DatasetFactory.Load(Path.Combine(workingDir, "torchCold.sav"));
            List<long> recordNums = data.GetRecordNumbers();
            List<double> timesSec = data.GetTimes().GetValues();
            string pcsPath = Path.Combine(workingDir, "torchCold.savPCS");
            int earlyIndex = 1;
            IDatasetTopology topologyEarly
                = DatasetTopology.Load(data, recordNums[earlyIndex], pcsPath);

            // Create Browser at early record and find heat rate from
            // submodel BAR to ROOM:
            IBrowser browserEarly
                = Browser.Create(data, topologyEarly, recordNums[earlyIndex]);
            HeatratesBetween heatratesEarly
                = browserEarly.GetHeatBetweenSubmodels("BAR", "ROOM");
            Console.WriteLine("At time {0} s, heat rate from BAR to ROOM: {1} W",
                timesSec[earlyIndex], heatratesEarly.TotalHeatrate);

            // Find heat rate between submodels at final record:
            // (Using early-record topology because we know it
            // doesn't change in this model.)
            int finalIndex = recordNums.Count - 1;
            IBrowser browserFinal
                = Browser.Create(data, topologyEarly, recordNums[finalIndex]);
            HeatratesBetween heatratesFinal
                = browserFinal.GetHeatBetweenSubmodels("BAR", "ROOM");
            Console.WriteLine("At time {0} s, heat rate from BAR to ROOM: {1} W",
                timesSec[finalIndex], heatratesFinal.TotalHeatrate);

            // Plot total external heat into bar:
            var barNodes = new ItemIdentifierCollection(DataTypes.NODE, "BAR", data);
            DataArrayCollection barQs
                = data.GetData(barNodes, StandardDataSubtypes.Q);
            var barQSum = new SumDataArray(barQs);
            var plot = new SimplePlot();
            plot.AddSeries(barQSum);
            plot.Show();
        }
```

```
    }
}
```

For this program, we only looked at heat rates between submodels, using IBrowser.*GetHeatBewteenSubmodels*. But any arbitrary group of nodes or ties can be examined using the IBrowser.*GetHeatBetween* method.

7.1.8. Read Text Files

In addition to SaveFile and CSR data sources, there are other types of Dataset classes available:

- SpreadsheetDataFile, for reading columns of comma-delimited data
- TextTransientFile, for reading the TD-specific Text Transient Dataset file, as defined in the TD manual.

All of them implement the OpenTD.Dataset.IDataset interface, so they can be used interchangeably. The following program demonstrates treating a csv file as a Dataset using OpenTD.Results:

```
using System;
using System.IO;
using OpenTD;
using OpenTD.Results.Dataset;
using OpenTD.Results.Plot;
namespace OpenTDGettingStarted
{
    class ReadSpreadsheetFile
    {
        public static void Main(string[] args)
        {
            // you may wish to change the location of the working dir:
            string workingDir = @"c:\temp\ReadTextFiles";
            if (Directory.Exists(workingDir))
                Directory.Delete(workingDir, true);
            Directory.CreateDirectory(workingDir);

            // We'll create a spreadsheet file to read:
            var pathname = Path.Combine(workingDir, "testData.csv");
            var spreadsheetData
                = "TIMEN,WALL.T100,WALL.T200,INCIDENT_FLUX,COOLANT.FR10\n"
                + "hr,,,W/m^2,kg/min\n"
                + "0,70,68,100,0.5\n"
                + "1,75,69,102,0.41\n"
                + "2,77,,101,0.43\n"
                + "3,NaN,70,102,0.52\n"
                + "4,78.5,70,103,0.5";
            File.WriteAllText(pathname, spreadsheetData);
```

```
// The file is comma-delimited. By default, it can be comma- or
// tab-delimited. Other delimiters can be chosen by adjusting the
// Delimiters array. Space cannot be used, because it can be used as part
// of certain units expressions.

// The first row holds names. SINDA/FLUINT names like MAIN.T1 or
// FLOW.GTX10 are valid. Other strings will be assumed to be register
// names.

// The second row holds optional units. For blank cells,
// SpreadsheetDataFile.Units will be assumed. All registers except TIMEN
// will be assumed to be dimensionless, and their units will be ignored.

// For this example, the file represents observations of a heated and
// cooled wall, taken every hour. Time was measured in hours,
// temperatures in deg F, heating flux in W/m^2, and coolant flowrate
// in kg/min. The observer noted units on the second row for everything
// except the temperatures. The observer also missed taking two of the
// data readings, with one listed as NaN and one listed as an empty cell.

// First, we'll create the Dataset from the spreadsheet file:
var data = new SpreadsheetDataFile(pathname);

// We'll set the Dataset temperature units to F since the temperature
// units aren't listed in the file:
data.Units.temp = UnitsData.Temp.F;

// We can use the Dataset methods to extract data from
// SpreadsheetDataFiles:
Console.WriteLine("Thermal submodels:");
foreach (string submodel in data.GetThermalSubmodels())
    Console.WriteLine("  " + submodel);
Console.WriteLine("Paths in submodel COOLANT:");
foreach (long id in data.GetPathIds("COOLANT"))
    Console.WriteLine("  " + id);

// Let's get heating and cooling data:

// Most registers are assumed dimensionless, so we'll have to tell
// the dataset that INCIDENT_HEAT is a flux. First, let's create a
// custom DataSubtype representing heat flux:
var heatFlux = new DataSubtype(
        dimension: UnitsData.UnitsType.FLUX,
        description: "heat flux");

// Now let's get the INCIDENT_HEAT data: (We'll use GetRegisterData so we
// can assign it a DataSubtype and state what units the values are in.)
var incident_heat = data.GetRegisterData(
    "INCIDENT_FLUX", heatFlux, Units.SI);

// The coolant flowrate is easier, since the dataset already knows its
// DataSubtype and units:
var coolantFR10 = data.GetData("COOLANT.FR10");

// Now let's create some plots, in SI units:
Units.WorkingUnits.SetToSI();

var inputPlot = new SimplePlot("Heating and Cooling Applied to Wall");
```

```
            inputPlot.AddSeries(incident_heat);
            inputPlot.AddSeries(coolantFR10);
            inputPlot.Show();

            var wallTempPlot = new SimplePlot();
            wallTempPlot.AddSeries(data.GetData("WALL.T100", "WALL.T200"));
            wallTempPlot.Show();
        }
    }
}
```

The following program treats a TD Text Transient Data file as a Dataset:

```
using System;
using System.IO;
using OpenTD;
using OpenTD.Results.Dataset;
using OpenTD.Results.Plot;
namespace OpenTDGettingStarted
{
    class ReadTextTransientFile
    {
        public static void Main(string[] args)
        {
            // you may wish to change the location of the working dir:
            string workingDir = @"c:\temp\ReadTextFiles";
            if (Directory.Exists(workingDir))
                Directory.Delete(workingDir, true);
            Directory.CreateDirectory(workingDir);

            // We'll create a text transient file to read:
            var pathname = Path.Combine(workingDir, "testData.txt");
            var angleData = "# Case 0 Motor Angle\n"
                          + "#\n"
                          + "#TIME_UNITS hr\n"
                          + "#\n"
                          + "Label=Angle\n"
                          + "3\n"
                          + "MOTOR.69\n"
                          + "MOTOR.70\n"
                          + "MOTOR.71\n"
                          + "1.\n"
                          + "53.7093946669\n"
                          + "53.7093946669\n"
                          + "53.2207743327\n"
                          + "2.\n"
                          + "60.\n"
                          + "65.\n"
                          + "70.";
            File.WriteAllText(pathname, angleData);

            // File specification:
            // 1. Optional arbitrary number of comment lines beginning with #
            // 2. Optional #TIME_UNITS line
            //      * Must be only one space between TIME_UNITS and value
            //      * First character of value is compared against first
            //        character of Time enum values SEC, MIN or HR
```

```
            // 3. Optional arbitrary number of comment lines beginning with #
            // 4. Optional Label = line
            //      * Label= does not need to appear at start of line
            //      * Value is 2048 characters max
            // 5. Integer specifying number of items n
            //      * 2048 characters max
            // 6. n lines containing item names
            //      * Throw away spaces, \n, and \r
            //      * If no '.', prepend 'MAIN.' to name
            //      * 2048 characters max
            // 7. Single line containing first time
            //      * 2048 characters max
            // 8. n lines containing data for that time
            //      * 2048 characters max


            // We'll use a custom DataSubtype to describe the motor angle data
            // stored in the text transient file. The description "motor crank
            // angle" will be overwritten by the "Label =" line in the example
            // text file.
            var motorAngle = new DataSubtype(UnitsData.UnitsType.ANGLE,
                    false, "motor crank angle", null,
                    DataTypes.NODE);

            // If we didn't provide a DataSubtype to the TextTransientFile
            // constructor, it would assume T (node temperature). If we didn't
            // provide units, it would assume the data was in the current
            // Units.WorkingUnits. (With the exception of time units, which can be
            // overwritten by a #TIME_UNITS statement in the file.
            var data = new TextTransientFile(pathname, motorAngle, Units.SI);

            // You can use the Dataset methods to extract data from
            // TextTransientFiles
            Console.WriteLine("Thermal submodels:");
            foreach (string submodel in data.GetThermalSubmodels())
                Console.WriteLine("  " + submodel);
            Console.WriteLine("Nodes in submodel MOTOR:");
            foreach (long id in data.GetNodeIds("MOTOR"))
                Console.WriteLine("  " + id);

            // In addition to the Dataset methods TextTransientFiles also implement
            // the ISimpleDataset interface, which includes the GetAllData() method.
            // This returns a DataArrayCollection all of the data in the
            // SimpleDataset. It defaults to returning time as the first array, but
            // we'll override that behavior here with the includeXDataAsFirstArray
            // parameter.
            var allData = data.GetAllData(includeXDataAsFirstArray: false);

            // Find and plot max motor angle in radians
            var maxMotorAngle = new MaxDataArray(allData);
            var plot = new SimplePlot();
            plot.AddSeries(maxMotorAngle);
            plot.AutoSetAxes();
            plot.YAxes[0].Units.angle = UnitsData.Angle.RADIANS;
            plot.Show();
        }
    }
}
```

7.1.9. Compare Datasets

The *Comparer* class can be used to determine if two Datasets are the same. By default, it compares the following:

- Number of records
- Max and min times
- Thermal submodel names
- Node names
- Fluid submodel names
- Lump names
- Path names
- All T data for node names that are common between datasets
- All TL data for common lumps
- All PL data for common lumps
- All FR data for common paths

Any of the above can be excluded, and the following can be added to the comparison:

- Conductor names
- Tie names
- FTie names
- IFace names
- Any of the hundreds of StandardDataSubtypes or FullStandardDataSubtypes (by adding them to the *DataToCompare* member) for the relevant common entities.

All floating point data is compared in SI units with a default tolerance of 1%. The tolerance can be adjusted by changing the *DefaultComparisonMethod.PercentTol* member. Any items that exceed tolerance will be saved in the *Exceedances* member and can be plotted using the *PlotExceedances* method.

CompareAssertions can be used to store a Comparer along with an assertion about whether the two Datasets should be identical.

CompareSuites can hold a collection of CompareAssertions, run all comparisons, and report on the success or failure of all assertions. The following program demonstrates the use of a simple CompareSuite:

```
using System;
using System.IO;
using OpenTD.Results.Dataset;
namespace OpenTDGettingStarted
{
    class CompareDatasets
    {
        public static void Main(string[] args)
        {
            // you may wish to change the location of the working dir:
            string workingDir = @"c:\temp\CompareDatasets";
            if (Directory.Exists(workingDir))
                Directory.Delete(workingDir, true);
            Directory.CreateDirectory(workingDir);

            #region Create Files

            // for clarity, we'll just create a few csv files and read them
            // in as SpreadsheetFiles, but of course Comparers can be used
            // with any kind of Dataset, including sav files and CSR's

            // file 1: (baseline)
            string pathBaseline = Path.Combine(workingDir, "baseline.csv");
            string dataBaseline
                = "TIMEN,WALL.T100\n"
                + "min,F\n"
                + "0,70\n"
                + "1,75\n"
                + "2,77";
            File.WriteAllText(pathBaseline, dataBaseline);

            // file 2: (identical to baseline)
            string pathCopy = Path.Combine(workingDir, "baselineCopy.csv");
            File.Copy(pathBaseline, pathCopy);

            // file 3: (add extra node)
            string pathExtraNode = Path.Combine(workingDir, "extraNode.csv");
            string dataExtraNode
                = "TIMEN,WALL.T100,MAIN.T4\n"
                + "min,F,F\n"
                + "0,70,32\n"
                + "1,75,32\n"
                + "2,77,32";
            File.WriteAllText(pathExtraNode, dataExtraNode);

            // file 4: (same as baseline, but different time units)
            string pathDifferentUnits = Path.Combine(
                workingDir, "differentUnits.csv");
            string dataDifferentUnits
                = "TIMEN,WALL.T100\n"
                + "s,F\n"
                + "0,70\n"
                + "60,75\n"
```

```
            + "120,77";
        File.WriteAllText(pathDifferentUnits, dataDifferentUnits);

        // file 5: (different temperatures)
        string pathDifferentT = Path.Combine(workingDir, "differentT.csv");
        string dataDifferentT
            = "TIMEN,WALL.T100\n"
            + "min,F\n"
            + "0,50\n"
            + "1,52\n"
            + "2,53";
        File.WriteAllText(pathDifferentT, dataDifferentT);

        #endregion

        // Read all of the files as Datasets:
        var baseline = new SpreadsheetDataFile(pathBaseline);
        var copy = new SpreadsheetDataFile(pathCopy);
        var extraNode = new SpreadsheetDataFile(pathExtraNode);
        var differentUnits = new SpreadsheetDataFile(pathDifferentUnits);
        var differentT = new SpreadsheetDataFile(pathDifferentT);

        // Create a CompareSuite with our assertions: (we'll get one
        // wrong on purpose)
        var suite = new CompareSuite()
        {
            new CompareAssertion(baseline, copy, assertDatasetsSame: true),
            new CompareAssertion(baseline, extraNode, true), // wrong
            new CompareAssertion(baseline, differentUnits, true),
            new CompareAssertion(baseline, differentT, false),
        };

        Tuple<int, int> SuccessAndTotal = suite.Run();
        Console.WriteLine("{0} out of {1} assertions were true.",
            SuccessAndTotal.Item1, SuccessAndTotal.Item2);
        Console.WriteLine();
        Console.WriteLine("Full log output:");
        Console.WriteLine();
        Console.Write(suite.Log);

        // Let's use an exceedance plot to see how different baseline
        // and differentT are:
        var compareToDifferentT = new Comparer(baseline, differentT);
        compareToDifferentT.Run();
        compareToDifferentT.PlotExceedances();
        Console.WriteLine();
        Console.WriteLine("Running baseline to differentT comparison again" +
            " and plotting exceedance plot:");
        Console.WriteLine();
        Console.WriteLine(compareToDifferentT.Message);
    }
  }
}
```

7.1.10. Use Different Comparison Algorithms

The Comparer class allows you to control how different types of data will be compared. By default, all data is compared by calculating the percent difference between dataset values for each common-named entity (MAIN.1, etc.) at each record, with a tolerance of 1%.

The default comparison algorithm is contained within the *Comparer.DeafultComparisonMethod* member, which is an object that implements the *ICompareData* interface.

When you create a new Comparer, DefaultComparisonMethod is set to a new *PercentDifferenceCompareData*, that is, an object that implements the ICompareData interface and will calculate the percent difference between record values as discussed above.

To change the comparison algorithm used for specific data types (T, PL, etc.), you can use the *Comparer.ComparisonMethods* dictionary. Its keys are StandardDataSubtypes and its values are objects that implement the ICompareData interface. For example, to change the tolerance for PL and to use a custom method to compare FR, you could do something like this:

```
Var c = new Comparer(…)
c.ComparisonMethods = new Dictionary<StandardDataSubtypes, ICompareData>
{
      {
            StandardDataSubtypes.PL, new PercentDifferenceCompareData(c, c)
            {
                  PercentTol = 2.5,
            }
      },
      {
            StandardDataSubtypes.FR, new CustomCompareData(c, c)
      },
}
```

We passed the Comparer object c to the PercentDifferenceCompareData constructor because it needs to know where to get input and send output. This is accomplished using the *ICompareInput* and *ICompareOutput* members of the ICompareData interface. The Comparer implements these input/output interfaces, so we can pass it to the PercentDifferenceCompareData constructor to connect it.

Details of the ICompareData, ICompareInput, and ICompareOutput interfaces can be found in the “OpenTD Class Reference” document. (See Section 10.)

The source code for the PercentDifferenceCompareData class can be found at https://www.crtech.com/forum/topic/percentdifferencecomparedata-source-code-see-getting-started-guide-explanation , to use as an example for creating custom ICompareData algorithms.

7.2. Work with Datasets in TD using OpenTD.PostProcessing

To work with solution results within Thermal Desktop, use the OpenTD.PostProcessing namespace, specifically the ThermalDesktop.*DatasetManager*. This gives you the same functionality as the "Postprocessing Datasets" dialog in the GUI.

7.2.1. Create Contour Plots

The following program will create and run a simple model, use the OpenTD.Results namespace to find when the max mCp-weighted temperature of a component occurs, then use the DatasetManager to create a temperature contour at that time with TD. It will also display an XY plot of the mCp-weighted temperature, to confirm that the correct time was selected. This barely scratches the surface of what you can do with the DatasetManager!

```
using System.IO;
using System.Linq;
using OpenTD;
using OpenTD.Dimension;
using OpenTD.Results.Dataset;
using OpenTD.Results.Plot;
namespace OpenTDGettingStarted
{
    class CreateContourPlots
    {
        public static void Main(string[] args)
        {
            // you may wish to change the location of the working dir:
            string workingDir = @"c:\temp\CreateContourPlots";
            if (Directory.Exists(workingDir))
                Directory.Delete(workingDir, true);
            Directory.CreateDirectory(workingDir);

            var td = new ThermalDesktop();
            td.ConnectConfig.StartDirectory = workingDir;
            td.Connect();

            #region Create and run model

            td.CreateThermoProps("dummy");

            var disk = td.CreateDisk();
            disk.BreakdownU.Num = 10;
            disk.BreakdownV.Num = 10;
```

```
disk.TopMaterial = "dummy";
disk.TopStartSubmodel = "MYDISK";
disk.Update();

var hotNode = td.GetNode(disk.AttachedNodeHandles[50]);
var coldNode = td.GetNode(disk.AttachedNodeHandles[75]);

hotNode.NodeType = RcNodeData.NodeTypes.BOUNDARY;
hotNode.UserOverride = true;
hotNode.UseVersusTime = 1;
hotNode.TimeArray = new DimensionalList<Time>
    { 0, 10, 20, 30, 40, 50 };
hotNode.ValueArray = new DimensionalList<Temp>
    { 500, 480, 460, 480, 490, 480 };
hotNode.Update();

coldNode.NodeType = RcNodeData.NodeTypes.BOUNDARY;
coldNode.UserOverride = true;
coldNode.InitialTemp = 250;
coldNode.Update();

var caseset = td.CreateCaseSet(
    "myCase", groupName: "", sindaFilenames: "myCase");
caseset.SindaControl.timend = 60;
caseset.SteadyState = 0;
caseset.Transient = 1;
caseset.SaveCap = 1;
caseset.Update();
caseset.Run();

// TD will automatically create a contour plot. Let's hide it:
td.ResetGraphics();

#endregion

// Let's find the record for which the disk's mCp-weighted average temp
// is highest:
var savPath = Path.Combine(workingDir, "myCase.sav");
var data = DatasetFactory.Load(savPath);
var nodes = new ItemIdentifierCollection(
    DataTypes.NODE, "MYDISK", data);
var CapAvgTemp = new WeightedAverageDataArray(
    StandardDataSubtypes.T, StandardDataSubtypes.C, nodes, data);
int maxTindex
    = CapAvgTemp.GetValues().IndexOf(CapAvgTemp.GetValues().Max());

// XY plot the mCp-weighted average temp:
var plot = new SimplePlot();
plot.AddSeries(CapAvgTemp);
plot.Show();

// create a T contour plot at the max mCp-weighted average record:
var tdDataset = td.DatasetManager.CreateDataset(
    "myCase Dataset",
    savPath,
    OpenTD.PostProcessing.Dataset.DataSourceTypes.SF);
tdDataset.CurrentTimeIndex = maxTindex;
tdDataset.ShowContourPlot();
```

```
            td.ZoomExtents();
        }
    }
}
```

8. Launching Programs in the Thermal Desktop Process using Add-Ins

All of the example programs presented in this document run as a separate process, communicating with Thermal Desktop instance(s) using interprocess communication. There is another option: OpenTD Add-Ins allow you to create programs using familiar OpenTD syntax that can be run as commands in Thermal Desktop. You can even add code to add GUI buttons to launch these commands. A full description of how to create Add-Ins is beyond the scope of this document. To learn more, see https://www.crtech.com/forum/topic/create-your-own-td-add-version-63

9. Extras

9.1. Control how OpenTD Connects to Thermal Desktop

In this guide, we mostly call ThermalDesktop.Connect() with default options, which means it starts a new instance of TD with the latest version of AutoCAD available, then creates a new, blank drawing. To change this behavior, set the ThermalDesktop.*ConnectConfig* member before calling Connect(), as shown in the following program, which hides the AutoCAD window while it's running:

```
using OpenTD;
namespace OpenTDGettingStarted
{
    class ControlHowOpenTDConnects
    {
        public static void Main(string[] args)
        {
            var td = new ThermalDesktop();
            td.ConnectConfig.AcadVisible = false;
            td.Connect();
            // useful code goes here...
            td.Quit();
        }
    }
}
```

This program also demonstrates the ThermalDesktop.*Quit()* method, which tells AutoCAD to close, regardless of whether there are unsaved changes. (To save a dwg, use ThermalDesktop.*SaveAs*.) To *really* force AutoCAD to quit – by killing the acad.exe process – use ThermalDesktop.Quit(true). This should be considered a last resort. It may leave lock files in the dwg directory and/or have other unintended consequences.

Here are some useful members of ConnectConfig:

- **AcadExePathname**: By default, Connect() starts the latest version of AutoCAD found on your machine. Set AcadExePathname to the full path of any acad.exe to use it instead.
- **AcadVisible**: Set to false to hide the AutoCAD window. This only works when starting new TD instances, not attaching to already-running instances.
- **DwgPathname**: If this is set, Connect() will try to attach to or start a TD instance with this dwg. It will throw an exception if the dwg does not exist. Instead of editing ConnectConfig, this can also be set using an overloaded ThermalDesktop constructor:

```
var td = new ThermalDesktop(@"path\to\dwg");
```

- **PipeEndpointName**: Use this after issuing one of the TD commands for starting servers (see Section 9.2) to connect to an already-running instance of TD.
- **Type**: This determines whether Connect() will try to attach to an already-running instance of TD or start a new one. The default is AUTO, which means it will try both – first it will try to attach, but if that fails, it will start a new instance.

A complete list of ConnectConfig members (along with all OpenTD types) can be found in the "OpenTD Class Reference" document. (See Section 10.)

9.2. Control OpenTD Servers from AutoCAD

A client program communicates with TD via an OpenTD server launched within the acad.exe process, identified with a string called a *Pipe Endpoint Name*. When you load a dwg containing TD entities, an OpenTD server is started using the dwg pathname to generate the pipe endpoint name. That's how the Connect() method finds the correct acad process when DwgPathname is set (see Section 9.1). An OpenTD server is also automatically started when you start a new instance of TD via OpenTD.

There are commands you can use at the AutoCAD command line to query and start OpenTD servers:

- RCOPENTD: This command will ask you for a pipe endpoint name and start a server with that name.
- RCOPENTDRANDOM: This command will start a server with a random name.
- RCOPENTDDWG: This command will start a server with a name based on the dwg pathname. It is included for completeness, but there should never be a reason to call it, since it is called automatically by TD.
- RCOPENTDLISTSERVERS: This command will list the pipe endpoint names of the active OpenTD servers.

To connect a client program to a server based on endpoint name, set ThermalDesktop.ConnectConfig.PipeEndpointName before calling Connect() – see Section 9.1.

9.3. Execute AutoCAD Commands

When an OpenTD method is not available to perform a task, you might be able to use the ThermalDesktop.*SendCommand* method to send an AutoCAD command to TD. SendCommand has the following signature:

```
void SendCommand(string command, bool echo = true, int delay = 1000)
```

In addition to the string containing the command, there are two optional parameters: *echo* and *delay*. Echo determines whether your command will be echoed to the AutoCAD command line and defaults to true. Delay is the time in milliseconds that your program will wait after issuing the command before continuing. SendCommand executes asynchronously, and currently there is no way for OpenTD to check when it is done. The

workaround is to wait a fixed amount of time before continuing. The default delay is 1000 ms. You may wish to experiment with shorter delays to speed up execution.

The following program demonstrates using SendCommand:

```
using OpenTD;
namespace OpenTDGettingStarted
{
    class ExecuteAutocadCommands
    {
        public static void Main(string[] args)
        {
            var td = new ThermalDesktop();
            td.Connect();

            // draw an AutoCAD rectangle:
            td.SendCommand("rectang 3,3 4,5 ");

            // draw AutoCAD text: (Using a dash in front
            // of the command to make it command-line only.
            // This works for some AutoCAD commands.)
            td.SendCommand("-text 3,2.6 0.25 0 A Door\n");

            // zoom extents using OpenTD method:
            td.ZoomExtents();

            // zoom view to scale factor 1.5: (using
            // abbreviated command names)
            td.SendCommand("z s 1.5 ");
        }
    }
}
```

9.4. Execute TD COM Commands

Before OpenTD, there was a Windows Component Object Model (COM) interface to TD. It could be used to perform a subset of OpenTD functions. To use it, commands were entered as text strings and parsed by the TD COM server.

In OpenTD, the ThermalDesktop.*SendLegacyComCommand* method emulates the TD COM parser without actually using a COM connection.

Most TD COM functionality can be accomplished using native OpenTD methods. The SendLegacyComCommand method is included for completeness, and to ease the process of migrating an existing TD COM application to OpenTD. SendLegacyComCommand is not recommended to be used for new OpenTD applications.

Since the underlying RadCAD functions used by the COM server report success/failure via the AutoCAD console and not via a return value, SendLegacyComCommand may appear to succeed when it did not. It will only throw exceptions if it doesn't recognize a command or if the command crashes AutoCAD, and not if the command fails in some non-catastrophic way.

SendLegacyComCommand usually returns an empty string. The only exception to this is the "get" command, which attempts to return a string representing the value of a symbol.

The following commands are supported:

- **acadcommand "command" ["optional parameter" "optional parameter" ...]**
  Executes an AutoCAD console command. The command and any parameters are individually enclosed in double quotes. New applications should use ThermalDesktop.SendCommand instead. Examples:
  - SendLegacyComCommand("acadcommand \"zoom\" \"extents\"")
  - SendLegacyComCommand("acadcommand \"line\" \"1,1\" \"2,2\" \"\"")
- **case [optional parameters]**
  Calculates radiation and Cond/Cap data for the current case set. New applications should use OpenTD CaseSetManager and/or CaseSet instead. Examples:
  - SendLegacyComCommand("case") // Calculates all radiation and cond/cap data.
  - SendLegacyComCommand("case rad0 rad3 cc") // Calculates 1st and 4th radation tasks and cond/cap data.
- **caseset parameters**
  Provides access to the Case Set Manager. New applications should use OpenTD CaseSetManager and/or CaseSet instead.
- **copyradkfiles fromFilename toFilename**
  Copies radk files.
- **createmapset**
  Undocumented interactive command.
- **disableuserbreaks**
  Disables the ability to press ESC to end tasks.
- **displaycurrentdataset**
  Displays the current post-processing dataset. New applications should use OpenTD.PostProcessing instead.

- **dumpppmap outputFilename**
  Undocumented post-processing command. New applications should use OpenTD.Results instead.
- **exitautocad**
  Attempts to exit Thermal Desktop without saving the dwg. New applications should use ThermalDesktop.Quit instead.
- **exportnodeinfo**
  Calls the RcExportNodeInfo command. New applications should use ThermalDesktop.ExportNodeInfo instead.
- **get symbolName**
  Returns a string representing the evaluated value of a symbol. New applications should use ThermalDesktop.GetSymbolValue instead.
- **importcomet filename**
  Undocumented.
- **mapnastran inputFilename outputFilename [optional tolerance]**
  Maps the current post-processing data to the Nastran mesh defined in inputFilename, writing the results in Nastran format in outputFilename. New applications should use PostProcessing.*DataMapper* instead.
- **object**
  This command is used by Dynamic SINDA/FLUINT to display the value of OBJECT in the TD Dynamic SINDA/FLUINT status window.
- **opticalias alias opticalProp**
  Changes what optical property an alias refers to. The names cannot include spaces. New applications should use ThermalDesktop.OpticalPropAliasManager instead.
- **orbit parameters**
  Provides access to the Heating Rate Case Manager. New applications should use OpenTD Orbit instead.
- **output message**
  Writes message to the TD Dynamic SINDA/FLUINT status window.
- **postprocess filename [optional delay in ms]**
  Creates or updates a TEXT-type dataset from a file. New applications should use PostProcessing.DatasetManager instead.
- **ppnexttime**
  Steps forward to the next record in the current dataset. New applications should use PostProcessing.DatasetManager instead.
- **ppsavefile savOrCsrPathname**
  Creates or updates a SF-type dataset from a save file or CSR directory. New applications should use PostProcessing.DatasetManager instead.

- **ppsettime index**
  Sets the record for the current post-processing dataset. Uses 0-indexed record array index (0, 1, 2, etc) not record number. New applications should use PostProcessing.DatasetManager instead.
- **sendf4**
  Sends the F4 keycode. Useful for capturing the screen using hypersnap/cam.
- **set symbolName expressionIncludingSpaces**
  Updates an existing symbol's expression. New applications should use OpenTD Symbol instead.
- **setmapconstanttol toleranceInDwgUnits**
  Sets the constant tolerance to be used by the mapnastran command, in dwg units. New applications should use PostProcessing.DataMapper instead.
- **setmapcurrentorall parameter**
  If parameter is ALL, the mapnastran command will perform the mapping for all records in the dataset. If parameter is any other value, it will perform the mapping at the current record. New applications should use PostProcessing.DataMapper instead.
- **setmapset mapsetName**
  Instructs the mapnastran command to only use the specified object mapset.
- **setmapvariabletol TolerancesSeperatedBySpaces**
  Sets the progressive tolerance to be used by the mapnastran command, in dwg units.
- **startcase [optional name]**
  Runs the current case set, or the one specified by the optional name. New applications should use OpenTD CaseSetManager and/or CaseSet instead.
- **tdmapallmappers appendToFilename**
  Executes all mappers. Inserts the appendToFilename string at the end of each output file, before any extension. New applications should use PostProcessing.DataMapper instead.
- **thermoalias alias thermoProp**
  Changes what thermophysical property an alias refers to. The names cannot include spaces. New applications should use ThermalDesktop.ThermoPropAliasManager instead.
- **update**
  Updates all entity values based on symbol expressions.

9.5. The Magic of Implicit Casting

In Section 2.2, we created a conductor with the following code:

```
var c = td.CreateConductor(n1, n2);
```

where n1 and n2 were existing Node objects. CreateConductor has several overloads. The one we used is:

```
Conductor CreateConductor(Connection from, Connection to)
```

We took advantage of the fact that OpenTD will implicitly cast a *DbObject* (like a Node) to a new Connection. Explicitly, the C# compiler constructed Connections from our Nodes like this:

```
var c = td.CreateConductor(new Connection(n1, 1), new Connection(n2, 1));
```

OpenTD uses implicit casting in many places. For example, to cast StandardDataSubtypes to DataSubtypes. (See Section 7.1.5.) Some languages do not support .NET implicit casting. If you're using one of those languages to interface with OpenTD, you can always explicitly cast to the required type. (See Sections 12 and 13.)

9.6. A Note on OpenTD Versioning

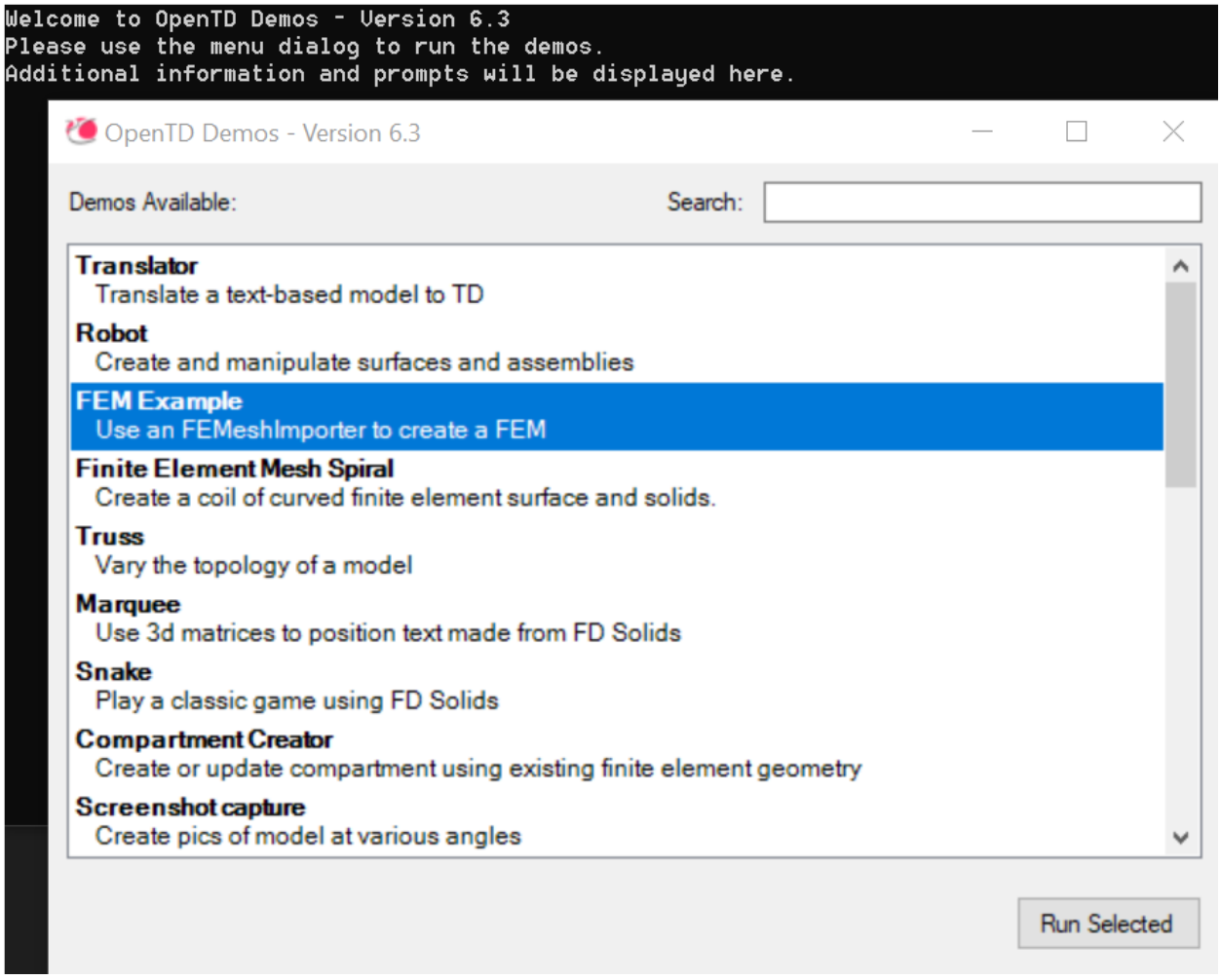
OpenTD is included with TD as the OpenTD.dll, OpenTD.CoSolver.dll, and OpenTD.Results.dll assemblies installed in the Global Assembly Cache (GAC). All OpenTD types are contained within the "OpenTD" namespace. Any changes we make to this interface will be additive. That is, we will not remove classes or methods. Therefore, you can write software referencing OpenTD knowing that it will not be broken by updates or new releases of TD.

Previously we changed the dll filenames and namespaces to reflect the current version, and released a new version of OpenTD with each version of TD. Starting with assembly version 92.0.0.0 released with TD 2025 R1, we will no longer do either of those things. That is, from this point forward versioning will be accomplished entirely using .NET version numbers.

Each time the dll's are built, we will update the File Versions. For interface-breaking changes, we will also update the Assembly Version of the affected dll(s).

10. Further Reading

- Also installed with TD is a file called “OpenTD Class Reference.chm”. This contains a complete list of all public types, members, and methods in OpenTD. You can find it in the TD installation directory, usually in the same location as this guide.
- The TD user forum (http://www.crtech.com/forum) contains more OpenTD demos, written in C#, MATLAB, and Python. Two boards are especially useful:
  - Software Usage->Tutorials->OpenTD contains a post called OpenTD Demos that contains a Visual Studio project that demonstrates many more features of OpenTD:

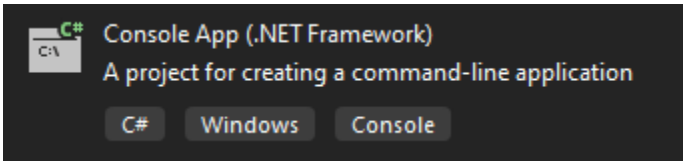


  - Software Usage->Product-specific Discussions->OpenTD is a place to ask and answer questions about OpenTD. There are dozens of discussions, covering many aspects of OpenTD usage.

11. Troubleshooting

Unhandled exception. System.IO.FileNotFoundException: Could not load file or assembly […]. The system cannot find the file specified.

When you try to run your program, it throws an exception like the above, referring to a dll such as SinapsXNet, CRlog4net, or something else. This likely means you tried to create a .NET Core or Standard project instead of a .NET Framework project, or your Thermal Desktop installation is corrupt. If the former, start a new .NET Framework project. In Visual Studio the project template should look something like this:



If that does not work, download and reinstall Thermal Desktop.

SINDA/FLUINT throws an error when I try to run a case using OpenTD

You have tried to run a case using CaseSet.Run but it does not work, even though you can normally run cases using the GUI. Are you running your client program from within Visual Studio using the debugger or the “Start Without Debugging” command? Try building your client and launching it by double-clicking on the exe file in Windows Explorer instead. Sometimes Visual Studio introduces environment variables into child processes that interfere with SINDA/FLUINT and cause it to fail.

Unhandled Exception: System.BadImageFormatException: Could not load file or assembly 'OpenTD.Results […]’ or one of its dependencies. An attempt was made to load a program with an incorrect format.

When you try to run a program that references OpenTD.Results.dll, you might get this exception. We had to compile OpenTD.Results.dll for x64 platforms, not “Any CPU” like many .NET assemblies. This means that if you reference it, your program also needs to

compile for x64 platforms. See Section 7.1 above for more information on how to set up your program correctly.

My problem is not listed here

Please contact us at crtech.support@ansys.com. Please include “OpenTD” and a descriptive title for your problem in the subject line, with a detailed description in the main body. We will be happy to help get OpenTD working for you.

12. Appendix A: Using OpenTD with MATLAB

While it is not feasible for us to maintain separate "Getting Started with OpenTD" guides for every programming language, we would still like to help you get started with OpenTD, even if you are not using C#. The following .m script is a MATLAB port of the program in Section 5.1 “Create and Run a Case”. This can be used as a sort of "Rosetta Stone" to help you translate other C# examples to MATLAB.

```
%% Using OpenTD with MATLAB
% CRTech
% Tested with MATLAB R2023b

% OpenTD is an Application Programming Interface (API) for Thermal Desktop
% (TD) that allows you to automate many of the tasks currently performed
% interactively using TD’s Graphical User Interface (GUI). OpenTD gives you
% the tools to programmatically create, query, edit, delete, and run
% models. You can use any .NET language to interact with OpenTD (C#,
% VB.NET, F#, etc.) or any system that can load .NET assemblies such as
% MATLAB or Python.

% Regardless of how you interact with OpenTD, you’ll need to have at least
% an intermediate understanding of .NET object-oriented programming. If you
% are starting from scratch, we recommend learning C#, since it is the
% language that we support. However, we understand that there might be
% compelling reasons for you to connect to OpenTD via MATLAB. It is
% possible, although the way MATLAB handles .NET enums is awkward and
% MATLAB does not support implicit constructors.

% To get started with OpenTD, read "Getting Started with OpenTD.pdf",
% which can be found in your TD v241 installation directory under "Manual".
% The Getting Started guide explains the fundamental concepts of OpenTD,
% using several C# examples. We've ported one of those examples to MATLAB
% below.

%% The "Create and Run a Case" example ported to MATLAB
% See "Getting Started with OpenTD.pdf" in your TD v241 installation
% directory under "Manual" for an explanation of this script.

% Note: Please contact us at crtech.support@ansys.com if you think there are
% better ways to use OpenTD with MATLAB, especially with regard to .NET
% enums and implicit constructors. For examples of awkward code, see how a
% node is set to be a boundary node and how the InitialTemp of a node is
% set -- in the script below vs. in the original C#.

openTD = NET.addAssembly('OpenTD');
import OpenTD.*;
```

```
td = ThermalDesktop;
td.Connect();

% *** Create a simple model of a heated bar ***
barNodes = NET.createArray('OpenTD.Node', 10);
for i = 1:10
    n = td.CreateNode();
    n.Submodel = SubmodelNameData('bar');
    n.Id = i;
    n.MassVol = 10;
    n.Origin = Point3d(0.01 * (i - 1), 1, 0);
    n.InitialTemp = Dimensional(n.InitialTemp, 300);
    n.Update();
    barNodes(i) = n;
end
for i = 1:9
    c = td.CreateConductor(...
        Connection(barNodes(i).Handle), Connection(barNodes(i+1).Handle));
    c.Submodel = SubmodelNameData('bar');
    c.Value = 0.1;
    c.Update();
end

roomAir = td.CreateNode();
roomAir.Submodel = SubmodelNameData('room');
roomAir.NodeType = OpenTD.('RcNodeData+NodeTypes').BOUNDARY;
roomAir.Origin = Point3d(0.055, 1.1, 0);
roomAir.InitialTemp = Dimensional(roomAir.InitialTemp, 300);
roomAir.Update();

barConnections = NET.createGeneric(...
    'System.Collections.Generic.List', {'OpenTD.Connection'},10);
for i = 1:10
    barConnections.Add(Connection(barNodes(i).Handle));
end
convection = td.CreateConductor(...
    Connection(roomAir.Handle), barConnections);
convection.Value = 1;
convection.Submodel = SubmodelNameData('room');
convection.Update();

qTorch = td.CreateSymbol('qTorch', '80');
heatLoadConnections = NET.createGeneric(...
    'System.Collections.Generic.List', {'OpenTD.Connection'},1);
heatLoadConnections.Add(Connection(barNodes(1).Handle));
torch = td.CreateHeatLoad(heatLoadConnections);
torch.ValueExp.Value = qTorch.Name;
torch.Submodel = SubmodelNameData('torch');
torch.Update();

td.ZoomExtents();
% *** End simple model creation ***

% Create a transient case and run it:
nominal = td.CreateCaseSet(...
    'transient with nominal torch', '', 'torchNom');
nominal.SteadyState = 0;
```

```
nominal.Transient = 1;
nominal.SindaControl.timend...
    = Dimensional(nominal.SindaControl.timend, 600);
nominal.Update();
nominal.Run();

% Create a cold case by overriding a symbol, and run it:
cold = td.CreateCaseSet(...
    'transient with cold torch', '', 'torchCold');
cold.SteadyState = 0;
cold.Transient = 1;
cold.SindaControl.timend...
    = Dimensional(nominal.SindaControl.timend, 1200);
cold.SymbolNames.Add(qTorch.Name);
cold.SymbolValues.Add('50');
cold.SymbolComments.Add('cold torch heat input');
cold.SaveAll = 1;
cold.Update();
cold.Run();

%% Working with Dimensionals

% All dimensional quanitities in the API are stored using a custom .NET
% generic type called a Dimensional. For example, a Dimensional<Temp>
% stores temperatures. Using C#, Dimensionals are implicitly cast to and
% from doubles as required, but this does not appear to work in MATLAB.
% Instead, we've overloaded the double function and created a Dimensional
% function to explicitly cast doubles to Dimensionals.

function x = double(Dimensional)
% Cast a .NET generic Dimensional type to a double
    x = Dimensional.op_Implicit(Dimensional);
end

function x = Dimensional(Dimensional, double)
% Cast a double to a .NET generic Dimensional type
    x = Dimensional.op_Implicit(double);
end

%% Acknowledgements

% Thank you to Dan Hensley and Daniel Reasa with ATA Engineering for
% performing some of the early work to determine how to use OpenTD with
% MATLAB.
```

13. Appendix B: Using OpenTD with Python

While it is not feasible for us to maintain separate "Getting Started with OpenTD" guides for every programming language, we would still like to help you get started with OpenTD, even if you are not using C#. The following .py script is a MATLAB port of the program in Section 5.1 “Create and Run a Case”. This can be used as a sort of "Rosetta Stone" to help you translate other C# examples to Python. It uses the pythonnet module, found at: http://pythonnet.github.io/

```
#### Using OpenTD with Python ####
# CRTech
# Feb, 2022
# Created with Python 2.7.15 and pythonnet 2.3.0

# OpenTD is an Application Programming Interface (API) for Thermal Desktop
# (TD) that allows you to automate many of the tasks currently performed
# interactively using TD's Graphical User Interface (GUI). OpenTD gives you
# the tools to programmatically create, query, edit, delete, and run
# models. You can use any .NET language to interact with OpenTD (C#,
# VB.NET, F#, etc.) or any system that can load .NET assemblies such as
# MATLAB or Python.

# Regardless of how you interact with OpenTD, you'll need to have at least
# an intermediate understanding of .NET object-oriented programming. If you
# are starting from scratch, we recommend learning C#, since it is the
# language that we support. However, we understand that there might be
# compelling reasons for you to connect to OpenTD via Python. It is
# possible using the pythonnet module:

# http://pythonnet.github.io/

# To get started with OpenTD, read "Getting Started with OpenTD.pdf",
# which can be found in your TD v241 installation directory under "Manual".
# The Getting Started guide explains the fundamental concepts of OpenTD,
# using several C# examples. We've ported one of those examples to Python
# below.


#### The "Create and Run a Case" example ported to Python ####
# See "Getting Started with OpenTD.pdf" in your TD v241 installation
# directory under "Manual" for an explanation of this script.

# Note: Please contact us at crtech.support@ansys.com if you think there are
# better ways to use OpenTD with Python, especially with regard to setting
# dimensional values.
```

```
# REQUIREMENT: You must install the pythonnet module to use this script.

import sys
import clr

# Need to add explicit GAC path to sys.path so clr.AddReference
# can find OpenTD.dll. Note the use of forward slashes in the path:
sys.path.append("C:/Windows/Microsoft.NET/assembly/GAC_MSIL/OpenTD/ReplaceMe")
clr.AddReference("OpenTD")
from OpenTD import *

# We'll want to use .NET System types and generic Lists:
from System import *
from System.Collections.Generic import List

# To access dimensional quantities in OpenTD, we need to use Dimensionals.
# These are cast to/from doubles implicitly in C#, but here we'll need to
# refer to them explicitly. (See setting InitialTemp, below.)
from OpenTD import Dimension
from OpenTD.Dimension import *

td = ThermalDesktop()
td.Connect()

# *** Create a simple model of a heated bar ***
barNodes = List[Node]()
for i in range(10):
    n = td.CreateNode()
    n.Submodel = SubmodelNameData("bar")
    n.Id = i + 1
    n.MassVol = 10.0
    n.Origin = Point3d(0.01 * i, 1.0, 0.0)
    n.InitialTemp = Dimensional[Dimension.Temp](300.0)
    n.Update()
    barNodes.Add(n)
for i in range(9):
    c = td.CreateConductor(Connection(barNodes[i]), Connection(barNodes[i+1]))
    c.Submodel = SubmodelNameData("bar")
    c.Value = 0.1
    c.Update()

roomAir = td.CreateNode()
roomAir.Submodel = SubmodelNameData('room')
roomAir.NodeType = RcNodeData.NodeTypes.BOUNDARY
roomAir.Origin = Point3d(0.055, 1.1, 0.0)
roomAir.InitialTemp = Dimensional[Dimension.Temp](300.0)
roomAir.Update()

barConnections = List[Connection]()
for n in barNodes:
    barConnections.Add(Connection(n))
convection = td.CreateConductor(Connection(roomAir), barConnections)
convection.Value = 1.0
convection.Submodel = SubmodelNameData("room")
convection.Update()

qTorch = td.CreateSymbol("qTorch", "80")
heatLoadConnections = List[Connection]()
```

```
heatLoadConnections.Add(Connection(barNodes[0]))
torch = td.CreateHeatLoad(heatLoadConnections)
torch.ValueExp.Value = qTorch.Name
torch.Submodel = SubmodelNameData("torch")
torch.Update()

td.ZoomExtents()
# *** End simple model creation ***

# Create a transient case and run it:
nominal = td.CreateCaseSet("transient with nominal torch", "", "torchNom")
nominal.SteadyState = 0
nominal.Transient = 1
nominal.SindaControl.timend = Dimensional[Dimension.Time](600.0)
nominal.Update()
nominal.Run()

# Create a cold case by overriding a symbol, and run it:
cold = td.CreateCaseSet("transient with cold torch", "", "torchCold")
cold.SteadyState = 0
cold.Transient = 1
cold.SindaControl.timend = Dimensional[Dimension.Time](1200.0)
cold.SaveQ = 1
cold.SymbolNames.Add(qTorch.Name)
cold.SymbolValues.Add("50")
cold.SymbolComments.Add("cold torch heat input");
cold.SaveAll = 1;
cold.Update()
cold.Run()

#### Acknowledgements
# Thank you to James Etchells with the European Space Agency (ESA) for
# performing some of the early work to determine how to use OpenTD with
# Python.
```

14. Appendix C: Using OpenTD with Powershell

Windows Powershell can be used to interact with OpenTD, which is useful since it is included with Windows, i.e., you don't need to install Visual Studio, MATLAB, or anything extra to use OpenTD. And unlike Python or MATLAB, Powershell was designed to support .NET objects, so its .NET syntax isn't too bad.

Powershell is likely already installed on your Windows machine. If not, or you'd like help finding it, check out the official documentation:

https://docs.microsoft.com/en-us/powershell/

Here's a simple Powershell script that loads OpenTD, creates a ThermalDesktop instance and opens it:

```
Add-Type -Path "ReplaceWithPathTo\OpenTD.dll"
$td = New-Object -TypeName 'OpenTD.ThermalDesktop'
$td.Connect()
```

15. Appendix D: Using OpenTD Interactively with the C# Interactive Compiler

Normally C# is compiled before running, but you can open an interactive C# Read-eval-print loop (REPL) in Visual Studio using the following command: View->Other Windows->C# Interactive

Once open, you can use it to interactively execute C# code, including OpenTD. For example:

```
> #r "OpenTD" // loads the dll as a reference
> using OpenTD;
> var td = new ThermalDesktop();
> td.Connect();
> var n = td.CreateNode(new Point3d(1, 1, 3));
> td.ZoomExtents();
> n.Comment = "Hello World!";
> n.Update();
> var nTest = td.GetNodes().First();
> Console.WriteLine(nTest);
RcNode.MAIN.1::236 "Hello World!"
> td.Quit();
```