Cutting Optimization Library CutGLib.

1. General Information.

CutGLib is a software library to perform the cutting optimization for rectangular (2D) and linear (1D) parts.

Having a list of rectangular parts CutGLib finds how to cut them from rectangular stocks with minimal material waste.

The library not only finds the parts placement but also generates the cutting instructions that can be easily translated for the CNC controllers.

The main features of CutGLib are:

- Cutting complexity levels from 2 (simple XY cuts) to 6.
- Horizontal, Vertical and Automatic cutting directions.
- Minimization stock panels rotation.
- Multiple sizes of the stocks.
- Minimization of the cutting layouts (woodworking feature).
- Non-zero saw thickness / kerf, part-to-part gaps.
- Generates list of remaining parts for guillotine cutting.
- Preserve part / grain orientation.
- Rotation of parts by 90 degrees for better optimization.
- Incomplete optimization for limited number of stocks.
- Unlimited parts and stocks quantities.
- Simple programming interface.
- Support various operating systems and platform.

The typical workflow is:

- Create the instance of the calculation class.
- Specify stock parameters (AddStock / AddLinearStock).
- Specify list of rectangular parts to cut (AddPart / AddLinearPart).
- Specify the saw thickness / kerf if applicable.
- Run the optimization process using 1D or 2D calculation methods.
- Get the coordinates of the parts or/and cutting instructions.

The optimization engine performs two major steps of the calculations:

- **Parts placement**. All specified parts get moved and turned in different ways to find such layout that occupies as less space as possible. Also the cutting instructions are generated during this step.
- **Quality control**. The quality control module performs several post-processing checks to assure the solution is valid and accurate. It checks if there are any intersections between the parts, cutting planes and the parts and checks if the list of cutting planes is complete for all parts.

If the engine encounters any problems during the calculation then it returns the error message.

The library is written on C# for .Net Framework. It can be used for the desktop computers (Windows 2000, XP, Vista, 7 and later), CNC machines (Windows CE 4.x and higher) and even for handheld devices such as Pocket PCs and Smartphones (Windows Mobile 2003 and higher).
The library can be easily integrated into .Net development tools (Visual Studio .Net) as well into traditional Win32 tools such as Ms Visual Basic, Borland Delphi and Ms Visual C++.

Numerical Precision

CutGLib supports values with up to 7 digits of the precision for all objects. It means all part sizes, stock sizes, saw kerf and other numeric values specified by the user of the library should fit into 7-digits range.
Let’s consider some examples. Please note: the red digits don’t fit into 7-digits range and will be removed (rounded up).

Example 1: If a stock length is **2500.0** (4 digits before the decimal separator) then the saw kerf can only be specified with up to 3-rd digits after the decimal separator (**0.001**). Saw kerf specified as **0.0125** will be rounded up to **0.013**.

Example 2: A stock length of **150.0** and part width of **2.1078** fits into 7-digits range. However, a part width of **1.031235** will not fit and will be rounded up to **1.0312**.

Decimal Point

Property **MaxDecimalPoint** defines the maximum count of digits you can have after the decimal separator. By default it is **3**, but the user can change it from **0** to **7**. It plays an important role to allow you specify numbers with higher precisions.

Example:
```
Calculator.AddStock(52, 42, 1);
Calculator.AddPart(6.0125, 25.5625, 2);
```

With default precision of **3** your parts will be rounded to **6.013** and **25.563**. If your project specifies **MaxDecimalPoint = 4** then the parts dimensions will be preserved as **6.0125** and **25.5625**.

2. Library Interface.

All indices are zero-based in the library. Indices are **0..N-1** for **N** objects.
The library provides its functionality via one interface class ***CutGLib.CutEngine*** with the following public members:

Input Properties:

`bool` **`CompleteMode;`**
Indicates that all of the parts have to be placed to accept the calculation was successful. If only some of the parts are required to be placed then set **CompleteMode** to **false**.
This property plays important role when the stock size is less than size of the parts to be cut from the stock.

`double` **`SawWidth;`**
Defines the saw thickness / kerf size / part to part gap. This value will be taken in account during the calculation.

`double` **`TrimLeft;`**
Defines the size of unused (trim) size on the left side of the stock.

`double` **`TrimTop;`**
Defines the size of unused (trim) size on the topside of the stock.

`double` **`TrimRight;`**
Defines the size of unused (trim) size on the right side of the stock.

`double` **`TrimBottom;`**
Defines the size of unused (trim) size on the bottom side of the stock.

`bool` **`UseLayoutMinimization;`**
If this property is TRUE then the calculation engine tries to minimize the number of different cutting layouts. This is a very important for wood cutting when the operator can load several stocks into the cutting machine and process them at once. If this property is FALSE (default) then the engine tries to minimize the number of stocks.

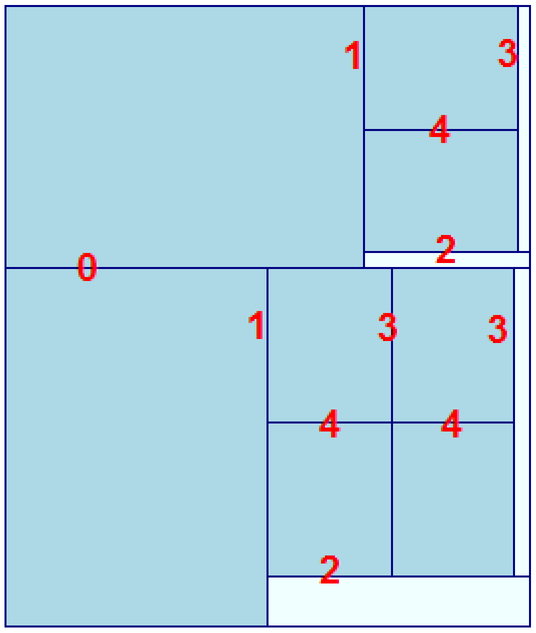
`int` **`MaxLayoutSize;`**
Defines the maximum number of stocks that can be cut at once from one layout. Works only if **`UseLayoutMinimization`** is **true**.

`double` **`WasteSizeMin;`**
Minimal acceptable size of the waste parts (0 - no restrictions). It plays an important role when cut glass stocks – because it’s impossible to cut tiny pieces of glass.

`String` **`Version`**`;`
Version of the library.

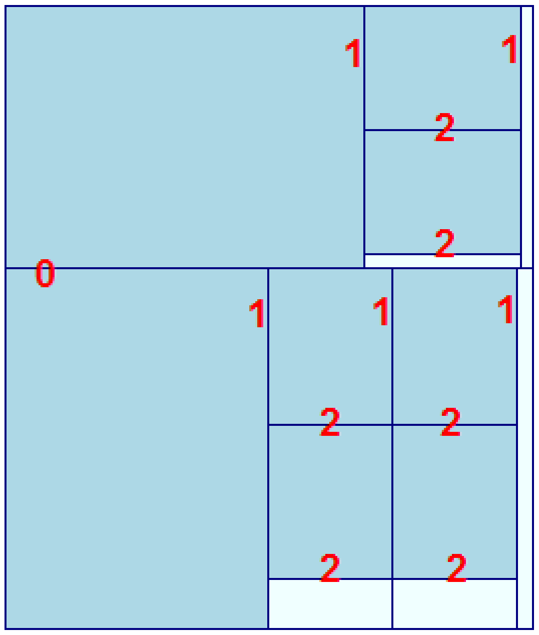
`bool` **`MinimizeSheetRotation;`**
It produces cutting layouts that require less stock panel rotations during the cutting operations. Therefore it minimizes the physical efforts and preparation time during loading/unloading stage of the cutting jobs. However, it may produce more waste parts and increase the total cutting lengths.

`MinimizeSheetRotation = false`



4 rotations, 4 waste parts

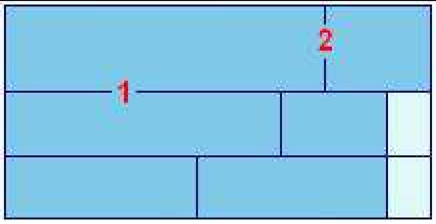
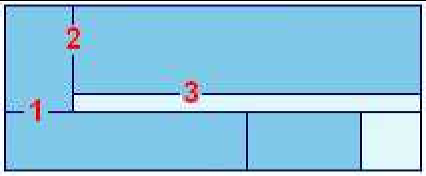
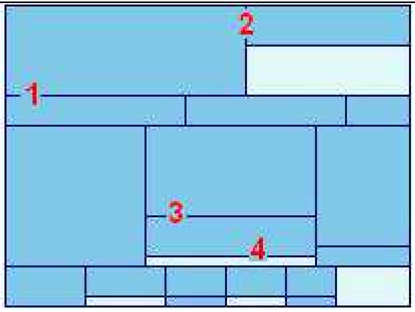
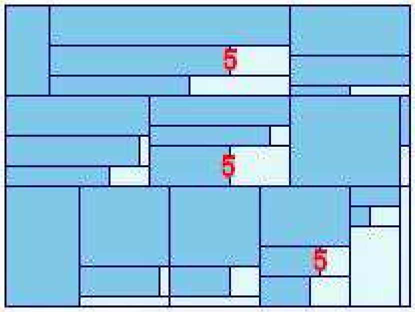
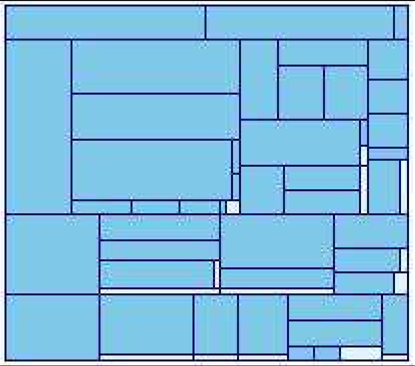
`MinimizeSheetRotation = true`



2 rotations, 5 waste parts

`int` **`MaxCutLevel;`**

Defines how complex the result layout will be. It goes from **2** to **6**. Level 2 allows only two cutting planes (so-called X/Y cuts) and it produces the result with the most material waste. However, this level is the simplest one to cut by hand.

Example	MaxCutLevel	Description
	**2**	Level 1 cuts split the stock panel by making cuts across the stock from one side to another. Level 2 cuts produce actual parts by slicing the stripes from the level 1 cuts.
	**3**	In addition to 1 and 2 levels the cuts on the level 3 produces the actual parts by making cuts of level 2 results.
	**4**	In addition to 1..3 levels the 4th level can cut the parts from the level 3 leftovers.
	**5**	Levels 5 cuts the leftovers from the level 4 cuts and therefore produces more complicated layout with the less waste.
	**6**	Level 6 produces the most complex and the most optimized layouts. It's a default level for CutGLib library.

Result Properties:

`int` **`StockCount;`**
Read-only property defines the total number of stocks specified in the cutting project.

`int` **`UsedStockCount;`**
Read-only property defines number of stocks that were used to cut all parts.

`int` **`LinearStockCount;`**
Read-only property defines the total number of linear stocks specified in the cutting project.

`int` **`UsedLinearStockCount;`**
Read-only property defines number of linear stocks that were used to cut all parts.

`int` **`RemainingPartCount;`**
Read-only property that defines number of remaining parts left after all cuts have been done. If the value is 0 then all parts have been used.

`int` **`PartCount;`**
Read-only property defines total number of parts specified in the cutting project.

`int` **`PlacedPartCount;`**
Specifies the number of parts have been processed and placed. It has the same value as **PartCount** in the most cases, but if **CompleteMode** is FALSE then **PlacedPartCount** can be less than **PartCount** indicating the fact that not of the parts have been placed.

`int` **`LayoutCount;`**
Read-only property defines number of different layouts / patterns of the cutting optimization.

`double` **`ElapsedTime;`**
Read-only property defines elapsed time in seconds spent for the calculation.

Setup Methods:

`void` **`Clear()`**
Clears all parts and settings from the calculator. This method is usually invoked to prepare another calculation.

Linear (1D) methods:

`bool` **`AddLinearStock`**`(double` **`ALength,`** `int` **`aCount,`** `string` **`aID`**`)`
Creates **aCount** new linear stocks with the specified length and text ID. After the optimization is done the list of used and un-used stocks gets created.

`bool` **`AddLinearStock`**`(double` **`ALength,`** `int` **`aCount,`** `string` **`aID,`** `bool` **`aWaste`**`)`
Similar to previous method, but has extra parameter **aWaste**, that defines if the stock is a waste / leftover from previous cutting jobs. If **aWaste** is ***true*** then such stock will be used first before any actual stocks that have **aWaste** as ***false***.

`bool` **`AddLinearStock`**`(double` **`ALength,`** `int` **`aCount)`**
Creates **aCount** new linear stocks with the specified length. After the optimization is done the list of used and un-used stocks gets created.

`bool` **`AddLinearStock(`**`double` **`ALength)`**
Same as **`AddLinearStock`**`(`**`ALength, 1).`**

`bool` **`AddLinearPart(`**`double` **`aLength ,`** `int` **`aCount)`**
Creates and stores **aCount** new linear (1D) segments defined by **aLength** parameters.
The method returns ***true*** if the segments have been successfully created and added. Otherwise returns ***false***. The segments created by this method are stored in internal part list.

`bool` **`AddLinearPart(`**`double` **`aLength ,`** `int` **`aCount,`** `string` **`aID)`**
In addition to the previous method it uses user-defined string **aID** (part label) that is assigned to the part.

`bool` **`AddLinearPart(`**`double` **`aLength)`**
Same as **`AddLinearPart(aLength , 1).`**

```
bool AddLinearPart(double aLength , int aCount,
                   double aAngleStart, double aAngleEnd)
```
Creates and stores **aCount** new linear (1D) segments defined by **aLength** parameters with start and end angles defined as described below.
The method returns ***true*** if the segments have been successfully created and added. Otherwise returns ***false***. The segments created by this method are stored in internal part list.

```
bool AddLinearPart(double aLength , int aCount,
                   double aAngleStart, double aAngleEnd, string aID)
```
In addition to the previous method it uses user-defined string **aID** (part label) that is assigned to the part.

Linear Properties

`LinearMaxSizePerStock.`
This property specifies maximum number different lengths that can be cut from one stock. For example: there are parts with three different lengths: 10 of 2m, 8 of 3m and 5 of 4m. If the property is not specified or set to be 0 then the engine calculate a cutting layout for one stock using all three sizes (2m, 3m and 4m). If the property set to 2 then the engine will use only two sizes: (2m, 3m) or (2m, 4m) or (2m, 4m) to generate cutting layout for one stock. Using this property may result in more material waste therefore there is another property:

`LinearMaxSizePerStockThreshold` to fine-tune the engine and allow more part lengths to be used. If a cutting layout material utilization is less than specified threshold then the engine will ignore the `LinearMaxSizePerStock` and use a different length.

Setting `LinearMaxSizePerStockThreshold` = 0 forces the engine always obey the `LinearMaxSizePerStock` constraint and produce more waste as a result. Setting `LinearMaxSizePerStockThreshold` = 1.0 will relax the engine to produce less waste and may violate `LinearMaxSizePerStock` constraint for some layouts.

`LinearSortAscending` defines sorting linear parts on stocks in ascending (true) (increasing length) or descending (false) order.

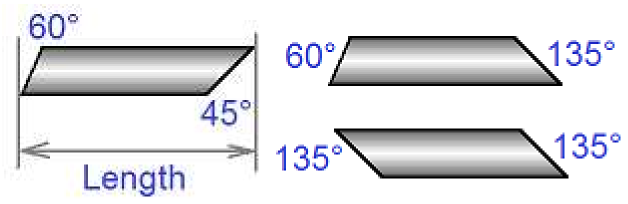
Combining Linear Stocks.

When some parts longer than any stocks, this option allows combining several stocks into one. Newly combined stock will be used as any other user-defined stocks with ID as a combination of each consistent stock.

`MaxSizePerStock` Allows to combine several linear stocks into a big one to accommodate parts that exceed any of specified stocks:
0: Do not allow combining stocks (default mode).
-1: Allow to combine unlimited number of stock length.
Any positive number specifies how many different stock lengths can be combine into one stock.
For example: having stocks of 10, 12 and 18 the user wants to combine only two different sizes (**`MaxSizePerStock = 2`**), the combination of stock lengths will be (10; 10 and 12; 10 and 18; 12; 12 and 18; 18).

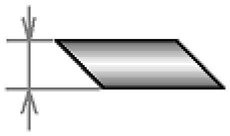
`AllowCombineStockRegularAndWaste.` If you have specified any waste stocks (remnants from previous cuts) you can combine them to actual stocks by selecting this option.
Setting **`AllowCombineStockRegularAndWaste = false`** will force strictly actual stocks or waste ones without mixing them.

Angle Cutting Information and Properties



CutGLib assumes all cuts start at the bottom side of stocks and go to the top side. Therefore angles are measured from the bottom point (cut's start) of stocks and go counterclockwise direction toward a top point (cut's end). Allowed range of angles is from 10° to 170°

`double` **`CrossSection;`**



The cross-section size is used to calculate the horizontal cutting difference according the formula:
HorizDif = CrossSection * Tangency(Angle - 90°)

If **HorizDif** is negative then the top point (cut's end) is located on the right side of the bottom point (cut's start). This happens for angles less than 90°. If **HorizDif** is positive then top point (cut's end) is located on the left side of the bottom point (cut's start). This happens for angles bigger than 90°.

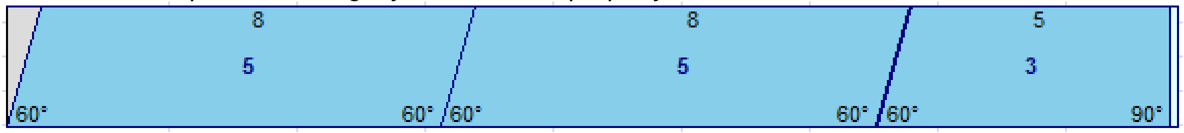
`bool` **`LinearExactAngle;`**

Different parts in your project could have different cutting angles. You can set this property to ***False*** to allow cutting parts with different adjacent angles.

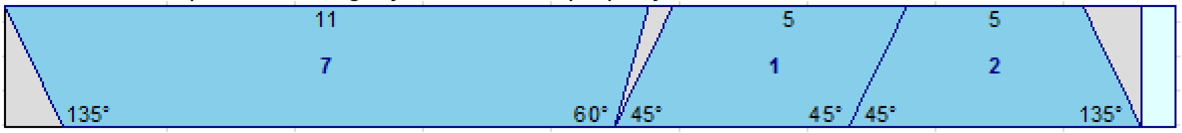
For example, the first part has end cut angle of 60° and the second one has beginning cut angle of 45° . If this property is ***False*** then CutGLib could place the second part after the first one.
If you set **`LinearExactAngle`** = **True** then second part must have beginning angle of 60°.

If you set **`LinearExactAngle`** = **False** then optimization engine has more flexibility and the results will be more optimized.

This is an example of a cutting layout when this property set to ***True***:

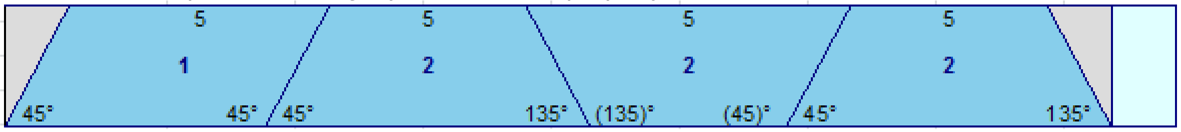


This is an example of a cutting layout when this property set to ***False***:



bool **LinearAllowRotate**;

When you cut non-symmetrical stocks, such as profile beams it's important to preserve the cutting angles as they specified. If you cut symmetrical stock, such as pipe, bars or tubes the cutting angles can be turned by 180° to get more optimized results.
For example, the first part has end angle of 60° and the second part has beginning angle of 120°. Cutting the second part after the first will produce some material waste. However, if we turned the second part upside-down (by 180°) then the start angle would become 60° and there would be no waste during the cutting.
Bottom line - if you cut symmetrical stocks set **LinearAllowRotate = True.**
This is an example of a cutting layout when this property is ***True***:

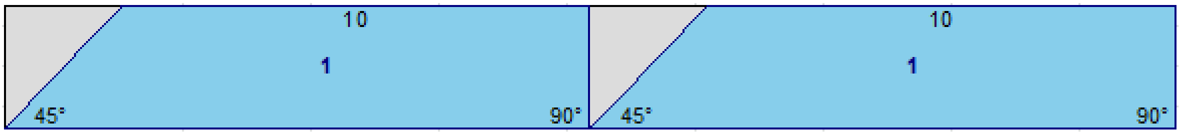


Note: **Angles in the bracket have been rotated. If the original angle was 65° then when a part gets rotated the angle became (135°).**

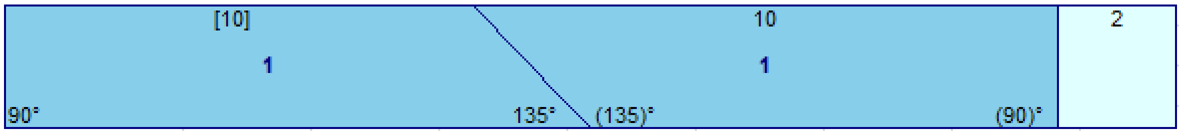
bool **LinearAllowFlipping**;

This option allows flipping parts by swapping/exchanging their start and end angles. Together with allow rotating options it can produce more optimized results.

For example, the parts have start angle of 45° end angle of 90°. Without flipping it produces the following layout:



If we flip the first part and rotate the second part upside-down (by 180°) then the following layout produced:



***Note*:** Square brackets indicate the part has been flipped and angles have been modified as well. The original angle was 45° then when a part gets rotated the angle became 135°

Rectangular (2D) methods:

`bool` **`AddPart(`**`double` **`aWidth,`** `double` **`aHeight,`** `int` **`aCount,`** `bool a`**`Rotatable`**`)`
Creates and stores **aCount** of new rectangular parts defined by **aWidth** and **aHeight** parameters. **aRotatable** defines if the parts can be rotated during the calculation: if **aRotatable** is ***false*** then the part cannot be rotated and it's orientation will be preserved.
The method returns ***true*** if the parts have been successfully created and added. Otherwise returns ***false***. The parts created by this method are stored in the internal list.

`bool` **`AddPart(`**`double` **`aWidth,`** `double` **`aHeight,`** `int` **`aCount,`** `bool a`**`Rotatable,`** `string` **`aID`**`)`
In addition to the previous method it uses user-defined string **aID** (part label) that is assigned to the part.

`bool` **`AddPart(`**`double` **`aWidth,`** `double` **`aHeight,`** `bool a`**`Rotatable)`**
Same as **`AddPart(aWidth, aHeight,`** `1, a`**`Rotatable).`**

`bool` **`AddPart(`**`double` **`aWidth,`** `double` **`aHeight,`** `int` **`aCount)`**
Same as **`AddPart(aWidth, aHeight, aCount, true).`**

`bool` **`AddPart(`**`double` **`aWidth,`** `double` **`aHeight)`**
Same as **`AddPart(aWidth, aHeight, 1, true).`**

`bool` **`AddStock`**`(double` **`aWidth`**`, double` **`aHeight,`** `int` **`aCount,`** `string` **`aID`**`)`
Creates and store **aCount** of the rectangular stocks specified by **`aWidth`**, **`aHeight`** `and text` **`aID.`**
If several different stock sizes are defined then the calculation engine tries to find out the combination of stocks that produces the least used area.

`bool` **`AddStock`**`(double` **`aWidth`**`, double` **`aHeight,`** `int` **`aCount,`** `string` **`aID,`** `bool` **`aWaste`**`)`
Similar to previous method, but has extra parameter **aWaste**, that defines if the stock is a waste / leftover from previous cutting jobs. If **aWaste** is ***true*** then such stock will be used first before any actual stocks that have **aWaste** as ***false***.

`bool` **`AddStock`**`(double` **`aWidth`**`, double` **`aHeight,`** `int` **`aCount)`**
Creates and store **aCount** of the rectangular stocks specified by **`aWidth`** `and` **`aHeight.`**
If several different stock sizes are defined then the calculation engine tries to find out the combination of stocks that produces the least used area.

`bool` **`AddStock`**`(double` **`aWidth`**`, double` **`aHeight)`**
Same as **`AddStock`**`(`**`aWidth`**`,` **`aHeight, 1`**`).`

`void` **`SetVerticalCutDirection(`**`bool` **`aRollMode)`**
Suitable for roll (stripe) cutting. If **aRoolMode** is **True** then it ensures the first cut is made completely from the stock's top to the bottom.

`void` **`SetVerticalCutDirection()`**
Defines the stock will be first cut vertically. Same as **`SetVerticalCutDirection(false).`**

`void` **`SetHorizontalCutDirection(`**`bool` **`aRollMode)`**
Suitable for roll (stripe) cutting. If **aRoolMode** is **True** then it ensures the first cut is made completely from the stock's left side to the right one.

`void` **`SetHorizontalCutDirection()`**
Defines the stock will be first cut horizontally. Same as **`SetHorizontalCutDirection(false).`**

`void` **`SetAutoCutDirection()`**
Defines the calculation engine will automatically detect what direction produce better results and use this direction to cut the stock first time.

Calculation Methods:

Linear (1D) method:

`string` **`ExecuteLinear();`**
`string` **`ExecuteLinear(`**`OnCalcState` **`aCalcStateEvent);`**
Performs 1D-cutting calculation of the linear parts in the source list on the stocks that sizes are defined by **`AddLinearStock()`**. It calculates the parts layout and defines number of stocks (**`UsedLinearStockCount`**) required to cut all parts. Returns empty string if calculation was done successfully. Otherwise returns an error message.

Rectangular (2D) method:

`string` **`Execute();`**
`string` **`Execute(`**`OnCalcState` **`aCalcStateEvent);`**
Performs guillotine cutting calculation of the parts in the source list on the specified (**AddStock**) stocks. Only orthogonal cuts from one side completely to another side are allowed.
It calculates the parts layout and defines number of stocks (**UsedStockCount**) required to cut all parts.
Returns empty string if calculation was done successfully. Otherwise returns an error message.

`void` **`OnCalcState(`**`ref bool` **`aStop)`**
Optional callback function `OnCalcState` provides a way to cancel the execution at any time by setting referenced parameter **aStop** = true.

Result Methods:

`bool` **`GetLayoutInfo(`**`int` **`aIndex,`** `out int` **`aFirstStock,`** `out int` **`aStockCount);`**
Gets information about the layout aIndex, such as index of the first stock (**aFirstStock**) in the layout and count of stocks (**aStockCount**) that share the same layout. **aIndex** is from **0** to **LayoutCount - 1**. Function returns ***True*** if the layout information successfully retrieved.

`double` **`GetLayoutFillRatio(`**`int` **`aLayout);`**
Gets layout's fill ratio in value between 0 and 1.

`int` **`GetPartCountOnStock(`**`int` **`aStockNo);`**
Returns number of parts cut (or placed) from the specified stock **aStockNo**.

`int` **`GetPartIndexOnStock(`**`int` **`aStockNo,`** `int` **`aPartNo);`**
Returns the global part index cut (or placed) from the specified stock **aStockNo**. aPartNo is the part ordinal number and goes from **0** to **GetPartCountOnStock**(aStockNo) – 1.

`int` **`GetRemainingPartCountOnStock(`**`int` **`aStockNo);`**
Returns number of remaining / waste parts left from the specified stock **aStockNo**.

`int` **`GetRemainingPartIndexOnStock(`**`int` **`aStockNo,`** `int` **`aPartNo);`**
Returns the global remaining part index left over from the specified stock **aStockNo**. aPartNo is the part ordinal number and goes from **0** to **GetRemainingPartCountOnStock**(aStockNo) – 1.

Linear (1D) methods:

```
int GetLinearCutsCount();
```

Gets the count of global linear cuts.

```
bool GetLinearCut(int aIndex, out int aStock, out double aLocation)
```

Gets the linear cut with the specified global index (from 0 to **GetLinearCutsCount**-1). Each cut is defined by **aLocation** point and indicates the position where to cut the linear **StockNo**. Returns ***false*** if the cut index is out of range.

```
bool GetLinearCut(int aIndex, out int aStock,
                  out double aLocation,
                  out double aAngle)
```

Gets the linear cut with the specified global index (from 0 to **GetLinearCutsCount**-1). Each cut is defined by **aLocation** point and indicates the position where to cut the linear **StockNo**. Returns ***false*** if the cut index is out of range. **aAngle** defines the angle with which the cut needs to be done.

```
int GetStockCutCount(int aStock);
```

Gets the count of cuts for the specified linear stock.

```
bool GetLinearStockCut(int aStock, int aCut,
                       out double aLocation,
                       out double aAngle)
```

Gets the linear cut with the specified index for the specified stock. Each cut is defined by **aLocation** start point and cutting angle **aAngle**. Returns ***false*** if the cut index is out of range (0.. GetStockCutCount-1).

```
bool GetResultLinearPart(int aPart,
                         out int Stock,
                         out double aLength,
                         out double aLocation);
```

Gets the length and calculated position of the linear part (1D) with the specified part index **aPart**.
aStock indicates the stock the part was cut from.
aPart is the same index from the source list and it goes from **0** to **PartCount** – 1.
The method returns ***false*** if the part **aPart** was not cut from the stock, for example, in case of incomplete optimization (**CompleteMode** = **false**).
aLocation defines the position of the part on the stock.

```
bool GetResultLinearPart(int aPart,
                         out int Stock,
                         out double aLength,
                         out double aLocation,
                         out string aID);
```

In addition to the previous method this one returns user-defined string **aID** (part label) that has been assigned to the part by **`AddLinearPart()`** method.

```
bool GetResultLinearPart(int aPart,
                         out int Stock,
                         out double aLength,
                         out double aAngleStart,
                         out double aAngleEnd,
                         out double aLocation,
                         out bool aRotated);
```

Gets the length, calculated position, start and end cutting angles and rotated status of the linear part (1D) with the specified part index **aPart**.
aStock indicates the stock the part was cut from.
aPart is the same index from the source list and it goes from **0** to **PartCount** – 1.
The method returns ***false*** if the part **aPart** was not cut from the stock, for example, in case of incomplete optimization (**CompleteMode** = **false**).
aLocation defines the position of the part on the stock.
`aAngleEnd` `and` **`aAngleEnd`** define part's start and end angles.
`aRotated` defines if the part has been rotated by 180°along its axis to get better match of its start and end angles.

```
bool GetResultLinearPart(int aPart,
                         out int Stock,
                         out double aLength,
                         out double aAngleStart,
                         out double aAngleEnd,
                         out double aLocation,
                         out bool aRotated,
                         out string aID);
```

In addition to the previous method this one returns user-defined string **aID** (part label) that has been assigned to the part by **`AddLinearPart()`** method.

```
bool GetRemainingLinearPart(int aPart,
      out int aStock,
      out double aLength,
      out double aLocation);
```

Gets the length and calculated position of the remaining part (cut-off) left over linear cuts from the stock.
aStock indicates the stock the part was left from.
aPart is from 0 to **`RemainingPartCount – 1`**.
The method returns ***false*** if the index is more or equal to the number of remaining parts.

```
bool GetRemainingLinearPart(int aPart,
      out int aStock,
      out double aLength,
      out double aLocation,
      out double aAngle);
```

Gets the length and calculated position of the remaining part (cut-off) left over linear cuts from the stock.
aStock indicates the stock the part was left from.
aPart is from 0 to **`RemainingPartCount – 1`**.
aAngle defines the angle of the waste part cut in case of angled cutting. If it's 0 then angle cut has not been used.
The method returns ***false*** if the index is more or equal to the number of remaining parts.

```
bool GetLinearStockInfo(int aStock,
     out double aLength, out bool aActive, out string aID)
```

Returns the length and text ID of the stock with specified Index **aStock**. The **aStock** is the same as **aStock** parameter in previous methods (**`GetResultLinearPart()`** `and` **`GetRemainingLinearPart()`**).
aActive indicates the stock was used during the calculation. If **aActive** is **false** then this stock was not used.
aStock is a global index of all stocks (used and not used) specified in the system and goes from **0** to **`LinearStockCount`** `– 1.`

`int` `GetLinearCombineStockCount(`**`int`** `aStock)`

Gets the count of actual linear stocks used to get the specified combined linear stock. If the stock was not combined then returns 0.

`int GetLinearCombineStockInfo(int` **`aStock`**`, int` **`aIndex`**`,`
`out double` **`aLength`**`, out bool` **`aActive`**`, out string` **`aID)`**

Returns information about actual stock **aIndex** that combined into the stock **`aStock.`** All returning parameters are the same as in **`GetLinearStockInfo()`**method.

Example of getting combined stocks information:

```
Calculator.GetLinearStockInfo(iStock, out width, out stockActive);
if (stockActive)
{
  int simpleCount = Calculator.GetLinearCombineStockCount(iStock);
  for (int iSimple = 0; iSimple < simpleCount; iSimple++)
  {
    Calculator.GetLinearCombineStockInfo(iStock, iSimple, out width, out stockActive, out
stckID);
  }
}
```

Rectangular (2D) methods:

```
bool GetResultPart(int aPart,
      out int aStock,
      out double aWidth, out double aHeight,
      out double aX, out double aY,
      out bool aRotated);
```

Gets the size and calculated position of the specified part.
If the part was rotated during the calculation then **Rotated** is ***true***. **aWidth** and **aHeight** don't get changed during part rotation and their values are preserved.
aStock indicates the stock the part was cut from.
aPart is the same index from the source list and it goes from **0** to **PartCount** – 1.
The method returns ***false*** if the part **aPart** was not cut from the stock, for example, in case of incomplete optimization (**CompleteMode** = **false**).

```
bool GetResultPart(int aPart,
      out int aStock,
      out double aWidth, out double aHeight,
      out double aX, out double aY,
      out bool aRotated,
      out string aID);
```

In addition to the previous method this one returns user-defined string **aID** (part label) that has been assigned to the part by **`AddPart()`** method.

```
bool GetRemainingPart(int aPart,
      out int aStock,
      out double aWidth, out double aHeight,
      out double aX, out double aY);
```

Gets the size and calculated position of the remaining part left over guillotine cuts from the stock.
aPart is from 0 to **`RemainingPartCount – 1`**.
The method returns ***false*** if the index is more or equal to the number of remaining parts.

```
bool GetStockInfo(int aStock,
      out double aWidth, out double aHeight,
      out bool aActive, out string aID)
```

Returns the sizes and text ID of the used stock with specified Index. **aStock** is a global index of all stock (used and not used) specified in the system goes from **0** to **StockCount** -1.
aActive indicates the stock was used during the calculation. If **aActive** is **false** then this stock was not used.

```
int GetPartCutsCount(int aPart);
```

Returns the number of nesting cuts to be done for the specified part. **aPart** indicates the index of the part. Returns -1 if the index is out of bonds (0..**PartCount**-1)

```
bool GetPartCut(int aPart, int aCut,
      out int aStock,
      out double aStart_X, out double aStart_Y,
      out double aEnd_X, out double aEnd_Y);
```

Gets the nesting cut **aCut** for the specified part (**aPart**). Each cut is defined by Start point and End point.
Each point is defined by X and Y coordinates.
Returns ***false*** if the part index more or equal to the number of parts or a**Cut** is more or equal to the number of cuts for the part.

Getting cuts

```
int GetCutsCount();
```
Returns global number of guillotine cuts.

```
bool GetCut(int aCut, out int aStock,
      out double aStart_X, out double aStart_Y,
      out double aEnd_X, out double aEnd_Y);
```
Gets the guillotine cut with the specified index. **aStock** indicates the stock index the cut is made for. Each cut is defined by Start point and End point. Each point is defined by X and Y coordinates. Guillotine cuts must be done in the order defined by the calculation, e.a. 0,1,2, etc. One should not make cut 3 before cut 2.
Returns ***false*** if the cut index is out of range (0.. GetGuillotineCutsCount -1).

```
bool GetCut(int aCut, out int aStock,
      out double aStart_X, out double aStart_Y,
      out double aEnd_X, out double aEnd_Y,
      out int aLevel);
```
Additional output parameter **aLevel** defines the level of the cut (0 - rip cut, 1 - cross cut, etc.).

```
int GetStockCutCount(int aStock);
```
Gets the count of guillotine cuts for the specified stock.

```
bool GetStockCut(int aStock, int aCut,
      out double aStart_X, out double aStart_Y,
      out double aEnd_X, out double aEnd_Y);
```
Gets the guillotine cut with the specified index for the specified stock. Each cut is defined by Start point and End point. Each point is defined by X and Y coordinates. Guillotine cuts must be done in the order defined by the calculation, e.a. 0,1,2, etc. One should not make cut 3 before cut 2.
Returns ***false*** if the cut index is out of range (0.. GetStockCutCount-1).

```
bool GetStockCut(int aStock, int aCut,
      out double aStart_X, out double aStart_Y,
      out double aEnd_X, out double aEnd_Y,
      out int aLevel);
```
Additional output parameter **aLevel** defines the level of the cut (0 - rip cut, 1 - cross cut, etc.).

```
int GetStockTrimCutCount(int aStock);
```
Gets the count of guillotine trim cuts for the specified stock.

```
bool GetStockTrimCut(int aStock, int aCut,
      out double aStart_X, out double aStart_Y,
      out double aEnd_X, out double aEnd_Y);
```
Gets the trim cut with the specified index for the specified stock. Each trim cut is defined by Start point and End point. Each point is defined by X and Y coordinates. Trim cuts are done prior to the actual cutting.
Returns ***false*** if the cut index is out of range (0.. GetStockTrimCutCount-1).

```
bool GetStockTrimCut(int aStock, int aCut,
      out double aStart_X, out double aStart_Y,
      out double aEnd_X, out double aEnd_Y,
      out int aLevel);
```
Additional output parameter **aLevel** defines the level of the cut (0 – main trim cut, 1 – second trim cut orthogonal to the main one).

Export results to file

CutGLib provides several methods to export cutting optimization results as external files in different formats.

Image File

```
bool CreateStockImage(int aStock,
                      string aImageFileName,
                      int aMaxSize);
```

Generates image file (**PNG format**) with the specified name for the specified 2D (panel) stock. **aMaxSize** defines the maximum image file width or height. If a stock has width = 500 and height = 1000 then the PNG image will be 500x1000 pixels. It returns ***true*** if the file has been created successfully.

```
bool CreateStockImage(int aStock, string aImageFileName);
```

Same as **`CreateStockImage(aStock, aImageFileName, 1000);`**

```
bool CreateStockImage_Stream(int aStock,
                      Stream aImageStream,
                      int aMaxSize);
```

Writes image file (**PNG format**) to the specified stream. This method is similar to **`CreateStockImage()`** and only available for .Net client applications.

```
bool CreateLinearStockImage(int aStock,
                      string aImageFileName,
                      int aMaxSize);
```

Generates image file (**PNG format**) with the specified name for the specified 1D (linear) stock. **aMaxSize** defines the maximum image file width. If a stock has length = 500 then the PNG image will have width of 500 pixels. It returns ***true*** if the file has been created successfully.

```
bool CreateLinearStockImage(int aStock, string aImageFileName);
```

Same as **`CreateLinearStockImage(aStock, aImageFileName, 1000);`**

```
bool CreateLinearStockImage_Stream(int aStock,
                      Stream aImageStream,
                      int aMaxSize);
```

Writes image file (**PNG format**) to the specified stream. This method is similar to **`CreateLinearStockImage()`** and only available for .Net client applications.

AutoCAD DXF

```
bool ExportToDXF(string aFileName, bool aIncludePartIDs);
```

Generates AutoCAD DXF file (version R12) with the specified name for all used stocks.
For each exported stock the export creates a new layer with name Panel_1, Panel_2, etc.
Boolean parameter **`aIncludePartIDs`** defines if the part labels (IDs) will be exported to DXF file as well.

```
bool ExportCutsToDXF(string aFileName, bool aIncludeCutNumber, bool
aIncludeStockPanel);
```

Generates AutoCAD DXF file (version R12) with the specified name for all used stocks.
For each exported stock the export creates a new layer with name Panel_1, Panel_2, etc. It fills the list of cutting planes (cuts). Boolean parameter **`aIncludeCutNumber`** defines if the cut labels (1,2,3,etc.) will be exported to DXF file as well. Boolean parameter **`aIncludeStockPanel`** indicates the stock panel rectangle lines will be included into DXF file. If this parameter is False then only cutting planes will be exported.

Excel CSV

```
bool ExportToCSV(string aFileName);
```
Generates text comma-separated file (CSV) with the specified name for all used parts combined by the stocks.

Pattern Exchange Format (PTX)

The Pattern Exchange (PTX) format is a standard format for describing parts, boards, patterns and cutting information and can be used directly for some CNC saw machines.

```
bool ExportToPTX(string aFileName,
      string aJobName,
      string aCustomer,
      string aMaterial,
      double aMaterialThickness,
      int aUnits,
      double aUnitScale,
      bool aRequireAllTrimCuts);
```

The file contains one job and one material.
aJobName – Job number/name - reference for job.
aCustomer – Customer code or name.
aMaterial – Material code.
aMaterialThickness – Material thickness in appropriate measurement mode.
aUnits - Measurement mode = 0 (metric), 1 (decimal inches).
aUnitScale - A factor all coordinates and sizes will be multiplied by to get values in mm or inches.
If a project is done in centimeters then the **aUnitScale** = **10**.
If it's done in millimeters then the **aUnitScale** = **1**.
aRequireAllTrimCuts – when it is TRUE the generated PTX file will include trim cuts for all waste parts. FALSE means the waste parts will NOT have any trim cuts produced and the size of the waste parts will be increased by the trim values.
Default value is FALSE when the parameter omit in **`ExportToPTX()`** calls.

Licensing.

In order to use CutGLib after 30-days trial period the library requires activation of the license.
There are two license types:

1. **Single** – issued for the particular computer and links to its hardware configuration. If the library moved to another computer then the license will not work and another activation is required.
2. **Site** – this license is issued to the company (or individual) with a unique license key that allows to use the library on unlimited number of computer. The **Site** license provides developers with ability to distribute CutGLib with their applications to the end-users.

The **Single** license activation requires the hardware code from the computer where CutGLib will be used. This code is accessible from the method `string ComputerHardwareCode()`:

```
// First we create a new instance of the cut engine
CutEngine Calculator = new CutEngine();
// Activation of the single license
string HardwareCode = Calculator.ComputerHardwareCode();
```

This code (`HardwareCode`) needs to be used at Optimalon Software website to generate the license key at the URL: https://www.optimalon.com/License/LicData

In order to activate **Single** license the client needs to call the method in their code after creation of the cut engine instance:

```
// First we create a new instance of the cut engine
CutEngine Calculator = new CutEngine();
// Activation of the single license
Calculator.SetComputerLicenseKey(<License_Key>);
```

If client has more than one single license they all can be registered within the calculation engine using the following method:
`Calculator.RegisterComputerLicenseKey(<HardwareCode>, <License_Key>);`
This method needs to be called for all licenses the user has.

In order to activate registered licenses the following method should be called:
`Calculator.LoadRegisteredComputerLicenseKey();`

The **Site** license does not require the hardware code and can be used for many computers.
In order to activate **Site** license the client needs to call the method in their code after creation of the cut engine instance:

```
// First we create a new instance of the cut engine
CutEngine Calculator = new CutEngine();
// Activation of the site license
Calculator.SetSiteLicenseKey(<License_Key>);
```

Where **`License_Key`** is a text site license key received from Optimalon Software.

The **Server** license does not require the hardware code and can be used on web servers with shared environments like Azure or other cloud solutions.
In order to activate **Server** license the client needs to call the method in their code after creation of the cut engine instance:

```
// First we create a new instance of the cut engine
CutEngine Calculator = new CutEngine();
// Activation of the site license
Calculator.SetServerLicenseKey(<License_Key>);
```

Where **`License_Key`** is a text site license key received from Optimalon Software.

3. Example of the Library Usage. (C# syntax)

3.1. Cut one linear stock.

This example demonstrates how to cut a linear stock (log/beam/wire) with size of **10.0** feet.
Let say we need to cut **9** parts of **3.0** feet, **3** parts of **5.0** feet and **2** parts of **7.0** feet.

```
// First we create a new instance of the cut engine
CutEngine Calculator = new CutEngine();

// Add 7 linear stocks of 10.0 feet
Calculator.AddLinearStock(10.0, 7);

// Add linear pieces we have to cut from the stock:
Calculator.AddLinearPart(3.0, 9); // 9 pieces of 3.0 feet
Calculator.AddLinearPart(5.0, 3); // 3 pieces of 5.0 feet
Calculator.AddLinearPart(7.0, 2); // 2 pieces of 7.0 feet

// Run the calculation:
string result = Calculator.ExecuteLinear();

// If result is not empty then it has an error message
if (result == "")
{
  // Calculator.UsedLinearStockCount specifies the number of linear stocks required.
  Console.Write("Need {0} linear stocks", Calculator.UsedLinearStockCount);

  int StockNo;
  double Len = 0, X = 0;
  // Get the results. Here we just iterate by parts and and get
  // indices of stocks where a part has to be cut from
  for (int iPart = 0; iPart < Calculator.PartCount; iPart++)
  {
    if (Calculator.GetResultLinearPart(iPart, out StockNo, out Len, out X))
    {
      // StockNo specifies the stock part iPart gets cut from
      // Len is the length of the part iPart
      // X is the coordinate of the part iPart on the stock StockNo.
      Console.Write("Part {0}:  stock={1}  X={2};  Length={3}",
                    iPart, StockNo, X, Len);
    }
    else Console.Write("Source piece {0} was not placed\n", iPart);
  }
}
else
{
  // Output the error message
  Console.Write("%S", result);
}
```

3.2. Cut multiple size linear stocks.

This example demonstrates how to cut a linear stock (log/beam/wire) with different sizes.
Let say we need to cut **6** pieces of **11.0** feet, **8** pieces of **9.0** feet, **12** pieces of **7.0** feet and **4** pieces of **16.0** feet.
There are **10** stocks of **20.0** feet, **5** stocks of **31.0** feet and **5** of **34.0** feet.

```
// First we create a new instance of the cut engine
CutEngine Calculator = new CutEngine();
// Add 10 linear stocks of 20.0 feet
Calculator.AddLinearStock(20.0, 6);
// Add 5 linear stocks of 31.0 feet
Calculator.AddLinearStock(31.0, 5);
// Add 5 linear stocks of 34.0 feet
Calculator.AddLinearStock(34.0, 5);
// Add linear pieces we have to cut from the stock:
Calculator.AddLinearPart(11.0,  6); // 6 pieces of 11.0 feet
Calculator.AddLinearPart( 9.0,  8); // 8 pieces of 9.0 feet
Calculator.AddLinearPart( 7.0, 12); // 12 pieces of 7.0 feet
Calculator.AddLinearPart(16.0,  4); // 4 pieces of 16.0 feet
// Run the calculation
string result = Calculator. ExecuteLinear();
if (result != "")
{
  // Output the error message and exit
  Console.Write("%S", result);
  return;
}
```

Now we use another approach to output results. The calculation created several different cutting layouts, so let's iterate by layouts and output the stock length used for each layout and parts cut.

```
int StockIndex,StockCount,iPart,iLayout,partCount,partIndex,tmp,iStock;
double partLength,X,StockLength;
bool StockActive;
for (iLayout = 0; iLayout < Calculator.LayoutCount; iLayout++)
{
  // StockIndex is global index of the first stock used in the layout iLayout
  // StockCount is quantity of stocks of the same length as StockIndex used
  Calculator.GetLayoutInfo(iLayout, out StockIndex, out StockCount);
  // Iterate by each stock in the layout, starting from StockIndex
  for (iStock = StockIndex; iStock < StockIndex + StockCount; iStock++)
  {
    // Output the stock index and length
    Calculator.GetLinearStockInfo(iStock, out StockLength, out StockActive);
    Console.Write("Stock={0}:  Length={1}", iStock, StockLength);
    // Output the information about parts cut from this stock
    // Get quantity of parts cut from the stock:
    partCount = Calculator.GetPartCountOnStock(iStock);
    // Iterate by parts and get indices of cut parts
    for (iPart = 0; iPart < partCount; iPart++)
    {
      // Get global part index of iPart cut from the current stock
      partIndex = Calculator.GetPartIndexOnStock(iStock, iPart);
      // Get length and location of the part
      // X - coordinate on the stock where the part beggins.
      Calculator.GetResultLinearPart(partIndex, out tmp, out partLength, out X);
      // Output the part information
      Console.Write("Part= {0}:  X={1};  Length={2}", partIndex, X, partLength);
     }
    }
  }
```

3.3. Cut one size stock.

This example demonstrates how to cut a **2D** rectangular stock panels with size of **2400**x**2000** mm.
Let say we need to cut **9** parts of **640**x**420** mm, **14** parts of **150**x**720** mm and **12** parts of **1000**x**420** mm. In addition the **14** parts of **150**x**720** mm cannot be rotated.

```
// First we create a new instance of the cut engine
CutEngine Calculator = new CutEngine();
Calculator.AddStock(2400, 2000, 5); // Add 5 stocks of 2400x2000 mm
// Add parts we have to cut from the stock:
Calculator.AddPart(640, 420, 9); // 9 parts of 640x420 mm
Calculator.AddPart(150, 720, 14, false); // 14 non-rotatable parts of 150x720 mm
Calculator.AddPart(1000, 420, 12); // 12 parts of 1000x420 mm
// Run the calculation:
string result = Calculator.Execute();
```

Let’s output the information about the parts, the stock indices and locations the parts cut from:

```
double W = 0, H = 0, X = 0, Y = 0;
bool Rotated;
int StockNo;
Console.Write("Part Count={0}", Calculator.PartCount);
for (int iPart = 0; iPart < Calculator.PartCount; iPart++)
{
  if (Calculator.GetResultPart(iPart, out StockNo, out W, out H,
                               out X, out Y, out Rotated))
  {
    // StockNo – stock index the part iPart cut from
    // W,H – part width and height
    // X,Y – coordinates of the top left corner of the part on the stock StockNo
    // If Rotated is true then the part has been roated by 90°
    Console.Write("Part {0}: stock={1}  X={2};  Y={3};  Width={4};  Height={5}",
                  iPart, StockNo, X, Y, W, H);
  }
  else Console.Write("Part {0} was not placed", iPart);
}
```

Also let’s output the information about the cuts we need to make (important for **CNC** machines). Each cut defined by two pairs of coordinates: Cut Start (**X1**, **Y1**) and Cut End (**X2**, **Y2**).

```
int StockNo, iCut, CutsCount;
double Width, Height, X1 = 0, Y1 = 0, X2 = 0, Y2 = 0;
bool active;
// Output guilltoine cuts for each stock
for (StockNo = 0; StockNo < Calculator.StockCount; StockNo++)
{
  Calculator.GetStockInfo(StockNo, out Width, out Height, out active);
  // Stock was not used during calculation and we skip it
  if (!active) continue;
  // First output any trim cuts for the stock StockNo
  CutsCount = Calculator.GetStockTrimCutCount(StockNo);
  for (iCut = 0; iCut < CutsCount; iCut++)
  {
    Calculator.GetStockTrimCut(StockNo, iCut, out X1, out Y1, out X2, out Y2);
  }
  // Now output any actual cuts for the stock StockNo
  CutsCount = Calculator.GetStockCutCount(StockNo);
  for (iCut = 0; iCut < CutsCount; iCut++)
  {
    Calculator.GetStockCut(StockNo, iCut, out X1, out Y1, out X2, out Y2);
  }
}
```

3.4. Cut multiple size stocks.

This example demonstrates how to cut a 2D rectangular stock/panels with different sizes. All parts cannot be rotated.

```
// First we create a new instance of the cut engine
CutEngine Calculator = new CutEngine();
Calculator.AddStock(2000, 2400, 5); // 5 stocks of 2000x24000
Calculator.AddStock(1800, 2000, 5); // 5 stocks of 1800x2000
Calculator.AddStock(1200, 1600, 10); // 10 stocks of 1200x1600
// Add parts we have to cut from the stocks:
Calculator.AddPart(650, 450, 36, false); // 36 non-rotatable parts of 650x450 mm
Calculator.AddPart(650, 732, 24, false); // 24 non-rotatable parts of 650x732 mm
Calculator.AddPart(500, 430, 24, false); // 24 non-rotatable parts of 500x430 mm
Calculator.AddPart(163, 422, 36, false); // 36 non-rotatable parts of 163x422 mm
Calculator.AddPart(444, 363, 36, false); // 36 non-rotatable parts of 444x363 mm
Calculator.AddPart(104, 362, 36, false); // 36 non-rotatable parts of 104x362 mm
// Run the calculation
string result = Calculator.Execute();
```

Let’s iterate by layouts and output the stock sizes used for each layout and parts cut.

```
int stockIndex,stockCount,iPart,iLayout,partCount,partIndex,tmp,iStock;
double width,height,X,Y,W,H;
bool rotated,stockActive;
string Txt;
Console.Write("Used {0} stocks", Calculator.UsedStockCount);
Console.Write("Created {0} different layouts", Calculator.LayoutCount);
// Iterate by each layout and output information about each layout,
// such as number and length of used stocks and part indices cut from the stocks
for (iLayout = 0; iLayout < Calculator.LayoutCount; iLayout++)
{
  Calculator.GetLayoutInfo(iLayout, out stockIndex, out stockCount);
  // Output information about each stock, such as stock Length
  for (iStock = stockIndex; iStock < stockIndex + stockCount; iStock++)
  {
    Calculator.GetStockInfo(iStock, out width, out height, out stockActive);
    Console.Write("Stock={0}:  Width={1}; Height={2}", iStock, width, height);
    // Output the information about parts cut from this stock
    // First we get quantity of parts cut from the stock
    partCount = Calculator.GetPartCountOnStock(iStock);
    // Iterate by parts and get indices of cut parts
    for (iPart = 0; iPart < partCount; iPart++)
    {
      // Get global part index of iPart cut from the current stock
      partIndex = Calculator.GetPartIndexOnStock(iStock, iPart);
      // Get sizes and location of the source part with index partIndex
      Calculator.GetResultPart(partIndex,
                                   out tmp, out W, out H, out X, out Y, out rotated);
      // W,H – widht and height of the part partIndex
      // X,Y – coordinates of the top left corner of the part on the stock iStock
      // If rotated is true then the part has been roated by 90°
      Console.Write("Part={0}; stock={1}; Width={2}; Height={3}; X={4}; Y={5}",
                    partIndex, iStock, W, H, X, Y);
    }
  }
}
```