Linear Programming Library Gipals32.

http://www.optimalon.com

1. General Information.

Gipals32 is a software library to solve the linear problems (LP) containing unlimited number of variables and constraints.

The library provides the following possibilities:

- Create new LP.
- Import new LP from an existing file in MPS format.
- Alter the specified LP by changing variables or constraints coefficients.
- Specify the settings for the linear programming solver to tune the calculation engine according to the user's needs.
- Perform the calculation.
- Getting the results for successful calculations.

Gipals32 is Windows 32 Dynamic Link Library (DLL) and can be easily integrated into any application developed by the different programming tools such as Microsoft Visual C++, Microsoft Visual C# .Net, Microsoft Visual Basic, Microsoft Visual Basic.Net, Borland Delphi and others.

2. Library Interface.

All indices are zero-based in the library. Indices are 0..N-1 for N objects.
The functions are described using C# syntax and it's straightforward to translate it to other languages.

2.1. Specifying and Updating Variables

Gipals32 provides several routines to manipulate with decision variables.

```
int AddVariable(double Cost,  // Objective function coefficient
                double Lower, // Lower bound of variable (ignored if LowerInf <> 0)
                double Upper, // Upper bound of variable (ignored if UpperInf <> 0)
                int LowerInf, // If <> 0 then indicates variable has no lower bound
                int UpperInf);// If <> 0 then indicates variable has no upper bound
                )
```

This function creates a new variable in the LP and sets cost and bounds for this variable. The function returns the variable's index if succeeded and -1 if the maximum size (5,000) has been reached. The function can be used to create a new variable only up to 5,000 variables because behind this size the performance is dramatically slows down. For the LP with more than 5,000 variables the procedure **SetVariableCount** should be used.

```
DeleteVariable(int Index)
```

This procedure deletes the specified variable from LP.

```
int GetVariableCount()
```

The function returns the number of the variables that have been specified in the LP.

```
int SetVariable(int Index,     // Variable index
                double Cost,   // Objective function coefficient
                double Lower,  // Lower bound of variable (ignored if LowerInf <> 0)
                double Upper,  // Upper bound of variable (ignored if UpperInf <> 0)
                int LowerInf,  // If <> 0 then indicates variable has no lower bound
                int UpperInf   // If <> 0 then indicates variable has no upper bound
               )
```

The function setups the parameters of the existing variable **Index**. If the variable doesn't exist then returns 0.

Example 1. Creating new variables using **AddVariable** function:

a) Define a new variable that has cost (objective function coefficient) of 5 and can be any positive number without upper bound. Such variable is called "**normal variable**".

```
AddVariable(5, 0, 0, 0, 1);
```

b) Define a new variable that has cost of -5 and can be any positive number not more than 1000. In this case the upper bound is specified (**1000**) and the indication the variable has the upper bound is set (**0**).

```
AddVariable(5, 0, 1000, 0, 0);
```

c) Define a new variable that has cost of 2.5 and doesn't have any bounds, it's called "**free variable**". In this case the indications the variable has not lower (**1**) and upper bounds is set (**1**).

```
AddVariable(2.5, 0, 0, 1, 1);
```

d) Define a new variable that has cost of 1 and should be any number in between **10** and **150**.

```
AddVariable(1, 10, 150, 0, 0);
```

e) Define a new variable that has cost of 3 and should have only one value **20**. This variable is called "**fixed variable**" and defined be setting lower and upper bound to the same value.

```
AddVariable(1, 20, 20, 0, 0);
```

Example 2. Creating new variables using pair of **SetVariableCount** and **SetVariable** functions. This is the only way to specify linear programs with more than 5,000 variables.

Create 10,000 variables at once:
```
SetVariableCount(10000);
```

Setup properties for first five variables from the previous example. The first parameter indicates the index (in bold) of the variable and the rest of parameters are the same as for **AddVariable** function:
```
SetVariable(0, 5, 0, 0, 0, 1);
SetVariable(1, 5, 0, 1000, 0, 0);
SetVariable(2, 2.5, 0, 0, 1, 1);
SetVariable(3, 1, 10, 150, 0, 0);
SetVariable(4, 1, 20, 20, 0, 0);
```

Function **SetVariable** also can be used to change some properties of the variable for the existing LP after the calculation was done. Let say to change the cost of the third variable (Index = 2) from existing value of **2.5** to **-4.1** one should call:
```
SetVariable(2, -4.1, 0, 0, 1, 1);
```

2.2. Specifying and Updating Constraints

NOTE: Only non-zero constraint elements should be specified.

Constraints form the matrix that has as many columns as variables and as many rows as constraints.
There are three types of constraint signs available:
```
  int sign_Less  = 0;  // <= (Less or Equal)
  int sign_Equal = 1;  //  = (Equal)
  int sign_More  = 2;  // >= (More or Equal)
```

Gipals32 provides several routines to manipulate with the constraints.

```
int AddConstraint(double Right, // Right side of the constraint
                  int Sign      // Sign (sign_Less, sign_Equal, sign_More)
                 )
```
This function creates a new constraint in the LP and sets its right side and sing. Sing can be one of the constants described above.
The function returns the constraint's index if succeeded and -1 if the maximum size (5,000) has been reached. The function can be used to create a new constraint only up to 5,000 constraints because behind this size the performance is dramatically slows down. For the LP with more than 5,000 constraints the procedure **SetConstraintCount** should be used.

```
DeleteConstraint(int Index)
```
This procedure deletes the specified constraint from LP.

```
void SetConstraintCount(int ACount)
```
This procedure creates the specified number of the constraints in the LP. This procedure works faster than function **AddConstraint** and provides the best performance.

```
int GetConstraintCount()
```
The function returns the number of the constraints that have been specified in the LP.

```
int SetConstraint(int Index,     // Index of the constraint
                  double Right,  // Right side of the constraint
                  int Sign       // Sign (sign_Less, sign_Equal, sign_More)
                 )
```

The function setups the right side and the sign of the existing constraint **Index**. If the constraint doesn't exist then returns 0.

```
int SetConstraintElement(int RowIndex,  // Index of the constraint (row in matrix)
                         int VarIndex,  // Index of the variable (column in matrix)
                         double Value   // Value of coefficient
                        )
```

This function assigns the specified **Value** to the coefficient for variable **VarIndex** in the constraint **RowIndex**. In other words it setups the matrix element at position Row = **RowIndex** and Column = **VarIndex** to **Value**. The function returns 1 if succeeded.

The following two functions provide the best performance during the setting up the constraints. These function are recommended for middle and big LPs.

```
int DirectSelectConstraint(int Index     // Index of the constraint
                           )
```

This function prepares the internal LP for specifying (adding elements) the constraint **Index**. It returns 0 if the constraint does not exist.

```
int DirectAddConstraintElement(int Index,    // Index of the variable (column in matrix)
                               double Value // Variable coefficient
                               )
```

This function appends the specified non-zero element the end of the constraint selected by function **DirectSelectConstraint**. Index indicates the variable and Value specifies the variable coefficient. Variable indices MUST follow in an ascending order. At the result the constraint looks like sequence of pair (Index, Value). For example the constraint (**0**:2.3), (**5**:-6.3), (**6**:2.0), (**10**:8.5) where the bold numbers are indices of all non-zero elements of the constraint can be specified by the following sequence of calls:

```
DirectAddConstraintElement(0,  2.3);
DirectAddConstraintElement(5, -6.3);
DirectAddConstraintElement(6,  2.0);
DirectAddConstraintElement(10, 8.5);
```

WARNING! The following calls are wrong and will result in error during the calculation:

```
DirectAddConstraintElement(0,  2.3);
DirectAddConstraintElement(6,  2.0); // Error - Element with index 6 must be specified after element 5.
DirectAddConstraintElement(5, -6.3);
DirectAddConstraintElement(10, 8.5);
```

Example 3. Creating new constraints using **AddConstraint** and **SetConstraintElement** functions:

```
Int Index;
```

a) Define a new constraint: 2***X0*** – 10.2***X3*** + 0.36***X8*** <= 50:

```
Index = AddConstraint(50, sign_Less);
SetConstraintElement(Index, 0, 2);
SetConstraintElement(Index, 3, -10.2);
SetConstraintElement(Index, 8, 0.36);
```

b) Define a new constraint: ***X1*** – ***X4*** + 15***X5*** – 10***X6*** = -88.5

```
Index = AddConstraint(–88.5, sign_Equal);
SetConstraintElement(Index, 1, 1);
SetConstraintElement(Index, 4, -1);
```

```
SetConstraintElement(Index, 5, 15);
SetConstraintElement(Index, 6, -10);
```

Example 4. Constraints from Example 3 created using the method with the best performance **SetConstraintCount, SetConstraint, DirectSelectConstraint** and **DirectAddConstraintElement** functions:

```
// Create two constraints
SetConstraintCount(2);

// Setup the right sides and signs of the constraints
SetConstraint(0, 50, sign_Less);
SetConstraint(1, -88.5, sign_Equal);

// Select the first constraint for direct element insertion
DirectSelectConstraint(0);
// Direct element insertion
DirectAddConstraintElement(0, 2);
DirectAddConstraintElement(3, -10.2);
DirectAddConstraintElement(8, 0.36);

// Select the second constraint for direct element insertion
DirectSelectConstraint(1);
// Direct element insertion
DirectAddConstraintElement(1, 1);
DirectAddConstraintElement(4, -1);
DirectAddConstraintElement(5, 15);
DirectAddConstraintElement(6, -10);
```

NOTE: The only methods to change existing constraints are **SetConstraint** and **SetConstraintElement.**

2.3. Importing LP from MPS file

Gipals32 can import linear program specified as a text file in MPS format:

```
int LoadFromMPS(char[] AFile)
```

Function clears existing LP and loads a new one from the specified text file. Text file must be in MPS format. The function returns the following result:

1 if succeeded.
0 if the file is in wrong format
-1 if the maximum allowed number of variables or constraint reached.

After the LP was successfully imported from the MPS file it can be changed by any routines specified above or calculated right away.

2.4. Calculating

Calculation performs after the LP has been specified.

```
int Calculate(IntPtr ProcessCallback)
```

This function performs the calculation and returns on the following results:

```
int calc_None          =  0;  // Calculation has not been provided yet
int calc_Optimal       =  1;  // Finished with optimal solution
int calc_Infeasible    = -1;  // Failed with primal infeasibility
int calc_DblInfeasible = -2;  // Failed with dual infeasibility
int calc_Unknown       =  2;  // Finished because reached the maximum number of iterations
int calc_Suboptimal    =  3;  // Finished due to slow convergence
int calc_Stoped        =  4;  // Stopped by the user
int calc_Error         = -3;  // Failed due to numerical error
int calc_Unbounded     = -4;  // Failed because the linear program is unbounded
```

In general, the successful results are positive and unsuccessful results are negative.

The function has one parameter that is callback function that provides some additional control and can stop or cancel the whole calculation process. The callback function has the following signature:

```
int Callback(int IterCount,     // Index of the current iteration
             double PrimalObj,  // Primal objective function value
             double DualObj,    // Dual objective function value
             double InfPrimal,  // Primal infeasibility
             double InfDual,    // Dual infeasibility
             double Optimality  // Optimality of the solution
            )
```

This function is an optional function that can stop the iterations with some user-defined checks. When it needs to stop the calculation it should return one of the predefined values:

```
int res_Abort = -1;  // Calculation is canceled
int res_None  =  0;  // Calculation continues
int res_Stop  =  1;  // Calculation is stopped and the results are ready
```

Note: Unfortunately this function is available now only for Visual C++ and Borland Delphi interfaces (you can use **CalculateDlg** that provides progress meter during the calculation).

The following function provides the same calculation as previous one. In addition it shows the progress meter dialog with single button that can interrupt the calculation and return either **`res_Abort`** or **`res_Stop`** value**.**

```
int CalculateDlg(int aType,  // Type of the build-in dialog (prd_None..prd_MeterButton)
                 int aLeft,  // Left screen position of the dialog in pixels
                  int aTop,  // Top screen position of the dialog in pixels
                int aWidth,  // Width of the dialog in pixels
          int aButtonResult,  // Result that returned when the user pressed the button
            char[] aCaption,  // Caption of the dialog
             char[] aButton   // Text on the button
 )
```

The **`aButtonResult`** must be either **`res_Abort`** or **`res_Stop`** value**.** Any other values will be ignored.

The dialog type is defined as the following:

```
int prd_None         = 0; // Empty dialog without the progress meter and without the button.
int prd_Meter        = 1; // Dialog with the progress meter only.
int prd_Button       = 2; // Dialog with the button only.
int prd_MeterButton  = 3; // Dialog with the progress meter and the button.
```

2.5. Getting the Calculation Results

The results are available after the calculation finished with one of successful (positive) states. There are five functions and their usage is obvious:

`double` **`PrimalObjective()`**
Function returns the primal objective function value.

`double` **`DualObjective()`**
Function returns the dual objective function value.

`double` **`PrimalValue(`**`int` **`Index)`**
Function returns value of the primal variable with the specified index. Index varies from 0 to GetVariableCount() – 1.

`double` **`DualValue(`**`int` **`Index)`**
Function returns value of the dual variable with the specified constraint index. Index varies from 0 to GetConstraintCount() – 1.

`double` **`ReducedCostValue(`**`int` **`Index)`**
Function returns value of the primal variable with the specified index. Index varies from 0 to GetVariableCount() – 1.

2.6. Calculation Settings

The following routines control the calculation process and are optional for the most of optimization tasks.

`void` **`SetPreprocessor(`**`int` **`Flags)`**
Function controls the preprocessor's behavior. It can disable the preprocessing or disable/enable some of checks. There are the following constants available for this:

```
int prep_Disable          = 0x00;  // Disable preprocessor
int prep_RemoveFixedVars  = 0x01;  // Remove all fixed variables
int prep_RemoveEmptyCols  = 0x02;  // Remove empty columns
int prep_RemoveSingular   = 0x04;  // Remove singular rows and columns
int prep_RemoveLinearRows = 0x08;  // Remove linear-dependent rows
int prep_RemoveLinearCols = 0x10;  // Remove linear-dependent columns
int prep_ForcedRows       = 0x20;  // Remove forced rows
```

Example 5. Controlling of the preprocessor.
a) Disable preprocessor:

```
SetPreprocessor(prep_Disable);
```

b) Enable removing empty columns only:

```
SetPreprocessor(prep_RemoveEmptyCols);
```

c) Enable removing all fixed variables and linear-dependent rows only:

```
SetPreprocessor(prep_RemoveFixedVars + prep_RemoveLinearRows);
```

d) Enable all preprocessor's checks (**default settings**):

```
SetPreprocessor(prep_RemoveFixedVars + prep_RemoveEmptyCols +
                prep_RemoveSingular +  prep_RemoveLinearRows +
                prep_RemoveLinearCols + prep_ForcedRows);
```

```
void SetTolerances(double Primal,   // Primal tolerance
                   double Dual,     // Dual tolerance
                   double Optimal   // Optimality tolerance
                  )
```

Defines the calculation tolerances and therefore controls the accuracy of the solution. Defaults are: Primal=**1e-8**, Dual=**1e-8**, Optimal=**1e-10**.

`void` **`SetMaxIterationCount(`**`int` **`Value)`**
Function defines the maximum number of iterations (default is 100). If this number is reached but the specified tolerances are not then the calculation stops with **`calc_Unknown`** result.

`void` **`SetScaling(`**`int` **`Value)`**
Function disables / enables the constraints matrix scaling for better numerical stability during the calculation.
If **Value** = **1** then scaling is enabled (default).
If **Value** = **0** then scaling is disabled.

`void` **`SetRefinement(`**`int` **`Value)`**
Function disables / enables the constraints matrix refinement for better numerical stability during the calculation.
If **Value** = **1** then refinement is enabled.
If **Value** = **0** then refinement is disabled (default).

`void` **`SetGondzio(`**`int` **`Value)`**
Function disables / enables the usage of Gondzio correction during the calculation.
If **Value** = **1** then Gondzio correction is enabled.
If **Value** = **0** then Gondzio correction is disabled (default).