Overview: MATLAB as an Automation Client

MATLAB is able to serve as COM-client (on Windows platform only) as described in "MATLAB COM Client Support" section of the MATLAB help. Good example shows how to control MS Excel from MATLAB could be found on MATLAB site:
http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_external/f62644.html#f62659

Accessing Objects

No special action required to let MATLAB know about QuickField type library. The only thing you need to do before writing the MATLAB code is switching working directory to the folder where your code is located.

Each MATLAB variable can hold a handle to a COM object. The handle can be obtained by two ways:

1. A COM object can be directly created by the MATLAB function **actxserver**;
2. An object can be obtained as result of invoking a methods or property of another object.

QuickField provides only one object that should be created directly: the **Application** object. Each program using QuickField shod create a single **Application** object as following:

```
QF = actxserver ('QuickField.Application');
```

As result of above line the *QF* variable holds a handle to the **Application** object. This command either starts QuickField application or returns the handle of the already running application. All other QuickField objects are accessible by properties of **Application** or other objects. E.g. the main window of QuickField can be obtained by the *MainWindow* property of the **Application** object:

```
mainWnd = QF.MainWindow;
```

Using Object's Properties

The MATLAB provides two different syntaxes for setting and getting the value of a property. E.g. when we have to get the value of *DefaultFilePath* property of an **Application** object we can write:

```
thePath = QF.DefaultFilePath;
```

where
QF is a variable holding the handle of Application object,
thePath is a variable receives the value of *DefaultFilePath* property.

The trailing semicolon blocks output to the MATLAB command window.

For setting the new value to a property we can use similar syntax vise versa:

```
QF.DefaultFilePath = 'C:\My Documents\MyProblems';
```

Another syntax for accessing to object's property uses get/set function. The first example may be rewritten as

```
thePath = QF.get ('DefaultFilePath');              % getting the value
QF.set ('DefaultFilePath', 'C:\My Documents\MyProblems');     % setting the value
```

In both cases the first parameter of get/set functions is a string containing the property name. If the property is indexed (i.e. has one or more parameters), the parameters follow by the property name separated by comma. This syntax seems a bit ugly, but it is the only way of using an indexed property.

Invoking Method

Let us create a Point with coordinates (3, 2). It can easily be done by the **PointXY** method of the **Application** object. Coordinates of newly created point are parameters of the PointXY method.

The simple syntax looks as simple as:

```
myPoint = QF.PointXY (3, 2);
```

where *myPoint* variable receves a handle to newly created point; *QF* - is a QuickField.Application object.

Another syntax uses the **invoke** function:

```
myPoint = QF.invoke ('PointXY', 3, 2);
```

similarly to the **get** function the first parameter is the method's name, while other a parameter passed to the calling method.

Using Named Constants

Many properties and methods use named constants (enumeraton members) as a parameter. In MATLAB name the name of a constant should be enclosed in single quotes just like as a string:

```
force = res.GetIntegral ('qfInt_MaxwellForce', win.Contour).Abs;
```

There 'qfInt_MaxwellForce' is a single quoted name of a constant, which is a member of **QfIntegrals** enum.

In the example above actually three ActiveField methods are called:

1. *GetIntegral* method of the **Result** object;
2. Contour property of the **FieldWindow** object;

3. *Abs* property of the **Quantity** object returned by the **GetIntegral** method.

For unclear reasons MATLAB 7.0.1 (R14) Service Pack 1 does not accepts named constant as a parameter of indexed property. So, we have to use its numerical value instead. The numerical values for named constants may be obtained from ActiveField help.

Optional Parameters

Many of QuickField methods and properties have one or more optional parameters. Generally MATLAB not allows to omit optional parameters. You must use them explicitly.

For example, the *Kxx* and *Kyy* properties of the **LabelBlockMS** and **LabelBlockES** objects have an optional parameter *Absolute* of boolean type. By default its value is *false*. In MATLAB when assigning a relative permeability value 1 to the label contents we write:

```
cntBlock.set ('Kxx', false, 1);
cntBlock.set ('Kyy', false, 1);
```

where *cntBlock* is the variable containing a handle of **LabelBlockMS** object.

C-Shaped Magnet (Language: MATLAB)

Lesson 1: Using the existing problem

You can find the complete Mag1 Tutorial sample in the following folders:

- "..\ActiveField\Tutorial\Lesson1\VB_Code\" - the Visual Basic 6.0 version
- "..\ActiveField\Tutorial\Lesson1\VBA_Code\" - VBA version (for MS Excel 2002)
- "..\ActiveField\Tutorial\Lesson1\C#_Code\" - C# version (Visual Studio 7 required)
- "..\ActiveField\Tutorial\Lesson1\C++_Code\" - C++ version with VS7 project.
- "..\ActiveField\Tutorial\Lesson1\VB.NET_Code\" - Visual Basic 7 (Visual Basic .NET) version
- "..\ActiveField\Tutorial\Lesson1\Delphi_Code\" - Borland Delphi version, tested on Borland Delphi 7

The comments below describe writing code on MATLAB 7.0. The code was tested with MATLAB 7.0.1.24704 (R14) Service Pack 1.

Working with a parametric model, generally we have to do the following:

- Step 1: Open an existing QuickField problem :
  the Magh1.pbm problem in the QuickField\Examples folder.
- Step 2: Modifying the geometric model.
  We have to locate the steel keeper block, move it to desired position, rebuild the finite element mesh and save the model.
- Step 3: Modifying the physical properties.
  We might also want to modify some of the physical properties, field sources or boundary conditions. In our case we will change the coercive force of ALNICO magnet.
- Step 4: Solving the problem.
  Solve the problem and obtain results.
- Step 5: Analyzing result: evaluating local and integral quantities.
  In order to get the mechanical force value we have to build a closed contour surrounding the steel keeper and calculate an integral field value.
- Conclusion
  User Interface, the final code and references.

Step 1: Launching QuickField application.

Now we are ready to operate with QuickField objects. The very first object we always get is **QuickField.Application** object. The *Application* object provides access to all other objects, so it is a good idea to declare it as a global level variable in our project.

```
global QF;

function Main()
   ' Here we will write our code
   QF = actxserver ('QuickField.Application');
   QF.MainWindow.Visible = True

  .................
```

The special MATLAB function **actxserver** is used for creating an object of the QuickField.Application type. The handle of created object is assigned to the QF variable. If the *QuickField.exe* is already running, the *actxserver* function returns a running object. If not, it launches the new instance of **QuickField.exe**. Only the Application object should be created by *actxserver* function. All other ActiveField object area created by designated properties of Application and other objects.

For some reasons QuickField may failed to launch. So, it is a good idea to put the call of *actxserver* function inside a try/catch block:

```
try
  QF = actxserver ('QuickField.Application');
catch
  error ('QuickField cannot start');
end
```

```
QF.MainWindow.Visible = true;     % make the QuickField main window visible

% display the welcome message
message = sprintf ('QuickField %s started', QF.Version);
disp (message);
```

The first line describes ***QF*** as a global variable holding an *Application* object. If the *QuickField.exe* is already running, the standard Visual Basic **CreateObject** function returns a running object. If not, it launches the new instance of **QuickField.exe**.

Now you can put the code above into your MATLAB editor window and run it. For our educational purposes the most convenient way is running the code line by line in debug mode. So, set a breakpoint onto first executable line of code (F12) and choose Debug -> Run command (F5). Another way to learn with ActiveField technique is executing command one by one directly entering them into MATLAB command window.

Step 2: Open an existing QuickField documents (problem, geometric model and properties document).

When we have an Application object, open the desired QuickField document is a very straightforward task. There are several documents collection one for each document kind. Consider the following code:

```
global QF;

function MyMacro()
    ' Here we will write our code
    QF = actxserver ('QuickField.Application');
    QF.MainWindow.Visible = true;

    ' Open the Magn1.pbm problem
    prb = QF.Problems.Open ('H:\QuickField\Examples\Magn1.pbm');
```

After executing Problems.Open method the **prb** variable holds an object ofQuickField.Problem type. It becomes the newly problem loaded from the *Magn1.pbm* file.

An argument of the **Open** method is fully qualified name of the problem document. Another possibility is first setting the DefaultFilePath property to the desired folder:

```
QF.DefaultFilePath = 'H:\QuickField\Examples';
prb = QF.Problems.Open ('Magn1.pbm');
```

Step 4: Modifying the geometric model.

Using the QuickField graphic user interface we load the model for editing from the problem tree view. With QuickField programming interface we do almost the same. The LoadModel method loads the Model document associated with the problem into QuickField and then it is accessible by the Model property.

It is worthy to put the geometric manipulation code into a separate function in order to call it with different parameters.

First of all we should to declare a variable as QuickField.Model to hold the *Model* object. Then we get the geometric model object using LoadModel method and Model property of the QuickField Problem object.

Our plan of geometric modification looks like following:

1. Open the geometry model and remove the finite element mesh;
2. Locate and remove the block labeled as 'Steel Keeper' in its previous position;
3. Add four edges constitutes the rectangular steel keeper is a new position;
4. Assign the label to newly created block;
5. Build the mesh, save the ready model, and close it.

```
function MoveKeeper (prb, left, bottom)
% modifies position of the steel keeper
% Global data
%
global skHigh;                  % Keeper's height
global skWidth;                 % Keepers width
% QuickField objects
global QF;                              % the QuickField application
    prb.LoadModel;
    mdl = prb.Model;    % get the geometry model

        %     The Shapes collection that contains all the geometrical entities in the model
        %     provides methods for adding edges and vertices
        mdl.Shapes.RemoveMesh;
        '  Delete the old Steel Keeper
        mdl.Shapes.get ('LabeledAs', '', '', 'Steel Keeper').Delete;
        '  Add the rectangle consisting of four edges
        mdl.Shapes.AddEdge (QF.PointXY (left, bottom), QF.PointXY(left, bottom + skHigh));
        mdl.Shapes.AddEdge (QF.PointXY (left, bottom + skHigh), QF.PointXY(left + skWidth, bottom + skHigh));
        mdl.Shapes.AddEdge (QF.PointXY (left + skWidth, bottom + skHigh), QF.PointXY(left + skWidth, bottom))
        mdl.Shapes.AddEdge (QF.PointXY (left + skWidth, bottom), QF.PointXY(left, bottom));

        '    Locate the newborn block and assign label to it
        foundShapes = mdl.Shapes.get ('Nearest', QF.PointXY (left + skWidth / 2, bottom + skHigh / 2));
        foundShapes.Blocks.Item (1).Label = 'Steel Keeper';
        ' Build the FEM mesh in the entire model
        mdl.Shapes.BuildMesh;
    ' Save the ready geometry model
    mdl.Save;
    ' and close the model window
    mdl.Close;
    ' it is also a good idea to release all non necessary objects
    mdl.release;
```

Most of operation with model we make with the Shapes object - a collection contains all the geometric entities in the model. The Delete method is used to remove the block labeled as "Steel Keeper". Please note the way we use to locate the block: LabeledAs property returns the collection of objects with the same label - just like a mouse click on the label in a problem tree view.

Then we add four edges building the rectangular block. For dimensioning it we use ***left*** and ***bottom*** parameters describing the block position and ***skWidth*** and ***skHigh*** global variables hold block's dimensions. Note that AddEdge method requires two Point objects as parameters. We employ an auxiliary PointXY method of the Application object to create new Point with coordinates we need.

When the edges are built we have to locate a newborn block and assign it the name. The most convenient way to find a block is the Nearest property returns the nearest Block, Edge and Vertex to the given point. The Label property sets and returns the object's label.

Before finish with geometric model, we build the finite element mesh (BuildMesh method) and then save and close the model document.

Step 3: Modifying the physical properties.

Let as imagine what we have to study the influence of the coercive force of ALNICO magnet on the mechanical force. In that case we have to modify label properties. Here you can see a procedure setting given value of coercive force to both magnet labels: "ALNICO up" and "ALNICO down".

```
function SetCoerciveForce (prb, Hc)
% Sets the value of coercive force of the magnet to 'ALNICO up' and 'ALNICO down' block labels
% Global data
global QF; % the QuickField application

    blockLabels = prb.Labels ('qfBlock');
    '    Get the desired label
    lab = blockLabels.invoke ('Item', 'ALNICO up');
    '    Get label's content for editing
    labCnt = lab.Content;
    '    Modify contents
    labCnt.Coercive = QF.PointRA(Hc, pi / 2);
    '    Put modified content back to the label
    lab.Content = labCnt;

    '    Do the same with another label
    lab = blockLabels.invoke ('Item', 'ALNICO down');
    labCnt = lab.Content;
    labCnt.Coercive = QF.PointRA(Hc, -pi / 2);
    lab.Content = labCnt;
    '    Save the data document
    lab.DataDoc.Save;
```

The most important object here are the Label object and its contents, in our case the LabelBlockMS. To modify some physical data we have to find the label in one of Labels collection, get it contents using Contents property, modify properties provided by the **Contents** object, and finally put it back into the *Label* object.

The exact type of the **Contents** object depends upon the kind of QuickField problem (electrostatic, magnetostatic and so on) and geometric object the label is applied to (block, edge or vertex). In our case we are working with block labels for magnetostatic problem, so our contents object is of the LabelBlockMS type.

When finish editing the labels we should save the corresponding data document. Each label knows which document it belongs to: use the DataDoc property.

Step 4: Solving the problem.

When the model and data are ready and corresponding documents have been saved, it's time to start solving. The simplest way to do it is using SolveProblem method of the Problem object. It is a good idea to check if the model and data are ready for solving by the CanSolve property, which returns true if everything seems okay. The Solved property let us known whether the problem is solved.

```
% The problem is fully defined.
% solve it and calculate the magnetic force
if prb.CanSolve
    prb.SolveProblem;
end

if prb.Solved
    prb.AnalyzeResults;
    % Calculation of the mechanical force is discussed later in this chapter
    force (i) = CalculateForce (prb.Result, xPos, yPos);
end
```

This approach is quite enough if the solving time is rather short. When the solving process lasts for several minutes or more, you probably want your application to do something else at the same time. In such situation you can use the asynchronous version of the **SolveProblem** method:

```
prb.SolveProblem (true);
```

Here we set the ***NoWait*** parameter as true instead of its default value false. In such case the **SolveProblem** method returns control just after the solving process started. You can also have more control for the separate solving process by the special SolvingState object:

```
% the state object provides information about solving process
state = prb.SolveProblem (true);
```

Step 5: Analyzing result: evaluating local and integral quantities.

When the problem is solved we can start the postprocessor for viewing and analyzing results. To start the postprocessor use the AnalyzeResults method of the Problem object. After that we can get a Result object gives access to all other postprocessing objects. *Result* object maintains the collection of postprocessor windows. Among them are FieldWindow object for the field picture, XYPlotWindow for xy-charts and TableWindow for tabulating of field quantities along the contour.

As with graphical user interface we can build a Contour in the *FieldWindow*, which can be used for calculating integral quantities, viewing the xy-charts and tabulating.

We put all the code concerning the force calculation in the *CalculateForce* function:

```
    ..........
if prb.Solved
    prb.AnalyzeResults;
    force (i) = CalculateForce (prb.Result, xPos, yPos);
end


function force = CalculateForce (res, left, bottom)
    % Builds the integration contour surrounding the steel keeper and calculates magnetic force
    % Parameters:
    %     res - the QuickField.Result object
    %     left - x-coordinate of bottom-left corner of the steel keeper
    %     bottom - y-coordinate of bottom-left corner of the steel keeper

    % QuickField objects
    global QF;          % the QuickField application
    global skHigh;      % Keeper's height
    global skWidth;     % Keepers width

    win = res.GetFieldWindow (1); % the FieldWindow

    win.Contour.AddLineTo (QF.PointXY (left - 5, bottom - 0.5));
    win.Contour.AddLineTo (QF.PointXY (left + skWidth + 5, bottom - 0.5));
    win.Contour.AddLineTo (QF.PointXY (left + skWidth + 5, bottom + skHigh + 5));
    win.Contour.AddLineTo (QF.PointXY (left - 5, bottom + skHigh + 5));
    win.Contour.AddLineTo (QF.PointXY (left - 5, bottom - 0.5));

    force = res.GetIntegral('qfInt_MaxwellForce', win.Contour).Abs;
```

Here we get a first field picture window from the Result object (the active window is always a first one in the collection) and build the Contour in it, surrounding the steel keeper. To add a line to contour we use the AddLineTo method. The contour appears closed because the last edge's end point is the same as the start point of the first edge.

Having a closed contour we can calculate the force acting on the body(-es) inside it. For this we use a GetIntegral method of the *Problem* object. It returns an integral quantity as Quantity object that wraps physical values of several types. The Abs property of the *Quantity* object return its absolute value (magnitude).

Conclusion

Now our code is almost complete.
We present result of calculation in two forms: a table and a plot.

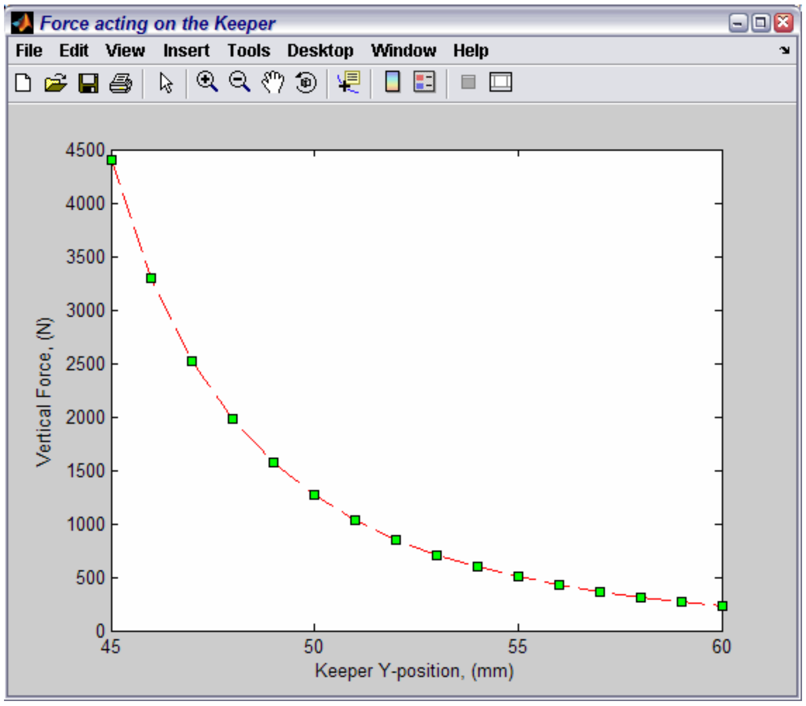
The *DisplayResult* function generates table output to the MATLAB command window

```
function DisplayStepResult (pos, force)
    % Displays calculation result
    disp (sprintf ('Position = %g, Force = %g', pos, force));
```

The code displays a simple xy-plot of force versus position looks like following:

```
% plot the resulting curve
figure ('Name', 'Force acting on the Keeper', 'NumberTitle', 'off');
plot (pos, force, '--rs', 'MarkerEdgeColor','k','MarkerFaceColor', 'g', 'MarkerSize',5);
ylabel ('Vertical Force, (N)');
xlabel ('Keeper Y-position, (mm)');
```

The following MATLAB plot is generated:



The entire code of this sample consist of three MATLAB files located in the ..\ActiveField\Tutorial\Lesson1\MATLAB_Code\ folder and consist of three files:

1. Lesson1.m
   initializes global values, calls DoCalculation function and plots result;
2. DoCalculation.m
   does the main calculation work;
3. RunQF.m
   reusable function for starting QuickField server.

C-Shaped Magnet (Language: MATLAB)

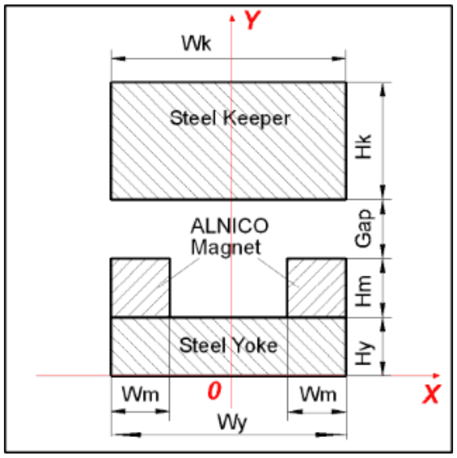
Lesson 2: Creating a new problem

Here we will continue with C-Shaped magnet described in the Lesson 1. To go further, now we consider all the geometric properties as variable. Our model becomes fully parametric.

The discussion is divided into six steps:

- Step 1: Problem formulation
  First of all we learn the problem formulation and the sketch of geometry model.
- Step2: Program structure
  Here we discuss the program structure, user interface and common variables declaration.
- Step 3: Creating a new QuickField problem
  Here we learn how to create a new problem and set its properties.
- Step 4: Building a geometric model.
  Now we have to create a new geometric model document, build the geometric model, assign labels to the blocks and edges, generate the mesh and save it.
- Step 5: Working with data labels.
  The next step is assigning values to the field sources, boundary condition and media properties for each model label.
- Step 6: Analyzing the solution
  Now is time to solve the problem and calculate the magnetic force, acting on the steel keeper.
- What's More:
  How we can develop the program further.

Step 1: Problem formulation



You can see the MATLAB version of this sample in the ..*\ActiveField\Tutorial\Lesson2\MATLAB_Code*\.. folder.

Step2: Program structure

In the following chapters will discuss the MATLAB (m-language) version of our small project. We do not concern graphical user interface in order to focus on interaction between MATLAB and QuickField technique

The MATLAB project consists of three modules:

1. The main module **Lesson2.m** defines input data as a global variables, assigns values to them, calls the main calculation function, and then prints the result.
2. The calculation module **DoCalculation.m** performs all the calculation work;
3. The **RunQF.m** module contains a reusable auxiliary function used for starting QuickField server.

The main module Lesson2.m declares all variables used as input data and assigns default value for each of them:

```
% Global data
%
% QuickField objects
global QF;              % the QuickField application
% Set of geometric dimensions
global yokeWidth;
global yokeHeight;
global maghetWidth;
global magnetHeight;
global keeperWidth;
global keeperHeight;
global airGap;
% Physical data
global Hc;              % Coercive force of the magnet

% Initial values for global data
yokeWidth = 80;
yokeHeight = 20;
maghetWidth = 20;
magnetHeight = 20;
keeperWidth = 80;
keeperHeight = 40;
airGap = 10;
Hc = 147218;

force = DoCalculation ();     % Call the main calclation function
disp (sprintf ('Force = %g', force));     % print the result
```

The **DoCalculation.m** file organised into several function. The first of them, named DoCalculation is the primary m-file function. The rest are subfunctions, only intended to be called by primary function. The primary function orchestrates the calculation process as following:

```
function force = DoCalculation ()
% Builds the QuickField problem from scratch, solve it and calculates the mechanical force.
%
% Global data % QuickField object
global QF;          % the QuickField application

    QF = RunQF (true);
    % We will create the Magn1.pbm problem
    % in Magn1 subfolder under your Matlab project folder
    QF.DefaultFilePath = fullfile (pwd, 'Magn1');

    prb = CreateProblem ();                          % Create a new QuickField problem
    BuildGeometry (prb.get ('ReferencedFile', 0));   % Build the geometry model with FE mesh
    SetData (prb);                                   % Set media properties and boundary conditions
    Solve (prb);                                     % Run solver
    ViewResults (prb);                              % Show the field picture
    force = CalculateForce (prb);                   % Calculate mechanical force

    QF.Quit ();                                     % Finish QuickField program
    QF.release;                                     % Clean up
```

Please note the line that called the BuildGeometry subfunction:

```
BuildGeometry (prb.get ('ReferencedFile', 0));  % Build the geometry model with FE mesh
```

This function receives the file name of the geometry model as an input parameter. To get the geometry file name from the **Problem** object we use the **ReferencedFile** property. In this line the *get*-syntax of calling the property is used, because the **ReferencedFile** is an indexed propery, i.e it requires an input parameter. According to ActiveField rules, the input parameter of the **ReferencedFile** property is of type *QfProblemFiles.* In our case it should have the 'qfModelFile' value. For unclear reasons MATLAB 7.0.1 (R14) Service Pack 1 does not accepts named constant as a parameter of indexed property. So, we have to use its numerical value (0) instead. The numerical values for named constants may be obtained from ActiveField help.

Concerning the internal structure of our small application, we will learn following things:

Step 3: Creating a new QuickField problem.

Creating of the Problem is very straightforward task: first we add a new empty problem by the Add method of the *Problems* collection, and then set desired properties for it. The last thing we have to do with new problem is saving it to the disk file. Newly created problem does not have a file name, so we should use SaveAs method.

```
function prb = CreateProblem ()
% Creates a problem
%
global QF;              % the QuickField application

    prb = QF.Problems.Add ();                   % Create an empty problem
    prb.ProblemType = 'qfMagnetostatics';       % Set the analysis type
    prb.Class = 'qfPlaneParallel';              % Set the geometry class
    prb.LengthUnits = 'qfCentimeters';          % Set length units
    prb.Coordinates = 'qfCartesian';            % Set coordinate system

    % Set geometry and physical data files the problem refers to
    prb.set ('ReferencedFile', 0, 'Magn1.mod');
    prb.set ('ReferencedFile', 1, 'Magn1.dms');

    prb.SaveAs ('Magn1.pbm');                   % Save the new problem
```

Here we use the **ReferencedFile** property in a way discussed below.

Step 4: Building a geometric model.

The BuildGeometry function is a bit lengthy but straightforward. It creates a new geometry model using the **Add** method of the **Model** object.

Then several rectangular blocks are created by using the **AddEdge** method of the **Shapes** object. Creating each rectangle we remember the edge group returned by any of **AddEdge** method in a *shp* variable. This variable, containing collection of edges built during the corresponding **AddEdge** call, is used later to refer to the block, surrounding by the edge.

The entire code of the **BuildGeometry** function is here:

```
function BuildGeometry (modelFileName)
% Creates a geometry model
%
% Global data
% QuickField object
global QF; % the QuickField application
% Set of geometric dimensions
global yokeWidth;
global yokeHeight;
global maghetWidth;
global magnetHeight;
global keeperWidth;
global keeperHeight;
global airGap;

    mdl = QF.Models.Add ();             % Create an empty geometry model
    mdl.SaveAs (modelFileName);         % Save it immediately to establish link to the problem

    % Create the yoke
    shp = mdl.Shapes.AddEdge (QF.PointXY(-yokeWidth / 2, 0), QF.PointXY(yokeWidth / 2, 0));
    mdl.Shapes.AddEdge (QF.PointXY (yokeWidth / 2, 0), QF.PointXY(yokeWidth / 2, yokeHeight));
    mdl.Shapes.AddEdge (QF.PointXY (yokeWidth / 2, yokeHeight), QF.PointXY(-yokeWidth / 2, yokeHeight));
    mdl.Shapes.AddEdge (QF.PointXY (-yokeWidth / 2, yokeHeight), QF.PointXY(-yokeWidth / 2, 0));
    shp.Left.Item (1).Label = 'Steel';

    % Right Magnet
    shp = mdl.Shapes.AddEdge (QF.PointXY(yokeWidth / 2, yokeHeight), ...
        QF.PointXY(yokeWidth / 2, yokeHeight + magnetHeight));
    mdl.Shapes.AddEdge (QF.PointXY(yokeWidth / 2, yokeHeight + magnetHeight), ...
        QF.PointXY(yokeWidth / 2 - maghetWidth, yokeHeight + magnetHeight));
    mdl.Shapes.AddEdge (QF.PointXY (yokeWidth / 2 - maghetWidth, yokeHeight + magnetHeight), ...
        QF.PointXY(yokeWidth / 2 - maghetWidth, yokeHeight));
    shp.Left.Item (1).Label = 'ALNICO down';

    % Left Magnet
    shp = mdl.Shapes.AddEdge(QF.PointXY(-yokeWidth / 2, yokeHeight), ...
        QF.PointXY(-yokeWidth / 2, yokeHeight + magnetHeight));
    mdl.Shapes.AddEdge (QF.PointXY(-yokeWidth / 2, yokeHeight + magnetHeight), ...
        QF.PointXY(-yokeWidth / 2 + maghetWidth, yokeHeight + magnetHeight));
    mdl.Shapes.AddEdge (QF.PointXY(-yokeWidth / 2 + maghetWidth, yokeHeight + magnetHeight), ...
        QF.PointXY(-yokeWidth / 2 + maghetWidth, yokeHeight));
    shp.Right.Item (1).Label = 'ALNICO up';

    % Steel Keeper
    yBase = yokeHeight + magnetHeight + airGap;
    shp = mdl.Shapes.AddEdge (QF.PointXY(-keeperWidth / 2, yBase), ...
        QF.PointXY(keeperWidth / 2, yBase));
    mdl.Shapes.AddEdge (QF.PointXY(keeperWidth / 2, yBase), ...
        QF.PointXY(keeperWidth / 2, yBase + keeperHeight));
    mdl.Shapes.AddEdge (QF.PointXY(keeperWidth / 2, yBase + keeperHeight), ...
        QF.PointXY(-keeperWidth / 2, yBase + keeperHeight));
    mdl.Shapes.AddEdge (QF.PointXY(-keeperWidth / 2, yBase + keeperHeight), ...
        QF.PointXY(-keeperWidth / 2, yBase));
    shp.Left.Item (1).Label = 'Steel Keeper';

    % Surrounding air
    yBase = yokeHeight + magnetHeight + airGap + keeperHeight;
    xBase = (yokeWidth + keeperWidth) * 1.5;
    shp = mdl.Shapes.AddEdge (QF.PointXY(-xBase, -yBase), QF.PointXY(xBase, -yBase));
    mdl.Shapes.AddEdge (QF.PointXY(xBase, -yBase), QF.PointXY(xBase, 2 * yBase));
    mdl.Shapes.AddEdge (QF.PointXY(xBase, 2 * yBase), QF.PointXY(-xBase, 2 * yBase));
    mdl.Shapes.AddEdge (QF.PointXY(-xBase, 2 * yBase), QF.PointXY(-xBase, -yBase));
    shp.Left.Item (1).Label = 'Air';

    outerEdges = mdl.Shapes.get ('Boundary', 0);
    for i = 1 : outerEdges.Count
        outerEdges.Item (i).Label = 'Zero';
    end

    % Set Spacing
    sp = min (magnetHeight, airGap) / 5;             % The spacing value
    mdl.Shapes.get ('LabeledAs', '', '', 'Steel Keeper').Spacing = sp;
    mdl.Shapes.get ('LabeledAs', '', '', 'Steel').Spacing = sp;
    mdl.Shapes.get ('LabeledAs', '', '', 'ALNICO up').Spacing = sp;
    mdl.Shapes.get ('LabeledAs', '', '', 'ALNICO down').Spacing = sp;
    mdl.Shapes.get ('LabeledAs', '', 'Zero', '').Spacing = sp * 4;

    % Generate the mesh
    mdl.Shapes.BuildMesh ();

    % Save the complete model
&    mdl.Save ();
```

Let us walk through the code step by step:

First we create a new empty model document and just save it. For saved document we use the filename that the problem refers to. So, just after saving QuickField establishes the link between problem and model documents. The benefit of establishing the link early is that the problem setting for coordinates and length unit will be applied to the model automatically.

```
    function BuildGeometry (modelFileName)
        .........
        mdl = QF.Models.Add ();             % Create an empty geometry model
        mdl.SaveAs (modelFileName);         % Save it immediately to establish link to the problem

        ........
```

NNow we are ready to start building the edges constituting our model.

```
    function BuildGeometry (modelFileName)

        ........
            % the Yoke
                ........
            % Right Magnet
                ........
            % Left Magnet
                ........
            % Steel Keeper
                ........
            % Surrounding Air
                ........
            % Set Mesh Spacing
                ........
            % Generate the Mesh br
            mdl.Shapes.BuildMesh ();

        % Save the complete model
        mdl.Save
```

The comments % (in green) outlines our next work with several parts of model geometry. When finish with building the model we will get the model picture to show it on the main screen (see discussion later) and save the model document again, this time by simple Save method.

Now consider building of the steel yoke block. We use here the geometric dimension variables, declared at the global level and set by user in the

```
        % Create the yoke
        shp = mdl.Shapes.AddEdge (QF.PointXY(-yokeWidth / 2, 0), QF.PointXY(yokeWidth / 2, 0));
        mdl.Shapes.AddEdge (QF.PointXY (yokeWidth / 2, 0), QF.PointXY(yokeWidth / 2, yokeHeight));
        mdl.Shapes.AddEdge (QF.PointXY (yokeWidth / 2, yokeHeight), QF.PointXY(-yokeWidth / 2, yokeHeight));
        mdl.Shapes.AddEdge (QF.PointXY (-yokeWidth / 2, yokeHeight), QF.PointXY(-yokeWidth / 2, 0));
        shp.Left.Item (1).Label = 'Steel';

            ..........
```

There is the sequence of four **AddEgde** command builds a rectangular block. Please, note the way used to assign label to the block: we remember an edge (generally speaking, group of edges) built by the first command in a ***shp*** variable. When the block is completely built, we can refer to it as a block, located from the left side of the ***shp*** edge by the shp.Left property. The Label property is used to assigning label to the block.

Please note the syntax we use for assigning the label to a block:

```
   shp.Left.Item (1).Label = 'Steel';
```

In other languages, like Visual Basic we prefer to write something like

```
   shp.Left.Label = "Steel"     ' works well in Visual Basic, but not in MATLAB
```

MATLAB does not allow using the **Label** property of a **ShapeRange** object for assigning the label to whole group of geometric shapes. Instead, we have to assign the label to each geometric shape individually. Access to individual item in a collection is provided by the **Item** method. We use *Item (1)* syntax if we know in advance that the collection contains only one element. If this is not the case, write the loop:

```
        outerEdges = mdl.Shapes.get ('Boundary', 0);
        for i = 1 : outerEdges.Count
            outerEdges.Item (i).Label = 'Zero';
        end
```

The fragment of code above refers to edges constitute the outer border of the area by Boundary property of the *Shapes* object. We need to assign them a ***Zero*** label.

Our next step is to set the mesh spacing values to some vertices and that build the mesh. For simplicity sake we employ a straightforward strategy: assign a small spacing value to all corners of the steel parts and a big spacing (four times bigger than small one) to the outer corners.

```
        % Set Spacing
        sp = min (magnetHeight, airGap) / 5;              % The spacing value
        mdl.Shapes.get ('LabeledAs', '', '', 'Steel Keeper').Spacing = sp;
        mdl.Shapes.get ('LabeledAs', '', '', 'Steel').Spacing = sp;
        mdl.Shapes.get ('LabeledAs', '', '', 'ALNICO up').Spacing = sp;
        mdl.Shapes.get ('LabeledAs', '', '', 'ALNICO down').Spacing = sp;
        mdl.Shapes.get ('LabeledAs', '', 'Zero', '').Spacing = sp * 4;
```

Another point worth to note at is using the **LabeledAs** property. MATLAB requires explicitly using all parameters, including optional ones, so all all three string parameter of this property should be given. As usually with an indexed property, the *get*-syntax form should be used.

When the mesh spacing is set, the mesh itself is generated by a single command: the BuildMesh method of the *Shapes* object.

```
        % Generate the mesh
        mdl.Shapes.BuildMesh ();
```

Step 5: Working with data labels.

Our current task is to set appropriate physical values to all the labels we have defined in the model. First have a look at the whole procedure below, and then we discuss some important points.

```
   function SetData (prb)
   % Set the media properties, filed sources and boundary conditions
   %
   % Global data
   % QuickField object
   global QF; % the QuickField application
   % Physical data
   global Hc; % Coercive force of the magnet

        % First set properties for block labels
        blockLabels = prb.get ('Labels', 3);
        for i = 1 : blockLabels.Count;
            lab = blockLabels.Item (i);
            cntBlock = lab.Content;
            switch lab.Name
                case 'Air'
                    cntBlock.set('Kxx', false, 1);
                    cntBlock.set('Kyy', false, 1);

                case {'Steel', 'Steel Keeper'}
                    spl = cntBlock.CreateBHCurve ();
                    spl.Add (QF.PointXY(0.73, 400));
                    spl.Add (QF.PointXY(0.92, 600));
                    spl.Add (QF.PointXY(1.05, 800));
                    spl.Add (QF.PointXY(1.15, 1000));
                    spl.Add (QF.PointXY(1.28, 1400));
                    spl.Add (QF.PointXY(1.42, 2000));
                    spl.Add (QF.PointXY(1.52, 3000));
                    spl.Add (QF.PointXY(1.58, 4000));
                    spl.Add (QF.PointXY(1.6, 6000));
                    cntBlock.Spline = spl;

                case {'ALNICO up', 'ALNICO down'}
                    if strcmp(lab.Name, 'ALNICO up')
                        cntBlock.Coercive = QF.PointRA (Hc, pi / 2);
                    else
                        cntBlock.Coercive = QF.PointRA(Hc, -pi / 2);
                    end

                    spl = cntBlock.CreateBHCurve;
                    spl.Add (QF.PointXY(0.24, 27818 - Hc));
                    spl.Add (QF.PointXY(0.4, 47748 - Hc));
                    spl.Add (QF.PointXY(0.5, 67641 - Hc));
                    spl.Add (QF.PointXY(0.6, 93504 - Hc));
                    spl.Add (QF.PointXY(0.71, 127324 - Hc));
                    spl.Add (QF.PointXY(0.77, 147218 - Hc));
                    cntBlock.Spline = spl;
            end % of switch

            lab.Content = cntBlock;
        end % of the loop

        % The only edge label
        cntEdge = prb.get ('Labels' , 2).Item ('Zero').Content;
        cntEdge.Dirichlet = 0;
        prb.get ('Labels' , 2).Item ('Zero').Content = cntEdge;

        % Saving data document
        prb.DataDoc.Save;
```

To look through the block labels defining in the model we employ the *for-end* loop. There are several way to get the label collection: from the *Problem* object or from the *DataDoc* object, represents the QuickField data document (as you remember, with a problem can be associated up to two data documents). In our case, the data document is now empty, and the only way to get the collection of labels is the Labels property of the *Problem* object. It is used with parameter of QfShapes type denoted which of three collection we want to iterate. Note the MATLAB specific: we have to use numerical value 3 instead of named constant *'QfShapes'* as a parameter of indexed property.

All operations with the content of an individual label we do with an object, accessible by the Content property of the *Label* object. Exact type of that object depends upon problem and label types as described in the DataDoc topic. When finished with the *Content*, we must put it back to the parent *Label* by assigning it to the *Label*'s **Content** property.

```
            lab = blockLabels.Item (i);
            cntBlock = lab.Content;
            ...................
            lab.Content = cntBlock;
```

Now lets consider parameters setting for each individual label. The most simple looks the code for linear media, like "Air" label:

```
        case 'Air'
            cntBlock.set('Kxx', false, 1);
            cntBlock.set('Kyy', false, 1);
```

Here we have to assign the relative magnetic permeability value 1 to both components of the permeability tensor. The boolean parameter of the Kxx and Kyy property defines whether the permeability value is absolute (true) or relative (false). In MATLAB this optional parameter cannot be omitted.

Working with a label describing the saturated media, we have to create and define a Spline object.

```
                case {'Steel', 'Steel Keeper'}
                    spl = cntBlock.CreateBHCurve ();
                    spl.Add (QF.PointXY(0.73, 400));
                    spl.Add (QF.PointXY(0.92, 600));
                    spl.Add (QF.PointXY(1.05, 800));
                    spl.Add (QF.PointXY(1.15, 1000));
                    spl.Add (QF.PointXY(1.28, 1400));
                    spl.Add (QF.PointXY(1.42, 2000));
                    spl.Add (QF.PointXY(1.52, 3000));
                    spl.Add (QF.PointXY(1.58, 4000));
                    spl.Add (QF.PointXY(1.6, 6000));
                    cntBlock.Spline = spl;
```

The label is empty now, so we have to first create a new spline by the CreateBHCurve method. Then we add each spline node individually by the Add method and finally assign the spline to the label's content.

With a non-linear permanent magnet we should also set the coercive force value. On the code fragment below we first set the coercive force value and then add the spline nodes. In this case we must subtract the coercive force from the magnetic field value to put the curve into the second quadrant.

```
               case {'ALNICO up', 'ALNICO down'}
                    if strcmp(lab.Name, 'ALNICO up')
                        cntBlock.Coercive = QF.PointRA (Hc, pi / 2);
                    else
                        cntBlock.Coercive = QF.PointRA(Hc, -pi / 2);
                    end

                    spl = cntBlock.CreateBHCurve;
                    spl.Add (QF.PointXY(0.24, 27818 - Hc));
                    spl.Add (QF.PointXY(0.4, 47748 - Hc));
                    spl.Add (QF.PointXY(0.5, 67641 - Hc));
                    spl.Add (QF.PointXY(0.6, 93504 - Hc));
                    spl.Add (QF.PointXY(0.71, 127324 - Hc));
                    spl.Add (QF.PointXY(0.77, 147218 - Hc));
                    cntBlock.Spline = spl;
```

It is possible first to define the curve, located in the first quadrant, and then set the coercive force.

With edge label we do not need to look through the edge labels collection because we have only one edge label. The following code puts zero Dirichlet condition to the "Zero" label:

```
        % The only edge label
        cntEdge = prb.get ('Labels' , 2).Item ('Zero').Content;
        cntEdge.Dirichlet = 0;
        prb.get ('Labels' , 2).Item ('Zero').Content = cntEdge;

        % Saving data document
        prb.DataDoc.Save;
```

The Dirichlet property set and returns Dirichlet boundary condition that do not depends upon coordinates.

The last line of the ***SetData*** procedure saves the modified data document. It has its file name assigned when the problem was created, so we only need a Save method.

Step 6: Analyzing the solution

When the model and data are ready, we are able to solve a problem and start to analyze result:

```
    function Solve (prb)
    % Solves the problem
    %
    % Global data
    global QF;          % the QuickField application

        if prb.CanSolve
            prb.SolveProblem;
        end
```

The AnalyzeResults method loads the solution for analyzing and creates the first FieldWindow object, represents a field picture window. After that we can get the Result object, which gives access to analyzing capabilities.

In our program we have to do two tasks with the problem solution: adjust the field picture and calculate the force acting to the steel keeper. To get more informative field picture we employ the **Zoom** method of the **FieldPicture** object, receiving two parameters of the **Point** type.

```
    function ViewResults (prb)
    % Solves the problem and starts the postprocessor
    %
    % Global data
    % QuickField object
    global QF;      % the QuickField application
    % Set of geometric dimensions
    global yokeWidth;
    global yokeHeight;
    global maghetWidth;
    global magnetHeight;
    global keeperWidth;
    global keeperHeight;
    global airGap;

        if prb.Solved
            prb.AnalyzeResults;
        end

        res = prb.Result;
        if ~isinterface (res)
            disp ('error: Cannot get problem result');
            return;
        end

        fieldWin = res.GetFieldWindow (1);
        fieldWin.WindowState = 'qfNormal';
        fieldWin.Height = fieldWin.Width + 10;

        bottomLeft = QF.PointXY (-yokeWidth / 2 - maghetWidth, -magnetHeight);
        topRight = QF.PointXY (yokeWidth / 2 + maghetWidth, ...
            yokeHeight + magnetHeight + airGap + keeperHeight + magnetHeight);
        fieldWin.Zoom (bottomLeft, topRight);
```

To calculate the mechanical force we have to create a contour surrounding the keeper's body. Each FieldWindow object always possesses only one the Contour, even an empty one. We get the Contour object from our FieldWindow object called ***fieldWin*** and use its AddLineTo method to add lines to the contour. The last **AddLineTo** connects the last contour point with its starting point.

When the closed contour oriented counter clockwise is ready, we can use the GetIntegral method of the *Result* object to calculate desired integral value. The force value is really a vector quantities, so being interested only in its absolute value we use its Abs property.

```
function force = CalculateForce (prb)
% Solves the problem and starts the postprocessor
%
% Global data
% QuickField object
global QF;     % the QuickField application
% Set of geometric dimensions
global yokeWidth;
global yokeHeight;
global maghetWidth;
global magnetHeight;
global keeperWidth;
global keeperHeight;
global airGap;

    force = 0;
    res = prb.Result;
    if ~isinterface (res)
        return;
    end

    fieldWin = res.GetFieldWindow (1);
    cont = fieldWin.Contour;

    cont.AddLineTo (QF.PointXY (-keeperWidth / 2 - maghetWidth, ...
        yokeHeight + magnetHeight + airGap / 2));
    cont.AddLineTo (QF.PointXY (keeperWidth / 2 + maghetWidth, ...
        yokeHeight + magnetHeight + airGap / 2));
    cont.AddLineTo (QF.PointXY (keeperWidth / 2 + maghetWidth, ...
        yokeHeight + magnetHeight + airGap + keeperHeight * 1.5));
    cont.AddLineTo (QF.PointXY (-keeperWidth / 2 - maghetWidth, ...
        yokeHeight + magnetHeight + airGap + keeperHeight * 1.5));
    cont.AddLineTo (QF.PointXY (-keeperWidth / 2 - maghetWidth, ...
        yokeHeight + magnetHeight + airGap / 2));

    force = res.GetIntegral('qfInt_MaxwellForce', cont).Abs;

    % final cleanup
    cont.Delete (true);
    cont.release;
    fieldWin.release;
```

What's More

This very simple application can be considered as an prototype for more developed custom project intended to automate analyzing of some parameterized model. To do it more realistic we obviously have to develop some code for validation user's input. Probably it will be useful to allow the customer to build his or her model from some parameterized "building blocks", implement some methods for optimization an much more.