Interfacing to Zemax OpticStudio from Mathematica

Beginning with the release of OpticStudio 15, OpticStudio supports an new extension programming capability, ZOS-API, that allows external applications to connect to OpticStudio by means of a .NET interface. Two types of connections are supported: 'standalone", in which the external application starts its own copy of OpticStudio with which to interact; and 'inherent', in which OpticStudio is already running and will call the external application.

This article provides an example of the standalone method, in which the external application is Mathematica. A Mathematica notebook is used as a user interface and scripting language. It will start an OpticStudio session, load an existing lens file, manipulate that lens file to alter the lens design, perform an analyses, and obtain and process the results to provide information not directly available through OpticStudio.

Information on using .NET/Link, which is the Mathematica .NET interface, can be found at

http://reference.wolfram.com/language/NETLink/tutorial/Overview.html

or through the help system of a running Mathematica notebook.

This example was developed using OpticStudio 18.7 and Mathematica 11.3 running on Windows 7, 64-bit. It was developed by closely following the examples in the first release of the Zemax document "ZOS-API Documentation.pdf."

David Keith
May 19, 2015

Updated November 3, 2018: The directory location of "ZOSAPI_NetHelper.dll" has changed since ZOS15

Some observations

For those not familiar with Mathematica

- (* This is a comment in Mathematica. *)
- f[x_]:=x^2 is a function in Mathematica.
- In[1]: marks an input cell.

- Out[1]: marks the corresponding output.
- Terminating an input with a semicolon suppresses printing of the output.
- Semicolons can also be used to put a sequence of *Mathematica* expressions in a single cell.

.NET Syntax in Mathematica

The Wolfram Language syntax for calling .NET methods and accessing fields is very similar to the syntax used in C# and Visual Basic .NET. The following box compares the Wolfram Language and C # ways of calling constructors, methods, properties, fields, static methods, static properties, and static fields. You can see that Wolfram Language programs that use .NET are written in almost exactly the same way as C# (or VB .NET) programs, except the Wolfram Language uses [] instead of () for arguments and @ instead of the . (dot) as the "member access" operator.

An exception is that for static methods, the Wolfram Language uses the context mark ` in place of the dot used by C# and VB. This also parallels the usage in those languages, as their use of the dot in this circumstance is really as a scope resolution operator (like :: in C++). Although the Wolfram Language does not use this terminology, its scope resolution operator is the context mark. .NET namespace names map directly to the Wolfram Language's hierarchical contexts.

	Constructors
C#:	`MyClass obj = new MyClass(args);`
Wolfram Language:	`obj = NETNew["MyClass", args];`
	Methods
C#:	`obj.MethodName(args);`
Wolfram Language:	`obj@MethodName[args]`
	Properties and fields
C#:	`obj.PropertyOrFieldName = 1;` `value = obj.PropertyOrFieldName;`
Wolfram Language:	`obj@PropertyOrFieldName = 1;` `value = obj@PropertyOrFieldName;`
	Static methods
C#:	`MyClass.StaticMethod(args);`
Wolfram Language:	`MyClass`StaticMethod[args];`
	Static properties and fields
C#:	`MyClass.StaticPropertyOrField = 1;` `value = MyClass.StaticPropertyOrField;`
Wolfram Language:	`MyClass`StaticPropertyOrField = 1;` `value = MyClass`StaticPropertyOrField;`

Appearances

This is a cell which is evaluated and produces an output

In[1]:=
```
2 + 3
```
Out[1]=
```
5
```

For this input cell the output is suppressed, but the cell is still executed

In[2]:=
```
x = 3;
```

Set default options and directory paths

In this section we set some convenient default options for *Mathematica* functions, set the default working directory to that from which the notebook was loaded, and define some directory paths for later use.

In[3]:=
```
(* make nice plots *)
SetOptions[ListLinePlot, PlotRange → All,
  PlotTheme -> "Scientific", ImageSize → 500, LabelStyle → 14];
```

In[4]:=
```
(* stop InterpolatingFunction messages during FindRoot *)
Off[InterpolatingFunction::dmval];
```

In[5]:=
```
(* set a default directory *)
SetDirectory[notebookDirectory = NotebookDirectory[]];
```

In[6]:=
```
(* path to the ZOS installation *)
OSPath = "c:\\program files\\Zemax OpticStudio\\";
```

In[7]:=
```
(* path to the sample directory *)
samplePath = $UserDocumentsDirectory <> "\\Zemax\\Samples\\Sequential\\Objectives\\"
```
Out[7]=
```
C:\Users\David\Documents\Zemax\Samples\Sequential\Objectives\
```

In[8]:=
```
(* path to the ZOA-API Libraries directory *)
librariesPath = $UserDocumentsDirectory <> "\\Zemax\\ZOS-API\\Libraries\\"
```
Out[8]=
```
C:\Users\David\Documents\Zemax\ZOS-API\Libraries\
```

Install NetLink

We first need to install Mathematica's NETLink context.

In[9]:=
```
Needs["NETLink`"]
```

In[10]:=
```
InstallNET[]
```

Out[10]=
```
LinkObject[ Name: C:\Program Files\Wolfram Research\Mathema
            Link mode: Listen ]
```

Load Assemblies

Following the Zemax documentation on ZOS-API we load the three ZOS-API assemblies using the *Mathematica* function LoadNETAssembly.

In[11]:=
```
helper = LoadNETAssembly[librariesPath <> "\\ZOSAPI_NetHelper.dll"]
```

Out[11]=
```
NETAssembly[ZOSAPI_NetHelper, 1]
```

In[12]:=
```
zosapi = LoadNETAssembly[OSPath <> "ZOSAPI.dll"]
```

Out[12]=
```
NETAssembly[ZOSAPI, 2]
```

In[13]:=
```
interfaces = LoadNETAssembly[OSPath <> "ZOSAPI_Interfaces.dll"]
```

Out[13]=
```
NETAssembly[ZOSAPI_Interfaces, 3]
```

A quick aside -- Use NETTypeInfo to get information

Now that we've loaded assemblies, here is something very useful. *Mathematica .NET/Link* has a function called NETTypeInfo which will allow you to quickly display information about the methods, properties, fields, and so on that exist in a given .NET type. I use this function by passing it each new type, as I go, on down the structure. The ZOS-API fields, properties, methods, etc, are named very logically. You can usually figure out what you can do with an object by passing it to NETTypeInfo and reading the results.

For example

In[14]:=
```
NETTypeInfo[zosapi]
```

```
Assembly: ZOSAPI
Full Name: ZOSAPI, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
Location: c:\program files\Zemax OpticStudio\ZOSAPI.dll

• Classes
class ZOSAPI.ZOSAPI_Connection

• Interfaces
interface ZOSAPI.IZOSAPI_Events
```

Connect and create an application and a primary system

In this section we initialize the interface, make the connection to it from *Mathematica*, and create an application, which is an instance of ZOS15.

Initialize

ZOSAPI_NetHelper is a static class. We need to load the ZOSAPI_Initializer method and call it as a static method.

In[15]:=
```
LoadNETType["ZOSAPI_NetHelper.ZOSAPI_Initializer"]
```

Out[15]=
```
NETType[ZOSAPI_NetHelper.ZOSAPI_Initializer, 1]
```

In[16]:=
```
ZOSAPIUInitializer`Initialize[]
```

Out[16]=
```
True
```

Make a connection and application

In[17]:=
```
theConnection = NETNew["ZOSAPI.ZOSAPI_Connection"]
```

Out[17]=
```
« NETObject[ZOSAPI.ZOSAPI_Connection] »
```

In[18]:=
```
theApplication = theConnection@CreateNewApplication[]
```

Out[18]=
```
« NETObject[ZemaxUI.Common.ViewModels.ZOSAPI_Application] »
```

Check the license status

Mathematica retrieves the enum license type as a .NET object. These are converted to enum Integers for comparison.

In[19]:=
```
license = NETObjectToExpression[theApplication@LicenseStatus]
```

Out[19]=
```
5
```

In[20]:=
```
licenseType = Switch[license,
  0, "Unknown",
  1, "KeyNotWorking",
  2, "NewLicense",
  3, "StandardEdition",
  4, "ProfessionalEdition",
  5, "PremiumEdition",
  6, "TooManyInstances",
  _, "InvalidResponse"
 ]
```

Out[20]=
```
PremiumEdition
```

Create a Primary System and load a lens file

In this section we create the Primary System and load the lens file.

In[21]:=
```
primarySystem = theApplication@PrimarySystem[]
```

Out[21]=
```
« NETObject[ZemaxUI.ZOSAPI.ZemaxSystem] »
```

The lens file name as a string

In[22]:=
```
(* the name of the lens *)
lens = "Cooke 40 degree field.zmx";
```

Join it to the path to create a full name

In[23]:=
```
lensFile = FileNameJoin[{samplePath, lens}]
```

Out[23]=
```
C:\Users\David\Documents\Zemax\Samples\Sequential\Objectives\Cooke 40 degree field.zmx
```

The LoadFile method is used to load the file

In[24]:=
```
(* load the lens *)
primarySystem@LoadFile[lensFile, False]
```

Out[24]=
```
True
```

Create the LDE

The LDE is the Lens Data Editor. We will create and use an instance of the LDE to modify the lens as needed.

In[25]:=
```
lde = primarySystem@LDE
```

Out[25]=
```
« NETObject[ZemaxUI.ZOSAPI.Editors.ZOSAPI_LDE] »
```

Functions to work with the LDE: Get and set a row thickness

This section defines several functions for working in the LDE. While it would be possible to do all of the work using only ZOS-API calls, the work is easier and the notebook more readable if we define *Mathematica* functions for the purposes needed. These functions can be saved separately to be used again later. Eventually a large collection of them could be assembled into a *Mathematica* "Package" which could be loaded for use in any notebook, without needing to define them again. In this example, we'll be manipulating a surface thickness, so we define functions to get and set a row thickness. They are demonstrated by way of example.

In[26]:=
```
getRowThickness[rowNumber_] := lde@GetSurfaceAt[rowNumber]@Thickness
```

In[27]:=
```
row6Thickness = getRowThickness[6]
```

Out[27]=
```
42.2078
```

In[28]:=
```
setRowThickness[rowNumber_, thickness_] :=
 lde@GetSurfaceAt[rowNumber]@Thickness = thickness
```

In[29]:=
```
setRowThickness[6, 45]
```

Out[29]=
```
45
```

In[30]:=
```
getRowThickness[6]
```

Out[30]=
```
45.
```

In[31]:=
```
(* restore to original thickness *)
setRowThickness[6, row6Thickness]
```

Out[31]=
```
42.2078
```

Create an FFTMTF analysis

In this example, we will be using an FFTMTF analysis as we manipulate the back focal distance of the lens. We will also use *Mathematica* to analyze the MTF data produced by OpticStudio to produce some derived information.

Create an Analyses

In[32]:=
```
fftMtf = primarySystem@Analyses@NewUFftMtf[]
```

Out[32]=
```
« NETObject[ZemaxUI.ZOSAPI.Analysis.Mtf.A_FftMtf] »
```

Set the maximum spatial frequency to a convenient value by setting a property of the analysis

In[33]:=
```
maxFrequency = 60.;
```

In[34]:=
```
(* set the maximum spatial frequency *)
fftMtf@GetSettings[]@MaximumFrequency = maxFrequency
```

Out[34]=
```
60.
```

And define some Mathematica functions that operate with it

Run an FFT and wait for completion

In[35]:=
```
runFftMtf := fftMtf@ApplyAndWaitForCompletion[]
```

Get results from the analysis

This function returns results in the same format as the DataSeries structure of ZOS-API
There is a list for each field in use.
Each field list has three items:
Description
A list of the spatial frequencies
A corresponding list of pairs, each pair being {tangential modulus, saggital modulus }

In[36]:=
```
getFftMtfResults := Module[{getXYvalues, results, dataSeries},
  getXYvalues[ds_] := {ds@Description, ds@XData@Data, ds@YData@Data};
  results = fftMtf@GetResults[];
  dataSeries = results@DataSeries;
  getXYvalues /@ dataSeries
 ]
```

Parse the output of getFftMtfResults into a more convenient form

These functions expects the results of getFftMtfResults and are to be applied to a single field list.

parseToPairLists restructures the lists as follows :
Description
A list of pairs as {spatial frequency, modulus of the tangential component},
A list of pairs as {spatial frequency, modulus of the sagittal component}.

In[37]:=
```
parseToPairsLists[{description_, xList_, ysList_}] := Module[{l1, l2},
   l1 = {xList, ysList[[All, 1]]} // Transpose;
   l2 = {xList, ysList[[All, 2]]} // Transpose;
   {description, l1, l2}
  ];
```

parseToRMSlists restructures the lists as follows :
Description
A list of pairs as {spatial frequency, RMS of the modulus of the tangential and sagittal components}

In[38]:=
```
parseToRMSlists[{description_, frequencies_, pairsList_}] :=
  Module[{rmsList, normalizedRmsList},
   rmsList = Norm /@ pairsList;
   normalizedRmsList = rmsList / Max[rmsList];
   {description, Transpose[{frequencies, normalizedRmsList}]}
  ];
```

Combine it all: Run an MTF, get the results, parse to lists, and plot it

Run the MTF

In[39]:=
```
runFftMtf
```

For future use, just get the field descriptions

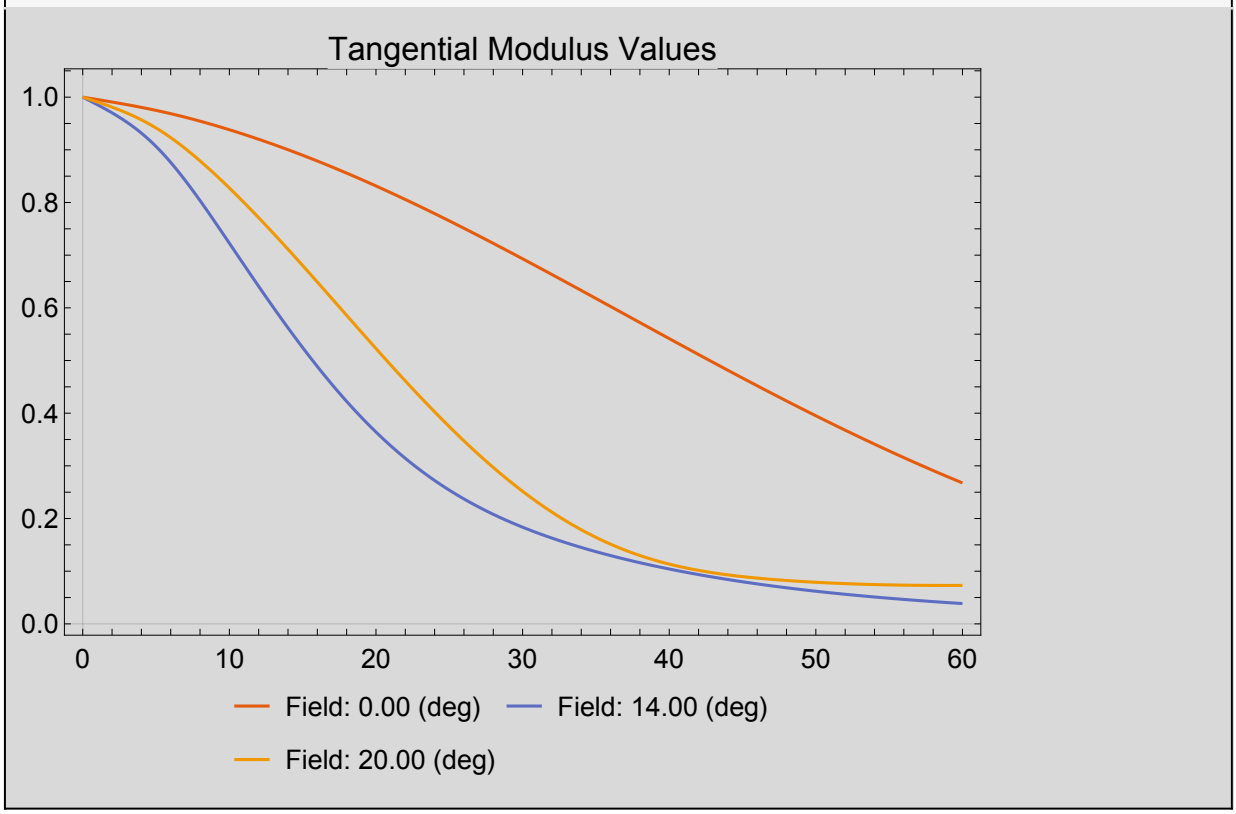
In[40]:=
```
fieldDescriptions = (parseToPairsLists /@ getFftMtfResults)[[All, 1]]
```

Out[40]=
```
{Field: 0.00 (deg), Field: 14.00 (deg), Field: 20.00 (deg)}
```

Get the output, parse to lists, and plot the MTF curves

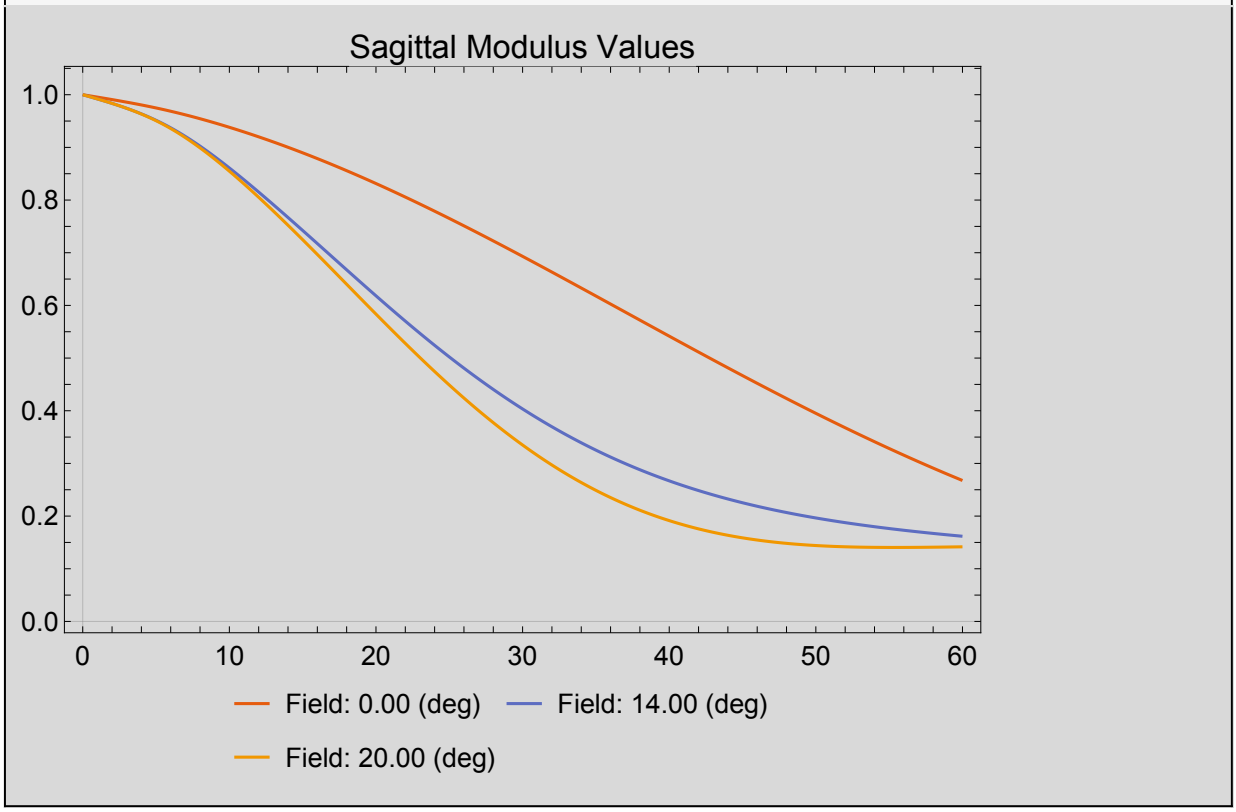
In[41]:=
```
p1 = ListLinePlot[(parseToPairsLists /@ getFftMtfResults)[[All, 2]],
  PlotLegends → Placed[fieldDescriptions, Bottom],
  PlotLabel → "Tangential Modulus Values", LabelStyle → 14]
```

Out[41]=

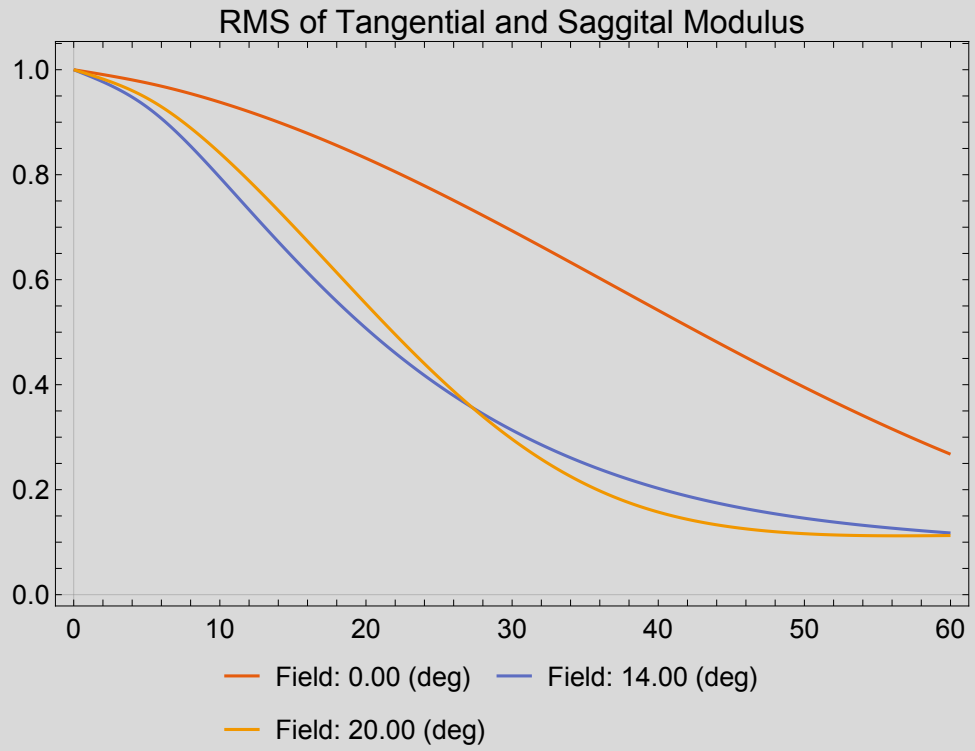


In[42]:=
```
p2 = ListLinePlot[(parseToPairsLists /@ getFftMtfResults)[[All, 3]],
  PlotLegends → Placed[fieldDescriptions, Bottom],
  PlotLabel → "Sagittal Modulus Values", LabelStyle → 14]
```

Out[42]=



In[43]:=
```
p3 = ListLinePlot[(parseToRMSlists /@ getFftMtfResults)[[All, 2]],
  PlotLegends → Placed[fieldDescriptions, Bottom],
  PlotLabel → "RMS of Tangential and Saggital Modulus", LabelStyle → 14]
```



Let's do some cool stuff

A function to determine the spatial frequency at which the RMS modulus has fallen to a specified value

This function accepts a list of pairs representing the MTF curve, generates an interpolating function, and then uses it to numerically solve for the first frequency for which the modulus has fallen to a prescribed value.

In[44]:=
```
contrastToFreq[contrast_, list_] := Module[{fMtf},
  fMtf = Interpolation[list];
  f /. FindRoot[fMtf[f] == contrast, {f, 0., maxFrequency}]
 ]
```

Here are the spatial frequencies at which the three fields have fallen to 50%

In[45]:=
```
contrastToFreq[0.5, #] & /@ (parseToRMSlists /@ getFftMtfResults)[[All, 2]]
```

Out[45]=
```
{42.7575, 20.3012, 21.8495}
```

50% contrast cutoff vs. back focal distance

Let' s look at how the 50 % contrast frequency is affected by defocus by building a table of values. We can also determine the time it takes to build the table.

In[46]:=
```
{time, cutOff50VsDefocus} = Timing[
   Table[
    setRowThickness[6, thick];
    runFftMtf;
    contrastToFreq[0.5, #] & /@ (parseToRMSlists /@ getFftMtfResults)[[All, 2]],
    {thick, 42.1, 42.3, 0.01}
   ]
  ];
```

It took a fraction of a second.

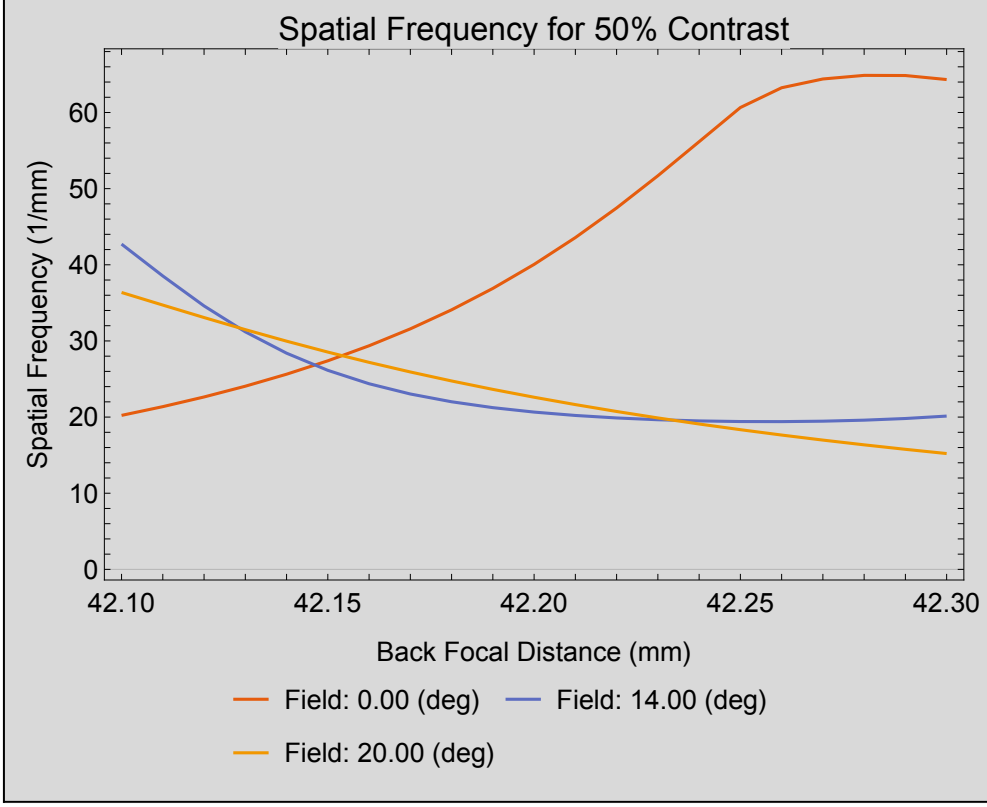
In[47]:=
```
time
```

Out[47]=
```
0.140401
```

Plot the results

In[48]:=
```
p4 = ListLinePlot[cutOff50VsDefocus // Transpose,
  DataRange → {42.1, 42.3}, PlotLegends → Placed[fieldDescriptions, Bottom],
  PlotLabel → "Spatial Frequency for 50% Contrast", FrameLabel →
   {"Back Focal Distance (mm)", "Spatial Frequency (1/mm)"}, LabelStyle → 14]
```

Out[48]=



Solve for the back focal distance for which the axial field has a 50% contrast at 30 lpmm

This function accepts a distance, sets the back focal distance to that value, runs an MTF, obtains the results, and determines the spatial frequency at which the modulus of the axial field is 50%. (Concise, isn't it?)

In[49]:=
```
frequency[distance_?NumberQ] :=
 (
  setRowThickness[6, distance];
  runFftMtf;
  (contrastToFreq[0.5, #] & /@ (parseToRMSlists /@ getFftMtfResults)[[All, 2]])[[1]]
 )
```

And here we use *Mathematica*'s numerical root finding to determine the back focal distance such that the modulus is 50% at 50 lpmm.

In[50]:=
```
(* and here's the distance *)
d /. FindRoot[frequency[d] == 30., {d, 42.2}]
```

Out[50]=
```
42.163
```

Export the graphics

In[51]:=
```
Export["p1.png", p1];
```

In[52]:=
```
Export["p2.png", p2];
```

In[53]:=
```
Export["p3.png", p3];
```

In[54]:=
```
Export["p4.png", p4];
```

And close the application

In[55]:=
```
theApplication@CloseApplication[]
```