How to Add Electromagnetic Field Sources to Simulations

David Ge

October 8, 2020

Contents

The software specification defines the structure and coding rules for the EM simulation system. See https://github.com/DavidGeUSA/TSS/blob/master/EM%20field%20Software%20Spec.pdf

This document gives step by step guidance in adding of new EM sources to the simulation system, not assuming much C/C++ knowledge from the readers. For those C++ guru readers, bear with me for my detailed explanations.

How to Add EM Sources

In the EM simulation system, a sample EM source is provided via project FieldSourceSamples. This project creates a DLL file (Dynamic Linking Library) named FieldSourceSamples.DLL. In that project, there is a class named `FieldSourceEz`. This class implements an EM source. To use this EM source in a simulation, add following lines in the task file for executing the simulation.

```
//DLL file for field source plugin module

SIM.FS_DLL=FieldSourceSamples.dll
```

```
//class name for field source plugin module, it is a Ricker source

SIM.FS_NAME=FieldSourceEz
```

See how the DLL file name and the EM source class name are specified.

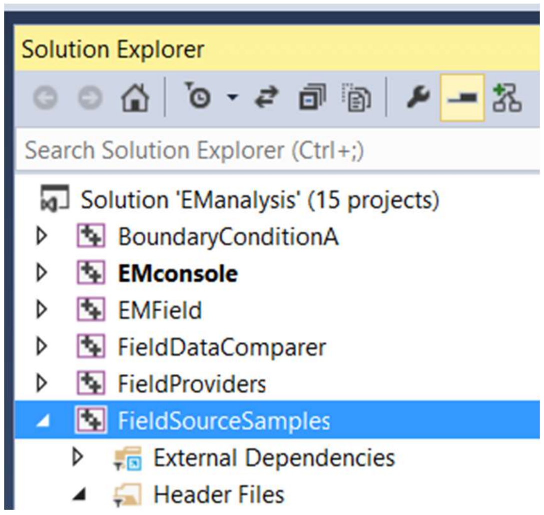
Now suppose we want to add our own EM source. We may add a new C++ class to this sample project FieldSourceSamples. Suppose we name the new C++ class SourceX. Then, we may use following lines in a task file to execute a simulation:

```
//DLL file for field source plugin module

SIM.FS_DLL=FieldSourceSamples.dll



//class name for field source plugin module, it is a Ricker source

SIM.FS_NAME=SourceX
```

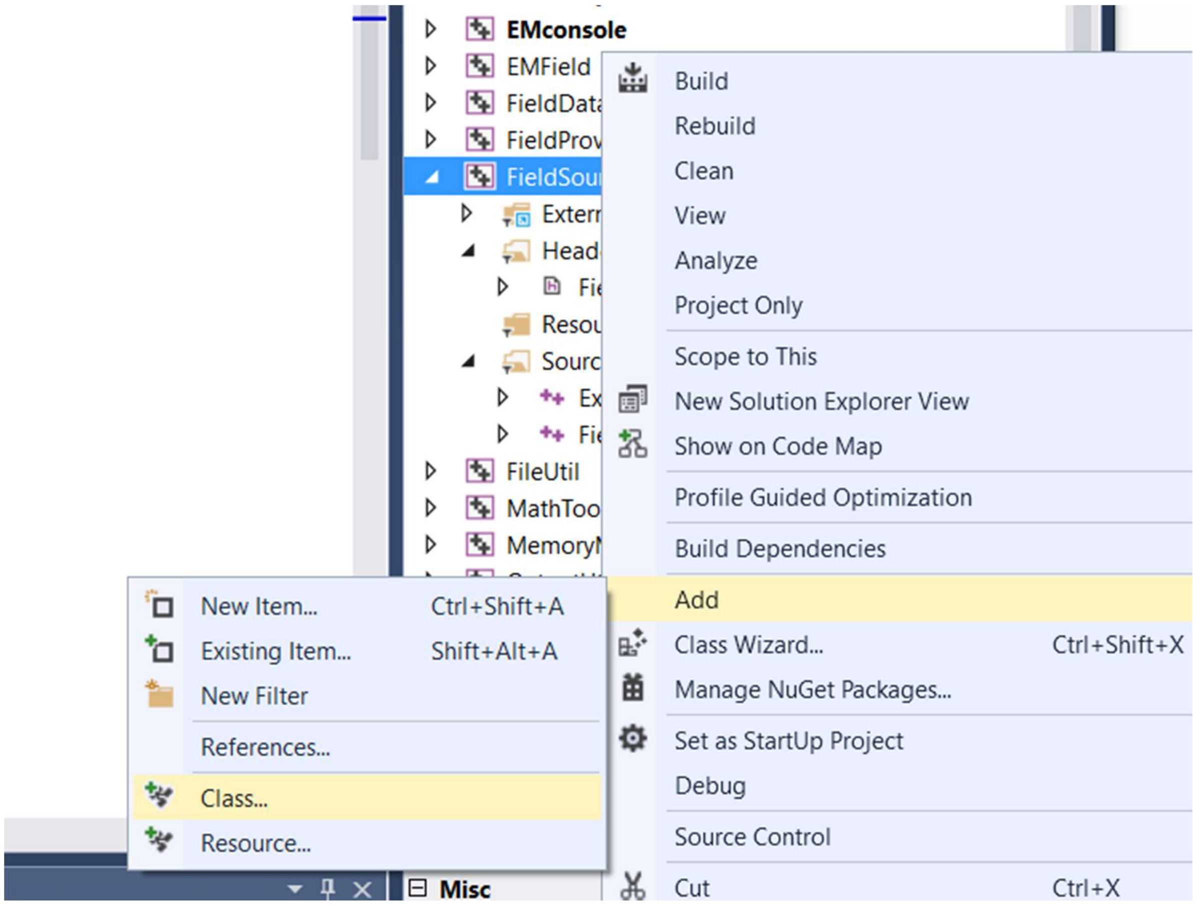
In the next section, we will show the steps of creating a new C++ class to implement a field source.

Step 1 – declare a new class

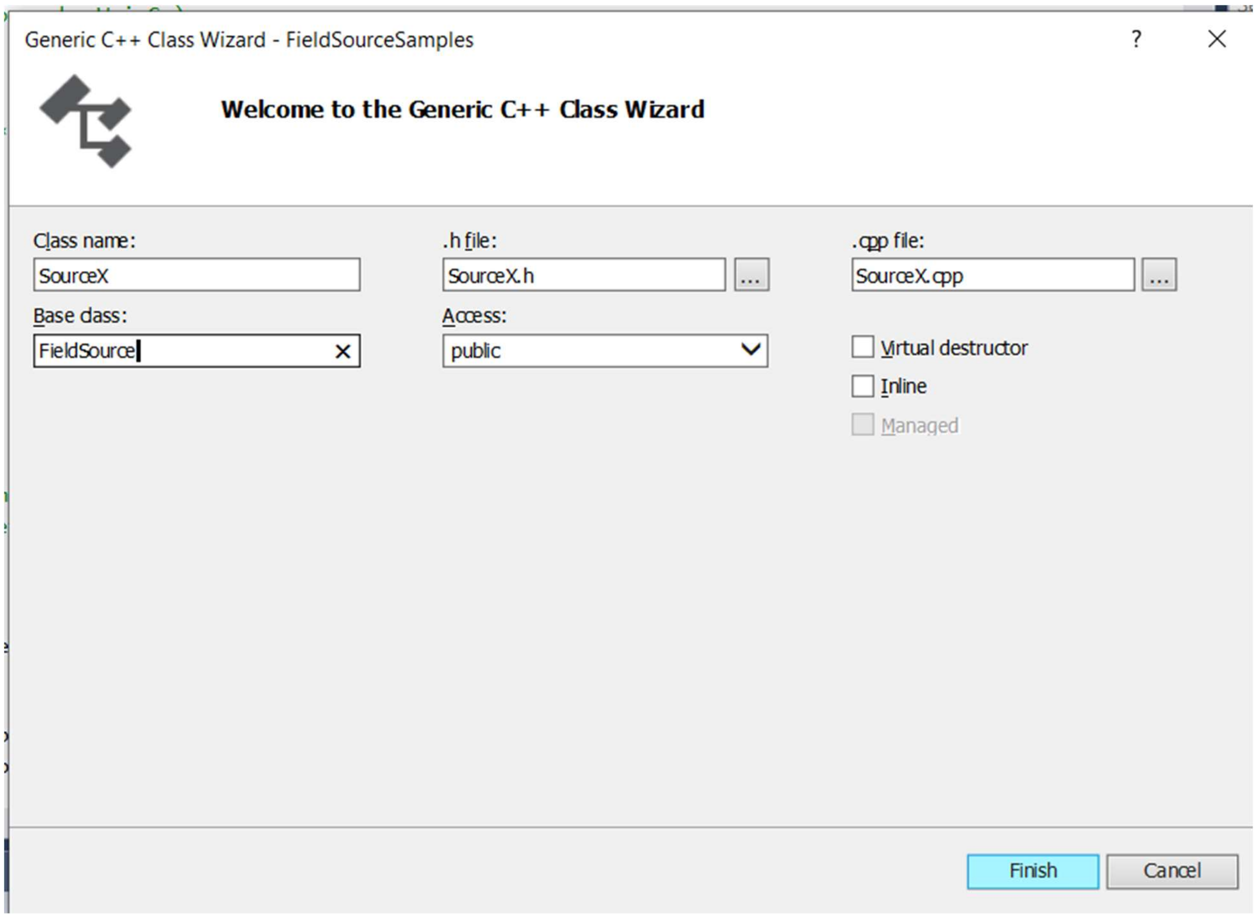
Add a new class to the project via the Solution Explorer of the Visual Studio:



Right-click the project name, choose “Add”, choose “Class”



Click “Add”, then fill the class name:


Generic C++ Class Wizard - FieldSourceSamples

Welcome to the Generic C++ Class Wizard

Class name: SourceX

.h file: SourceX.h

.cpp file: SourceX.cpp

Base class: FieldSource

Access: public

☐ Virtual destructor

☐ Inline

☐ Managed

Note that for the "Base class", use "FieldSource". FieldSource is defined in project EMField.

Click "Finish". Two files will be generated for you: SourceX.h and SourceX.cpp.

Now you can build the solution. The compiling should go through without a problem. A new field source is created, although it does nothing.

To make the new class do things, we need add contents to the class. We will use the sample class `FieldSourceEz` to show the process.

Step 2 – Add class data members

Class data members serve as local variables holding information specific to the class. You add class data members in the header file, SourceX.h. For example, in FieldSourceEz.h, we added a data member
```
double ppw;
```

Step 3 – Add class function members

Class function members define what the class can do. You add class function members in the header file, SourceX.h.

For an EM field source class, you must add three function members: `initialize`, `setRadius`, and `handleData`. So, at least, a field source class should contain following contents:

```
#pragma once
#include "..\EMField\FieldSource.h"
class SourceX :
	public FieldSource
{
public:
	SourceX();
	~SourceX();

	virtual int initialize(double Courant, int maximumRadius, TaskFile
*taskParameters);
	//
	virtual RadiusHandleType setRadius(int radius);
	virtual void handleData(int m, int n, int p);
};
```

Next, we need to implement those 3 functions, which is done in the file SourceX.cpp.

Step 4 – Implement `initialize and constructor`

This function is called before a simulation starts. This function is supposed to do initialization work. The minimum code is as below:

```
int SourceX::initialize(double Courant, int maximumRadius, TaskFile *taskParameters)
{
	int ret = FieldSource::initialize(Courant, maximumRadius, taskParameters);
	if (ret == ERR_OK)
	{
		//other initialization code here
	}
	return ret;
}
```

Another place of initialization happens at the constructor. The minimum implementation can be:

```
SourceX::SourceX()
{
	r = 0;
}
```

Where r is the radius.

Step 5 – Implement `setRadius`

Sample implementation could be:

```
RadiusHandleType SourceX::setRadius(int radius)
{
	return NeedProcess;
}
```

An EM field simulation is done by radius, radius =0, 1, 2,…. Before a simulation starts for a radius, this function is called and passing the value of the radius to it.

This function tells the simulation system how it wants to handle the radius. It can tell the simulation system following messages:

```
/*
	return value of setRadius
*/
typedef enum
{
	NeedProcess       = 0, //I want to process the radius
	DoNotProcess      = 1, //I do not want to process the radius
	Finish            = 2, //I do not want to process the radius and all other radius
	ProcessAndFinish = 3, //I want to process the radius, but I will not process all
other radius
}RadiusHandleType;
```

Let’s see how the sample field source, FieldSourceEz, handle this function:

```
RadiusHandleType FieldSourceEz::setRadius(int radius)
{
	if(radius > 0)
	{
		//end applying the source
		return Finish;
	}
	//radius == 0, apply the source
	return NeedProcess;
}
```

FieldSourceEz gives Ez value at space point (0,0,0) only. That is, it only needs to handle the case of radius=0. That is why when radius > 0, it returns Finish. Otherwise the radius is 0 and it returns NeedProcess.

Your source has its logic, you code the logic in this function.

Step 6 – Implement `handleData`

You implement your source in this function

```
void SourceX::handleData(int m, int n, int p)
{

}
```

The parameters m, n, and p specify the space location. You use these variables, together with a time value _time to determine how you want to apply the EM source. Let’s see how FieldSourceEz implements this function:

```
void FieldSourceEz::handleData(int m, int n, int p)
{
	double arg;
	arg = M_PI * ((Cdtds * _time - 0.0) / ppw - 1.0);
	arg = arg * arg;
	//m=n=p=0 is the first element of array _fields
	_fields[0].E.z += (1.0 - 2.0 * arg) * exp(-arg);
}
```

Note that because the way the function setRadius is coded (see the previous section), this function will only be called with m=n=p=0. It uses the time value _time to generate a Ez source value and added that value to Ez at the center of the simulation region.

Implement Adjustable Settings

Your EM source may include adjustable settings to make it more flexible. The values of such settings are specified in the task file. Let's see how FieldSourceEz does it.

From the previous section, we see that FieldSourceEz uses a parameter ppw for calculating the source value. The value of this parameter is read from the task file in the initialize function:

```
int FieldSourceEz::initialize(double Courant, int maximumRadius, TaskFile
*taskParameters)
{
	int ret = FieldSource::initialize(Courant, maximumRadius, taskParameters);
	if(ret == ERR_OK)
	{
		ppw = taskParameters->getDouble(TP_FS_PPW, false);
		ret = taskParameters->getErrorCode();
		if(ret == ERR_OK)
		{
			if(ppw <= 0.0)
			{
				ret = ERR_TASK_INVALID_VALUE;
				taskParameters->setNameOfInvalidValue(TP_FS_PPW);
			}
		}
	}
	return ret;
}
```

"false" in "ppw = taskParameters->getDouble(TP_FS_PPW, false);" indicates that this is a required parameter. If the task file does not contain it then an error occurs. The function should handle the error.
The function should also verify the validity of the value. In the above case, the value must be larger than 0.
You need to give such a setting a unique name to be used in the code and in the task file. For this example, the name is defined by

```
#define TP_FS_PPW "FS.PPW"
```

in FieldSourceEz.h

In a task file the value for the setting is specified using the name:

```
//the points per wavelength for Ricker source
FS.PPW=2
```

Step 7 – Add plugin code

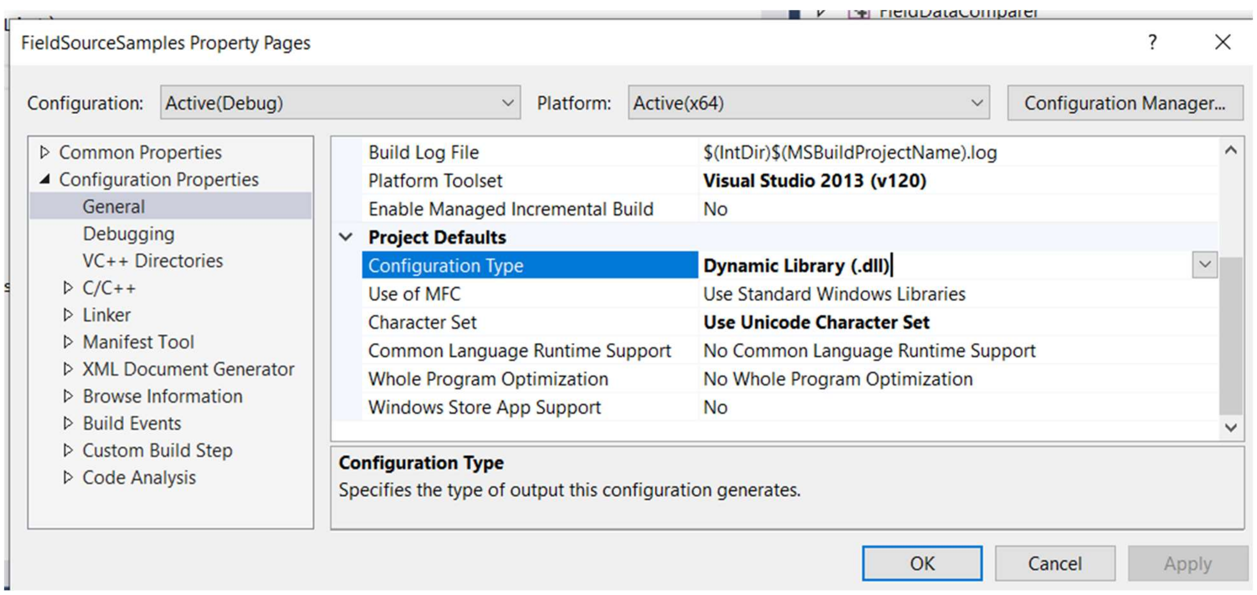
In order for a class defined in a DLL file to be usable by the simulation system, some plugin code need to be added into file ExportFieldSource.cpp. I'll highlight the related code:

```
#include "FieldSourceEz.h"
#include "SourceX.h"
```

```
#include <string.h>
#include <stdlib.h>

#ifdef __cplusplus
extern "C"
{
#endif /* __cplusplus */

	__declspec (dllexport) void* CreatePluginInstance(char *name, double *params);
	__declspec (dllexport) void RemovePluginInstances();

#ifdef __cplusplus
}
#endif /* __cplusplus */

FieldSourceEz **srcezList = NULL;
unsigned int srcezCount = 0;

SourceX **sourcexList = NULL;
unsigned int sourcexCount = 0;

__declspec (dllexport) void RemovePluginInstances()
{
	REMOVEALLPLUGINS(FieldSourceEz, srcezCount, srcezList);
	REMOVEALLPLUGINS(SourceX, sourcexCount, sourcexList);
}

__declspec (dllexport) void* CreatePluginInstance(char *name, double *params)
{
    if(strcmp(name,"FieldSourceEz") == 0)
	{
		FieldSource *p = NULL;
		CREATEPLUGININSTANCE(FieldSourceEz, srcezCount, srcezList);
		if (p != NULL)
		{
			p->setClassName(name);
		}
		return p;
	}
	else if (strcmp(name, "SourceX") == 0)
	{
		SourceX *p = NULL;
		CREATEPLUGININSTANCE(SourceX, sourcexCount, sourcexList);
		if (p != NULL)
		{
			p->setClassName(name);
		}
		return p;
	}
	return NULL;
}
```

How to a Create DLL File

To create a DLL file, open the project properties window, set the configuration type to Dynamic Library:


FieldSourceSamples Property Pages

Configuration: Active(Debug) Platform: Active(x64)

Build Log File	$(IntDir)$(MSBuildProjectName).log
Platform Toolset	**Visual Studio 2013 (v120)**
Enable Managed Incremental Build	No
Project Defaults	
Configuration Type	**Dynamic Library (.dll)**
Use of MFC	Use Standard Windows Libraries
Character Set	**Use Unicode Character Set**
Common Language Runtime Support	No Common Language Runtime Support
Whole Program Optimization	No Whole Program Optimization
Windows Store App Support	No

Configuration Type
Specifies the type of output this configuration generates.

Your project must include a source file similar to `ExportFieldSource.cpp`