MolSpin Extension Manual (preliminary version)

August 2019

Claus Nielsen
University of Southern Denmark

Table of Contents

1 Introduction

The MolSpin software is first and foremost an API that allows easy implementation of calculation methods – although a collection of standard task classes are provided with MolSpin, its real strength lies in its extensibility. This extension manual will help you create new task classes in order to utilize the true potential of MolSpin.

Note that this is a very early version of the extension manual, that only provides the very, very basics... It will be updated later!

2 Creating a new task class

There are two steps involved in creating a new task class: create the class itself, and register the class in MolSpin. Here we will implement a new task class called *MyTask*.

2.1 A simple task class

Create the header file for the class **RunSection/Tasks/Custom/MyTask.h** with the following content:

```
/////////////////////////////////////////////////////////////////////////
// MyTask header file - Custom task class example
/////////////////////////////////////////////////////////////////////////
#ifndef MOD_RunSection_MyTask
#define MOD_RunSection_MyTask

#include "BasicTask.h"

namespace RunSection
{
  class MyTask : public BasicTask
  {
    private:
      // Define private members here

    protected:
      bool RunLocal() override; // The Run method called every step
      bool Validate() override; // Validation method called before first step

    public:
      // Constructors / Destructors
      MyTask(const MSDParser::ObjectParser&, const RunSection&);  // Normal constructor
      ~MyTask();                          // Destructor
  };
}

#endif
```

And create the implementation file **RunSection/Tasks/Custom/MyTask.cpp**:

```
/////////////////////////////////////////////////////////////////////////
// MyTask implementation file - Custom task class example
/////////////////////////////////////////////////////////////////////////
#include <iostream>
#include "MyTask.h"

namespace RunSection
{
  // -----------------------------------------------------
  // // Constructor and Destructor
  // -----------------------------------------------------
  MyTask::MyTask(const MSDParser::ObjectParser& _parser, const RunSection& _runsection)
    : BasicTask(_parser,_runsection)
  {
  }

  MyTask::~MyTask()
  {
  }
  // -----------------------------------------------------
  // MyTask protected methods
  // -----------------------------------------------------
  // This method is run at every calculation step
  bool MyTask::RunLocal()
  {
    std::cout << "Hello World!" << std::endl;
    this->Log() << "Hello log stream!" << std::endl;
    this->Data() << "Hello data stream!" << std::endl;

    return true;
  }

  // Validation
  bool MyTask::Validate()
  {
    return true;
  }
  // -----------------------------------------------------
}
```

This is all that is needed to get started; you may use the *Validate* method for collection and verifying data from the input file (more on that later), and the RunLocal method is where your calculations are done. Note that you should normally use only the *Log()* or *Data()* streams for output, although of course the standard streams such as *std::cout* is also available.

2.2 Registration of the task class

You need to let MolSpin recognize the new task class such that it can be specified in the MolSpin input file. In order to do this, you need to add the following include-line to **RunSection/RunSection_CreateTask.cpp**:

```
#include "Custom/MyTask.h"
```

And add the following line to the end of the list of "else if" lines:

```
else if (_tasktype.compare("mytask") == 0) { task = std::make_shared<MyTask>(_obj, *this); }
```

Note that the string "mytask" is the *type* that you should specify for a Task object in your MolSpin input file to use your task class; it is recommended that you only use lowercase letters here, since all MolSpin input is converted to lowercase when read by the MSDParser module, unless it is specified as a string.

Finally you need to configure the **makefile** such that it may link MolSpin to your task class. First locate the following lines in **makefile**:

```
# RunSection custom tasks
PATH_RUNSECTION_CUSTOMTASKS = ./RunSection/Tasks/Custom
OBJS_RUNSECTION_CUSTOMTASKS =
DEP_RUNSECTION_CUSTOMTASKS =
```

Add **$(PATH_RUNSECTION_CUSTOMTASKS)/MyTask.o** to the list of object files:

```
OBJS_RUNSECTION_CUSTOMTASKS = $(PATH_RUNSECTION_CUSTOMTASKS)/MyTask.o
```

Note that objects files are separated by spaces if you add more custom task classes later.

At this point you may **compile MolSpin again** with your new task class. You may test your new task class by creating a MolSpin input file with the following Task object in the Run section:

```
Task run_my_task
{
  type = MyTask;
  logfile = "my_logfile.txt";
  datafile = "my_datafile.txt";
}
```

Note that the task type "MyTask" is converted to lowercase (as it is not a string) when read by MolSpin, and then matched against the string "mytask" in the else-if statement you added to **RunSection_CreateTask.cpp**.

2.3 Accessing spin system information

Access to any information about the spin system can be done through the method *SpinSystems()* which any task class inherits. The following code sample illustrates how input parameters may be accessed:

```
bool MyTask::RunLocal()
{
  // Get an STL vector of pointers to SpinSystem objects
  auto systems = this->SpinSystems();

  // Loop through all SpinSystems
  for (auto i = systems.cbegin(); i != systems.cend(); i++)
  {
    // Print the name of the SpinSystem
    this->Log() << "Found SpinSystem: " << (*i)->Name() << ", which contains:\n";
    this->Log() << " - " << (*i)->spins_size() << " spins\n";
    this->Log() << " - " << (*i)->interactions_size() << " interactions\n";
    this->Log() << " - " << (*i)->transitions_size() << " transitions\n";
    this->Log() << " - " << (*i)->states_size() << " state objects\n";
    this->Log() << " - " << (*i)->operators_size() << " operator objects\n";

    // Does the SpinSystem have a spin called my_electron?
    SpinAPI::spin_ptr spin = (*i)->spins_find("my_electron");
    if (spin == nullptr)
      this->Log() << "There is no spin called my_electron in this spin system!\n";
    else
      this->Log() << "There IS a spin called my_electron in this spin system!\n";

    if ((*i)->spins_size() > 0)
    {
      spin = *((*i)->spins_cbegin());
      this->Log() << "The first spin is " << spin->Name() << " with a spin of " << spin->S()
     << "/2 and multiplicity " << spin->Multiplicity() << ".\n";

      // Print spin operator in native coordinate frame and lab frame
      spin->Sx().print(this->Log(), "Sx spin operator:");
      spin->Tx().print(this->Log(), "Sx spin operator (lab frame):");

      // Convert to from sparse to dense matrices for nicer output
      arma::conv_to<arma::cx_mat>::from(spin->Sx()).print(this->Log(), "Sx spin operator:");
      arma::conv_to<arma::cx_mat>::from(spin->Tx()).print(this->Log(), "Sx spin operator (
    lab frame):");

      // Print isotropic and anisotropic parts of the tensor associated with the spin
      // The tensor is applied to the spin operators to get the lab frame spin operators
      this->Log() << "Spin tensor isotropic part: " << spin->GetTensor().Isotropic() << "\n";
      this->Log() << "Anisotropic part: " << spin->GetTensor().Anisotropic()(0) << " ";
      this->Log() << spin->GetTensor().Anisotropic()(1) << " " << spin->GetTensor().Anisotropic()(2) << "\n";

      // Get a custom property from a spin object
      double spin_mass;
      if (spin->Properties()->Get("mass", spin_mass))
        this->Log() << "A mass of " << spin_mass << " was specified!";
      else
        this->Log() << "No mass was specified!";

      this->Log() << std::endl; // Newline and flush stream
    }
  }

  return true;
}
```

This code sample requires including a few headers, so the include section at the top becomes:

```
#include "MyTask.h"
#include "Spin.h"
#include "SpinSystem.h"
#include "ObjectParser.h"
```

Note that you may need to include other SpinAPI header files if you use the corresponding SpinAPI objects. You can check the header files in the **SpinAPI** folder to see which public methods are available for the various types of SpinAPI objects.

2.4 Custom input for task classes

You may obtain custom parameters for your task class using the *Properties()->Get()* method, which requests the value of a specified keyword from the input file. The first parameter of the *Properties()->Get()* method is the keyword, and the second parameter is the variable that will hold the value specified in the input file. The method returns true if the keyword was found with a valid value. Note that the *Properties()* method returns a pointer to an ObjectParser object (MSDParser module), so you need to include **ObjectParser.h** in order to use this.

In order to demonstrate the use of the *Properties()->Get()* method, consider adding a *timestep* and a *steps* parameter to the task class. This could for example be parameters used for controlling a numerical integration algorithm. First add the parameters to the *private* section of the header (**.h**) file:

```
private:
  // Define private members here
  double timestep;
  unsigned int steps;
```

Then set the default values in the constructor in the implementation (**.cpp**) file:

```
MyTask::MyTask(const MSDParser::ObjectParser& _parser, const RunSection& _runsection)
  : BasicTask(_parser,_runsection), timestep(1.0), steps(1000)
{
}
```

Now you are ready to obtain the parameter values from the input file, and it is recommended that you do this in the *Validate* method:

```cpp
bool MyTask::Validate()
{
  this->Properties()->Get("timestep", this->timestep);
  this->Properties()->Get("steps", this->steps);
  return true;
}
```

That's it – now you may use these parameters in your *RunLocal* method since the *Validate* method is always called before the first time *RunLocal* is called. Note that the *Properties()->Get()* method is overloaded such that it can provide you with a variety of different data types depending on the type of the second parameter you pass to the method; you may even request a SpinAPI::Tensor object! See the header file **MSDParser/ObjectParser.h** for the specific overloads.

Although the approach above works, it is in general a good idea to validate the obtained input rather than just assuming that it is ok. Here is an improved version of the *Validate* method that checks whether the specified timestep is valid (finite and positive):

```cpp
bool MyTask::Validate()
{
  double input_timestep;
  if (this->Properties()->Get("timestep", input_timestep))
  {
    // A timestep was specified, check whether it is valid
    if (std::isfinite(input_timestep) && input_timestep > 0.0)
      this->timestep = input_timestep;
    else
      return false; // The timestep is invalid
  }

  this->Properties()->Get("steps", this->steps);

  return true;
}
```

3 Creating a new Action

To be added...

4 Creating a new Output type

To be added...

5 MolSpin programming reference

The programming reference is not written yet, but check the header file **RunSection/BasicTask.h**: the (non-virtual) protected methods provides you with access to all of the objects from the SpinAPI.

It is recommended to take a look at the header file **SpinAPI/SpinSpace.h** as well, as the SpinAPI::SpinSpace helper class provides many useful features.

Finally, check out the implemented tasks in **RunSection/Tasks/** which demonstrates how to use the SpinAPI.