Clarifications and necessary deal.II functions from this time.

Weixiong Zheng

Nov. 5, 2017

0. Why this document?
For a while, I hear concerns about using deal.II in BART development. They are mainly two types of concerns:
(1) deal.II is not C++, I know how to use C++, but not deal.II.
(2) Overwhelming usage of deal.II throughout the BART project.

Therefore, I think it would be nice to have some clarification about what deal.II is to C++ and how much deal.II you have to know if you want to get involved.

1. deal.II is not C++?
Well, logically, you are right. C++ is a language, yet, deal.II is a library using C++ and providing C++ interface. A similar "correct" phrase could be "SciPy is Python."

When using deal.II, you are writing C++ code. Similar examples are applicable to any third-party libraries to any languages. For instance, you are writing Python code using SciPy functions, or if you are a Jedi of C, you can think as you are wring C code using GSL. It is of course different from daily C++ functions as deal.II is not STL. Similar case is the Boost library. In old days while C++11 wasn't there (even today), Boost provides a lot of fancy functions that STL does not (shared_ptr is one of the examples).

2. Emm, is it so much of deal.II?
Admitting deal.II as regular third-party C++ libraries, we are faced with the second challenge: is it so much deal.II involved? I would say yes, and, not really.

(1) Yes, it is so much. In order to provide a chance to do FEM calculation, tons of codes were written in BART using deal.II functions.

(2) Not really. Logically, BART can be separated into two parts:
a. Methodology development part. This includes iterations schemes, weak formulation implementation etc. In short, this part is the mathematical/physical core of radiation calculations.
b. Implementation basis part. This include how we read-in parameters, how we set up MPI, how we initialize PETSc. In short, this includes anything that is not directly related to mathematical core of radiation calculations.

The reason we say "not really" is that, most deal.II-hard-coded part is in the Bullet b, which is not related to the radiation mathematical core. Yet, this part is already implemented and does not really require you touching it in development unless you aim to modify/optimize it. One potential example is if you in the future want to make BART read in meshes, you have to touch MeshGenerator, which is using deal.II mesh related functions.

In calculation related development, you surely have to know some functions from deal.II, which would be listed (or not if I didn't realize) in the following section. But that's it. What you surely understand is about C++.

3. So, what I need to know?
Assuming you are interested in writing a new weak formulation, there are several thing you need to know about deal.II relating you back to finite element courses.

NOTE: these are not exactly the same as what we do in BART, as we have some pre-assembly process. But the instruction here is really what you need to know (roughly) to get started with BART in terms of method development. If you want to do more on non-math part of BART, of course, **you need to do more deal.II learning, inevitably.**

(0) How to call basis functions and gradients of basis functions?
First of all, you should be glad that you don't need to implement basis functions as you did in finite element courses. deal.II does the dirty work for you and you just need to know two classes you can call basis functions from. In BART, we use shared_ptr to contain the objects of them:

```
// For cell volumetric basis
std_cxx11::shared_ptr<FEValues<dim> > fv;
// For cell side basis
std_cxx11::shared_ptr<FEFaceValues<dim> > fvf;
```

And both of them are already initialized in EquationBase<dim>, so basically, you don't have to worry about what to do with it.

In terms of calling basis function and gradients in a cell or a face, here is how:

```
// basis function in cell for i-th basis function at qi-th quadrature
point
double basis_value = fv->shape_value(i, qi);
// basis function in face for i-th basis function at qi-th quadrature
point
double face_basis_value = fvf->shape_value(i, qi);

// basis function gradient in cell for i-th basis function at qi-th
quadrature point
Tensor<1,dim> basis_gradient = fv->shape_grad(i,qi);
// basis function gradient in face for i-th basis function at qi-th
quadrature point
Tensor<1,dim> face_basis_gradient = fvf->shape_grad(i, qi);
```

(2) how do we do assembly of matrix in a cell/on a face
Note that the basis function is defined in reference cell with side length of 1. When doing calculations, you also need to know the quadrature weight in the reference cell and Jacobian transferring reference cell to real cell. Say, you want to know in a cell, the product of basis function i and j summing over all quadrature points, here's how (in the derived classes of EquationBase<dim>):

```
// assuming product is a zero value double type variable
for (unsigned int qi=0; qi<this->n_q; ++qi)
  product += (this->fv->shape_value(i, qi) *
              this->fv->shape_value(j, qi) *
              this->fv->JxW(qi));
```

where JxW is the Jacobian X Quadrature_weight_in_reference_cell function taking quadrature index as input parameter.

Summarizing everything, we do a little practice to assemble a mass matrix in a cell. The procedure is we test every basis function with every basis function at all quadrature points in a cell. Simple, right? Here's how:

Before we preceed, we need to know data structure for local matrix. It is a dealii::FullMatrix<double> object initialized somewhere as dealii::FullMatrix<double> (dofs_per_cell, dofs_per_cell). dofs_per_cell is an unsigned integer showing how many dofs we have per cell. It is retrieved in EquationBase<dim>. In 2D, it is 4, in 3D, it is 8 for bi/trilinear basis.

```
// FullMatrix<double> mass initialized and passed in as parameters
for (unsigned int qi=0; qi<this->n_q; ++qi):
  for (unsigned int i=0; i<this->dofs_per_cell; ++i)
    for (unsigned int j=0; j<this->dofs_per_cell; ++j)
      mass(i,j) += (this->fv->shape_value(i, qi) *
                    this->fv->shape_value(j, qi) *
                    this->fv->JxW(qi));
```

Another example is to calculate stiffness matrix which is quite similar:

```
// FullMatrix<double> stiffness initialized and passed in as
parameters
for (unsigned int qi=0; qi<this->n_q; ++qi):
  for (unsigned int i=0; i<this->dofs_per_cell; ++i)
    for (unsigned int j=0; j<this->dofs_per_cell; ++j)
      stiffness(i,j) += (this->fv->shape_grad(i, qi) *
                         this->fv->shape_grad(j, qi) *
                         this->fv->JxW(qi));
```

usage of fvf on cell face is the same as fv in cell.

(3) how do we get solution value:

In terms calculation of scattering source or fission source, you want to know local values of the solution. More specifically, you want to know the solution on quadrature points. Sometimes, you might also want to know the gradients. Here's how:

```
// retrieving local solution
std::vector<double> local_soln (this->n_q);
// local solutions assuming soln_process is dealii::Vector<double>
object representing solution living on current processor
this->fv->get_function_values (soln_process, local_soln);

// retrieving local gradients
std::vector<Tensor<1,dim> > local_grad (this->n_q);
this->fv->get_function_gradients (soln_process, local_grad);
```

Say you want to know the scattering source in a local cell assuming $\sigma_s = 1$:

```
// retrieving local solution
std::vector<double> local_soln (this->n_q);
// local solutions assuming soln_process is dealii::Vector<double>
object representing solution living on current processor
this->fv->get_function_values (soln_process, local_soln);

// assemble local source. In BART, this will be passed as a parameter
to assembly function
Vector<double> local_source (this->dofs_per_cell);
for (unsigned int qi=0; qi<this->n_q; ++qi)
  for (unsigned int i=0; i<this->dofs_per_cell; ++i)
    local_source(i) += (this->fv->shape_value(i, qi) *
                        local_soln[qi] *
                        this->fv->JxW(qi));
```

4. Other things:

(1) Please familiarize yourself with some simple STL data structure.

We don't use tons of STL functions/data structures, but still, you need to know some that we use in BART. Here's the list that I can recall:

```
// shared_ptr either from C++11 or boost depends on your deal.II
build and compiler
std_cxx11::shared_ptr<T>
// casting using shared_ptr
std_cxx11::shared_ptr<Base> derived_ptr =
        std_cxx11::shared_ptr<Base> (new Derived());

// vector and peeking front and back
std::vector<T>
```

```
std::vector<T>::front()
std::vector<T>::back()

// Hash table
std::unordered_map<T, T>

// Tree (maybe heap, not sure here) based ordered map
std::map<T, T>

// ordered_set
std::set<T>
```

Note: please be careful with “auto”. It is definitely convenient to use but it’s sometimes unclear when we need to know the data type. Also, it might not work well with non-STL data type, such as “DoFHandler<dim>::active_cell_iterator”.

(2) Please don’t change to std::make_shared for now.
At the time when I started the code, I didn’t know make_shared. If you want to change BART to use make_shared for casting, I will be okay only if you switch all of them to keep consistency.

(3) Please feel free to ask on Slack or come by my office.