Enzo 3.0: An Introduction to Field Objects

James Larrue-Baulch

February 17, 2015

The Enzo code is open-source implementation of AMR for astrophysics written in a mix of C/C++ and FORTRAN. Bryan et al. (2013) have given a high-level description of the software, while more details and the source code can be accessed through the Enzo Project website[1].

Prior to Enzo 3.0, adding new fields was a cumbersome process, particularly if the field values were not cell centered. At a minimum, numerous constants needed to be added, and often new code needed to be threaded throughout the application.

To faciliate adding new fields to Enzo, it was decided to introduce field objects in Enzo 3.0. Now, a field's values and properties are encapulated in a FieldDescriptor object, while a grid's fields are collected in the grid's FieldRegistry object. The Enzo version with field objects provides all previous functionality and tests indicate that its performance and memory are comparable with Enzo before field objects.

Field objects make manipulating fields much easier. New fields can now be easily created, regardless of the field's value-centering, dimensions, interpolation method, or other field properties.

Contents

[1] The website of the Enzo Project is at `http://www.enzo-project.org/`.

1 Current State of Field Objects

The field objects have been implemented throughout the code. The recent changes to the enzo-3.0 repository were merged into the field object code on January 9, 2015. The two forks should be sychronized up to the enzo-3.0 commit d4d9eb4.

The `grid::BaryonField[]` variable no longer exists, nor do the electromagnetic field variables. Several other variables that still exist in the `grid` class could be moved to field objects, but have not yet been done. Before converting any more fields to the field objects, we would like to get community feedback on the work thus far.

The Enzo *full* test suite gives expected results, both with and without MPI. The tests in the *push* test suite were used to test the speed performance of field objects. After some optimization using *gprof*, our test durations with field objects are now comparable to those before field objects, with the exception of some radiation transport tests. (See Next Steps for details.) This is the case with both with and without MPI.

In addition to testing with the test suite, we have also examined the memory usage. Our initial tests, using a custom-created MHD scenario with AMR, showed an average of 7% less memory used during the simulation.

2 How to Use Field Objects

Prior to field objects, the field values were contained in various members of the `grid` class, such as `BaryonField[]` and `MagneticField[]`. The value centering and dimensions of the fields were defined outside of the application and separate code was needed to handle each of the different field-containing members. In the class diagram of figure 1 (page 4), the `grid` class from before field objects has 37 public members, 64 protected members, and 592 public methods. In comparison, the `grid` class with field objects has 20 public members, 56 protected members, and 535 public methods. The removed members and methods have either been moved to be part of the field objects, or else rendered unnecessary by field objects.

The main classes for field objects are the `FieldRegistry` and the `FieldDescriptor`. The `FieldRegistry` behaves like the `std::map<std::string, FieldDescriptor*>` class, with a few exceptions and a few additional features. First, the method `FieldRegistry::operator[](std::string)` with a non-existant key will not create a new entry, as it would with `std::map`. Also, elements are added to the field registry with the `FieldRegistry::add(FieldDescriptor*)` method. The key for the new element will be the name of the field descriptor. By requiring the `FieldRegistry ::add(FieldDescriptor*)` method to add elements, it is guaranteed that any elements in the field registry will be a field descriptor (a pointer to a field descriptor, actually). Those elements can be iterated over with help from the `FieldRegistry::begin()` and `FieldRegistry::end()` methods.

The `FieldDescriptor` class contains all the information concerning a field, including the current field values, the old field values, the value centering, and the field dimensions. With all fields stored as `FieldDescriptor` objects, the code can iterate over the field descriptors in a field registry and process them in a generic manner.

A key advantage of field objects is that no changes are required to accommodate different field geometries, as was required before field objects. The `FieldDescriptor` class can represent new fields, whatever their geometry, and by simply adding the new field descriptor to the field registry, the Enzo infrastructure will take care of the rest.

The `grid` class before field objects.

The `grid` class with field objects.

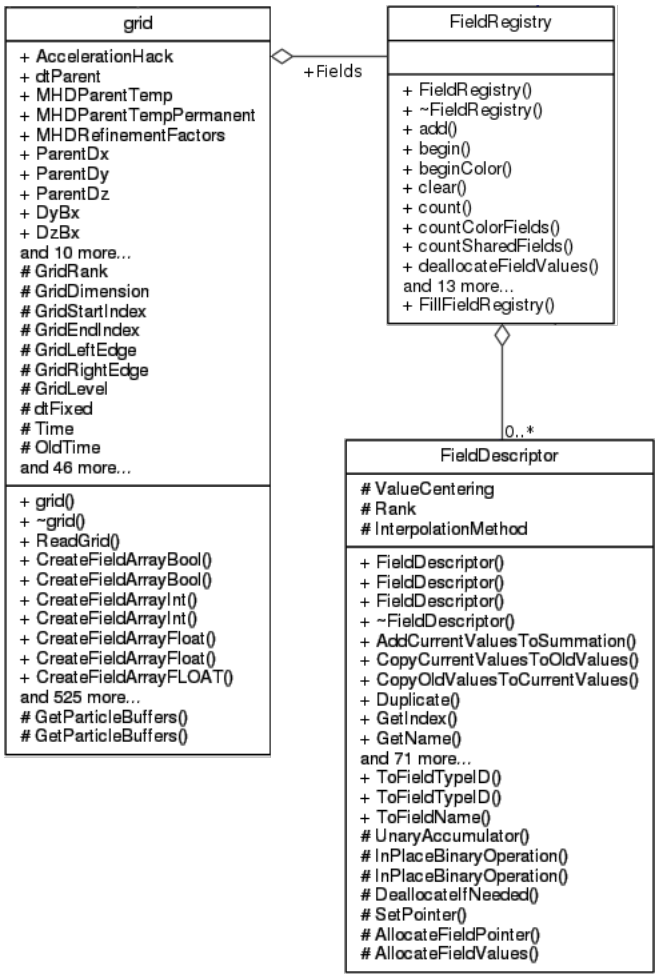

Figure 1: Class diagrams before and with field objects, excluding classes not explicitly related to field objects.

2.1 Getting and Setting Field Values

Before field objects, to access field values, we might code the following in a method of the grid class:

```
// Get the field indices.
int DensNum, GENum, Vel1Num, Vel2Num, Vel3Num, TENum;
IdentifyPhysicalQuantities(
  DensNum, GENum, Vel1Num, Vel2Num, Vel3Num, TENum
);

// Set total energy from velocities.
BaryonField[TENum][index] = 0.5 * (
  pow(BaryonField[Vel1Num][index], 2)
  + pow(BaryonField[Vel2Num][index], 2)
  + pow(BaryonField[Vel3Num][index], 2)
);

// Create a variable from the old velocity values.
float OldTotalEnergy = 0.5 * (
  pow(OldBaryonField[Vel1Num][index], 2)
  + pow(OldBaryonField[Vel2Num][index], 2)
  + pow(OldBaryonField[Vel3Num][index], 2)
);
```

The equivalent code with field objects would be:

```
FieldDescriptor *pVelocity1Field = Fields[FIELD_NAME_VELOCITY_1];
FieldDescriptor *pVelocity2Field = Fields[FIELD_NAME_VELOCITY_2];
FieldDescriptor *pVelocity3Field = Fields[FIELD_NAME_VELOCITY_3];
FieldDescriptor *pTotalEnergyField = Fields[FIELD_NAME_TOTAL_ENERGY];

// Set total energy from velocities.
// Option #1
pTotalEnergyField->SetValue(
  index,
  0.5 * (
    pow(pVelocity1Field->GetValue(index), 2)
    + pow(pVelocity2Field->GetValue(index), 2)
    + pow(pVelocity3Field->GetValue(index), 2)
  )
);

// Set total energy from velocities.
// Option #2 - Same result as option #1, different syntax
(*pTotalEnergyField)[index] = 0.5
  * (
    pow((*pVelocity1Field)[index], 2)
    + pow((*pVelocity2Field)[index], 2)
    + pow((*pVelocity3Field)[index], 2)
  );

// Create a variable from the old velocity values.
float OldTotalEnergy = 0.5 * (
    pow(pVelocity1Field->GetOldValue(index), 2)
    + pow(pVelocity2Field->GetOldValue(index), 2)
    + pow(pVelocity3Field->GetOldValue(index), 2)
);
```

In the above code with field objects (and in all other examples with field objects), we could use `Fields["Velocity1"]` rather than `Fields[FIELD_NAME_VELOCITY_1]`. An advantage of the latter is that the compiler will give an error if you make a typo. In other regards, the two syntaxes are synonymous.

2.2 Field Arithmetic

Field objects simplify field arithmetic. The `FieldDescriptor` class has methods for addition, subtraction, multiplication, and division. All four operators can accept a `float` value or a pointer to a `FieldDescriptor` object.

Before field objects, we might code the following:

```
for (int field = 0; field < NumberOfBaryonFields; field++) {
  if (MakeFieldConservative(FieldType[field])) {
    // Multiply by density
    for (int k = 0; k < GridDimension[2]; k++) {
      for (int j = 0; j < GridDimension[1]; j++) {
        for (int i = 0; i < GridDimension[0]; i++) {
          int index =  i
            + GridDimension[0] * (
              j + GridDimension[1] * k
            );
          BaryonField[field][index] *= DensityValues[index];
        }
      }
    }
  }
}
```

With field objects, we could code:

```
for (
  FieldRegistry::iterator Iterator = Fields.begin();
  Iterator != Fields.end();
  ++Iterator
) {
  FieldDescriptor *pField = Iterator->second;
  if (pField->GetInterpolationMethod() == MultiplyByDensity) {
    pField->Multipy(pDensityField);
  }
}
```

With field objects, we use a `FieldRegistry::iterator` to iterate over the fields and we do not need to iterate over the field elements, since the `FieldDescriptor::Multiply` method handles that.

2.3 Adding a New Field

Previously, to add a new field with cell-centered values and a standard geometry, we needed to:[2]

1. Add a field type ID to `typedefs.h`;
2. Add the field to `Grid_InitializeUniformGrid.C`; and
3. Add the field to the `DataLabel` and `DataUnit` arrays.

Then, you would access your new field with:

```
int myFieldIndex = FindField(NewFieldTypeID, FieldType, NumberOfBaryonFields);
float *pMyFieldValues = BaryonField[myFieldIndex];
```

To add a new field that was not cell-centered, or not a standard geometry, we needed to:

1. Create a new `float*` variable in `Grid.h`;
2. Add the field to `Grid_InitializeUniformGrid.C` (or a similar method);
3. Add code to pass your new field to other processors via MPI;
4. Add code to read and write your new field; and
5. Add more code in other places to handle your field.

Now, to add a new field, we must:

1. Call `grid::CreateField("My New Field")` in `Grid_InitializeUniformGrid.C` (or in a similar method).

Now the grid has a field called "My New Field" with cell centered values and the default number of ghost zones. The field will be accessible with:

```
grid->Fields["My New Field"]
```

There are various flavors of the `grid::CreateField(...)` method that control the value-centering, interpolation method, and other properties of the resulting field.

2.3.1 Optional: Adding a Field Template

If the new field is going to be used in many different problems, we could add it as a field template in:

```
FieldRegistry::FillFieldRegistry(...)
```

after which `grid::CreateFieldFromTemplate("My New Field")` can be used to create the field with the correct field properties.

The difference between `grid::CreateField(...)` and `grid::CreateFieldFromTemplate(...)` is that any non-default field properties must be specified in the parameters of the former, while the field properties of the template field are inherited in the latter method.

2.3.2 Optional: Adding a Field Name Constant

If we want to be neat-and-tidy, we could also add a string constant for the new field name in `FieldNameCosntants`:

[2] See `https://enzo.readthedocs.org/en/enzo-2.4/developer_guide/HowToAddNewBaryonField.html`.

```
// In FieldNameConstants.h
class FieldNameConstants {
  public:
    ...
    static const std::string FIELD_NAME_MY_NEW_FIELD;
    ...
}

// In FieldNameConstants.C
const std::string FieldNameConstants::FIELD_NAME_MY_NEW_FIELD = "My New Field";
```

Now, we can access the field with:

```
grid->Fields[FIELD_NAME_MY_NEW_FIELD]
```

2.4 Creating a New Problem Type

The steps for creating a new problem type are described in the Enzo documentation for version 2.4[3]. Many of the steps are the same with field objects, so we will only discuss steps that are different, in particular setting up the Data Labels and Data Units in the `MyProblemInitialize` method and initializing the fields in the `grid::MyProblemInitializeGrid` method.

Prior to field objects, it was **ABSOLUTELY ESSENTIAL** to set up the Data Label array. Setting up the Data Units array was optional. With field objects, the data labels and data units are stored in the `FieldDescriptor` objects, so there is no need to set either.

In the `grid::MyProblemInitializeGrid` method, we used to set the `grid::FieldType` array, then initialize the `grid ::BaryonField` values:

```
  //////////
  // Create the fields by setting the FieldType array.
  //////////

  NumberOfBaryonFields = 0;

  FieldType[NumberOfBaryonFields++] = Density;
  if (DualEnergyFormalism) {
    FieldType[NumberOfBaryonFields++] = InternalEnergy;
  }
  if( EquationOfState == 0 ) {
    FieldType[NumberOfBaryonFields++] = TotalEnergy;
  }
  FieldType[NumberOfBaryonFields++] = Velocity1;
  FieldType[NumberOfBaryonFields++] = Velocity2;
  FieldType[NumberOfBaryonFields++] = Velocity3;
  if (HydroMethod == MHD_RK) {
    FieldType[NumberOfBaryonFields++] = Bfield1;
    FieldType[NumberOfBaryonFields++] = Bfield2;
    FieldType[NumberOfBaryonFields++] = Bfield3;
    FieldType[NumberOfBaryonFields++] = PhiField;
  }

  ...

  //////////
  // Initialize the BaryonField values.
  //////////

  int DensNum, GENum, Vel1Num, Vel2Num, Vel3Num, TENum;
  IdentifyPhysicalQuantities(
    DensNum, GENum, Vel1Num, Vel2Num, Vel3Num, TENum
  );

  // All fields have cell-centered values and have the default number of ghost
  // zones (i.e. the same number as the grid).
  for (int k = 0; k < GridDimension[2]; k++) {
    for (int j = 0; j < GridDimension[1]; j++) {
      for (int i = 0; i < GridDimension[0]; i++) {
        int index =  i
          + GridDimension[0] * (
            j + GridDimension[1] * k
          );
        float x = Scale[0] * (i - GridStartIndex[0] + 0.5);
        float y = Scale[1] * (j - GridStartIndex[1] + 0.5);
        float v_x = v_o * sin(2 * pi * y);
        float v_y = v_o * sin(2 * pi * x);
        float v_z = 0.0;

        BaryonField[DensNum][index] = InitialDensity;
        BaryonField[Vel1Num][index] = v_x;
```

[3]See `https://enzo.readthedocs.org/en/enzo-2.4/developer_guide/NewTestProblem1.html`.

```
        BaryonField[Vel2Num][index] = v_y;
        BaryonField[Vel3Num][index] = 0.0;
        if (DualEnergyFormalism) {
          BaryonField[GENum][index] = InitialGasEnergy;
        }
        if (EquationOfState == 0) {
          BaryonField[TENum][index] = (
            InitialGasEnergy
            + 0.5 * (
              pow(CenteredB[0][index], 2)
              + pow(CenteredB[1][index], 2)
              + pow(CenteredB[2][index], 2)
            )
          ) / InitialDensity
          + 0.5 * (pow(v_x, 2) + pow(v_y, 2) + pow(v_z, 2));
        } else {
          BaryonField[TENum][index] = 0.0;
        }
      }
    }
```

With field objects, we no longer have a `FieldType` array. Most fields of the required fields will be created with the `grid::CreateStandardFields()` method. (See the API specification for a listing of which fields will be created under which scenarios.) In the unsually case where we want to create fields that have not been created as standard fields, we can use the `grid::CreateFieldFromTemplate` method or the `grid:CreateField` method. The above code rewritten with field objects might look like:

```
  //////////
  // Create the fields with the grid::CreateStandardFields() method.
  //////////

  CreateStandardFields();

  ...

  //////////
  // Initialize the field values.
  //////////

  FieldDescriptor *pDensityField = Fields[FIELD_NAME_DENSITY];
  pDensityField->CopyFrom(InitialDensity);

  // If we have DualEnergyFormalism, then the internal energy field will exist.
  if (Fields.count(FIELD_NAME_INTERNAL_ENERGY) > 0) {
    Fields[FIELD_NAME_INTERNAL_ENERGY]->CopyFrom(InitialGasEnergy);
  }

  FieldDescriptor *pVelocity1Field = Fields[FIELD_NAME_VELOCITY_1];
  FieldDescriptor *pVelocity2Field = Fields[FIELD_NAME_VELOCITY_2];
  FieldDescriptor *pVelocity3Field = Fields[FIELD_NAME_VELOCITY_3];

  FieldDescriptor *pTotalEnergyField = Fields[FIELD_NAME_TOTAL_ENERGY];

  FieldDescriptor *pMagneticField1 = Fields[FIELD_NAME_MAGNETIC_FIELD_1];
  FieldDescriptor *pMagneticField2 = Fields[FIELD_NAME_MAGNETIC_FIELD_2];
  FieldDescriptor *pMagneticField3 = Fields[FIELD_NAME_MAGNETIC_FIELD_3];

  pVelocity3Field->CopyFrom(0.0);

  // Here, we are assuming that the fields are all the same size, which will
  // be the case if they all use the same value-centering (e.g. cell centered)
  // and if they all use the same number of ghost zones.  We get the field
  // dimensions from the velocity 1 field, but we could have used any field
  // with the same dimensions.
  for (int k = 0; k < pVelocity1Field->GetFieldDimension(2); k++) {
    for (int j = 0; j < pVelocity1Field->GetFieldDimension(1); j++) {
```

```
      for (int i = 0; i < pVelocity1Field->GetFieldDimension(0); i++) {
        int index = pVelocity1Field->GetIndex(i, j, k);
        float x = Scale[0]
          * (i - pVelocity1Field->GetNumberOfGhostZones() + 0.5);
        float y = Scale[1]
          * (j - pVelocity1Field->GetNumberOfGhostZones() + 0.5);
        float v_x = v_o * sin(2 * pi * y);
        float v_y = v_o * sin(2 * pi * x);
        float v_z = 0.0;

        pVelocity1Field->SetValue(index, v_x);
        pVelocity2Field->SetValue(index, v_y);
        if (EquationOfState == 0) {
          pTotalEnergyField->SetValue(
            index,
            (
              InitialGasEnergy
              + 0.5 * (
                pow(pMagneticField1->GetValue(index), 2)
                + pow(pMagneticField2->GetValue(index), 2)
                + pow(pMagneticField3->GetValue(index), 2)
              )
            ) / InitialDensity
            + 0.5 * (pow(v_x, 2) + pow(v_y, 2) + pow(v_z, 2));
        } else {
          pTotalEnergyField->SetValue(index, 0.0);
        }
      }
    }
  }
```

For this example, we have initialized some fields with the `FieldDescriptor::CopyFrom(float)` method, while other fields have been initialized inside for-loops with the `FieldDescriptor::SetValue(int, float)` method. The choice of one over another depends on the situation, though we can always use the for-loop method.

2.5 Value-Centering and Non-Standard Geometry

Before field objects, all the fields in the `BaryonField` array contained cell-centered values and they all had the default number of ghost zones. Fields that did not contain cell-centered values needed to be created outside of the `BaryonField` data structure (e.g. `ElectricField` and `MagneticField`) and additional code was required to handle those fields. The same situation occured with fields that did not contain the default number of ghost zones (e.g. `divB` and `gradPhi[]`).

With field objects, field values can be cell-centered, face-centered, edge-centered, or vertex-centered. The value-centering is stored as part of the `FieldDescriptor` and the field dimensions are increased as necessary (e.g. +1 cell in one dimesion for face-centered values). The Enzo infrastructure has been modified to account for each field's value-centering, so the same code will work no matter what value-centering the fields have.

Similarly, fields that use a non-default number of ghost zones can be used without rewriting any of the Enzo infrastructure code. For example, the field for $\nabla \cdot B$ has zero ghost zones, while the normal default is 3 ghost zones.

The following table compares various field dimensions before and after field objects. Before field objects, fields inherited their dimensions directly from the grid. With field objects, while the field dimensions are based on the grid dimensions, they are not necessarily the same.

Before Field Objects	With Field Objects
`grid::GridDimension[]`	`FieldDescriptor::GetFieldDimensions()[]`
`// Varies by case.`	`FieldDescriptor::GetFieldExtensions()[]`
`grid::GridStartIndex[]`	`FieldDescriptor::GetNumberOfGhostZones()`
`grid::GridEndIndex[]`	`FieldDescriptor::GetFieldDimension(int)` `  - FieldDescriptor::GetNumberOfGhostZones() - 1`

3 Next Steps

1. Community review of the code.
   - The new code needs to be reviewed by the community so that we can both find further improvements and identify any conceptual bugs (e.g. if some feature was misunderstood when the field objects were implemented).
2. Community test of the code.
   - The code has been tested, but by a very small number of people. It needs a more thorough testing that can only be accomplished by the larger community.
3. Add developer documentation in Sphinx format.
4. Speed up the radiation transport tests.
   - The grid::WalkPhotonPackage method is called many times and its calls to FieldDescriptor methods are resulting in a performance bottleneck. A solution is not expected to be difficult.
5. Write more unit tests.
   - Some unit tests were initially written for field objects, but the feature set has grown significantly, so more unit tests are required.
6. More stress-testing.
   - We want to do some cosmology tests with MHD. Other tests by the community would be a great benefit.
7. Incorporate community feedback.
   - The community review will likely result in a list of desired changes. These need to be incorporated and retested, or else prioritized for future development.
8. Resolve any blocker issues found during testing.
   - Any issue that significantly hinders the normal use of the software must be resolved before the field objects are merged with the main branch of enzo-3.0.
9. Merge field objects into enzo-3.0.
   - Because so much of the code has been impacted by the migration to field objects, merging may be more complex than usual. The main fork enzo-3.0 was last merged into the development fork in early January 2015.
10. Prioritize future work items.
    - The community review process will likely produce some good ideas for future development. These ideas should be prioritized so that development can proceed in priority order.

4 References

Greg L. Bryan, Michael L. Norman, Brian W. O'Shea, Tom Abel, John H. Wise, Matthew J. Turk, Daniel R. Reynolds, David C. collins, Peng Wang, Samual W. Skillman, Britton Smith, Robert P. Harkness, James Bordner, Ji hoon Kim, Michael Kuhlen, Hao Xu, Nathan Goldbaum, Cameron Hummels, Alexei G. Kritsuk, Elizabeth Tasker, Stephen Skory, Christine M. Simpson, Oliver Hahn, Jeffrey S. Oishi, Geoffrey C. So, Fen Zhao, Renyue Cen, and Yuan Li. Enzo: An adaptive mesh refinement code for astrophysics. 2013.