An object-oriented environment for Newtonian particle mechanics

P. Mitic, P.G. Thomas
Faculty of Mathematics and Computing, The Open University, Walton Hall, Milton Keynes MK7 6AA, UK

Abstract

Newtonian particle mechanics is embedded within an object-oriented framework, such that it is possible to model physical situations using a small toolkit of logical objects. Modelling is done by constructing objects with appropriate methods and attributes and linking them. This results in the creation of more objects, including, eventually, an object representing an equation of motion. From this, an actual equation of motion can be extracted. This system produces correct solutions in a wide variety of situations, some of which are described.

1 Introduction

Existing undergraduate courses on mechanics provide many examples of how to solve problems which involve particles, forces, strings, springs and other similar elements. Such elements are mentioned implicitly, and methods are stressed in the sense that examples of solved problems are provided. There are certain problems with this approach. Students can have difficulty in determining a starting point for finding a solution, in isolating the necessary steps which are needed to obtain a solution, and in applying the correct techniques and processes at the appropriate time.

Generic computer algebra systems have been used to solve problems in particular contexts by providing a specific environment for problem solving. Mathematica is particularly flexible and adaptable in this respect. An environment for solving problems in Newtonian mechanics has been implemented by Dubisch[1]. This approach relies on a toolkit of templates, which works well if the a template is available. If not, a template must be created. This can lead to a large number of specific cases.

We consider that the implicit introduction of physical objects in texts on mechanics is quite natural, and is more important than has been thought. The axioms of Newtonian mechanics provide a scientific model for mechanics, and has no parallel elsewhere. The axioms themselves concern the way in which matter

behaves in space. We discuss the nature of this matter by approaching the problem in an object-oriented (O-O) way. A similar approach has been used by Viklund and Fritzson[2] in the context of finite element computations. They supplement Mathematica by *ObjectMath*, with which objects can be defined and manipulated.

In this paper, physical elements used in simple particle mechanics are first identified. They are then classified in terms of an object hierarchy. Each object is assigned relevant attributes and methods. This structure, with associated auxiliary functions, is termed the 'Applied Mathematics Kit' (AMK). The present discussion is limited to one and two dimensional Newtonian particle mechanics in which Newton's Second Law is used to derive an equation of motion. Solving the resulting equations of motion is to be the subject of a further paper.

A natural extension to this system will be to provide methods for manipulating the *EquationOfMotion* object. We have also produced a front end for AMK using Visual Basic. The front end constructs the necessary Mathematica inputs and forces the user to stick to the appropriate modelling cycle. This is described in Mitic and Thomas[3]. We also intend to construct an interface which is directly geometric in that the placement of objects on the screen will determine attributes of objects.

2 The modelling process

Examples teach us a number of things implicitly. First, which objects exist in the first place. Second, how to use the Newtonian scientific model to produce a mathematical model. Third, we learn the necessary steps to solve a problem. AMK uses predefined objects which interact using their methods. We use the term 'linking' for this. AMK modelling is therefore done by constructing instances of objects and specifying which objects interact. The way in which they interact is handled automatically by AMK. The result is to construct a new object or objects. For example, if a particle interacts with a spring, the result is the tension in the spring, which is a force. Information about any given instance of an object can be obtained at any time by invoking a method of that object. The user has to know a general strategy for problem solving and has to be aware of the consequences of the interaction of two objects.

We can therefore provide a modelling cycle, which is summarised in the pseudocode below.

```
For each physical object in the system
  Define an instance of that object
End_For
For each particle in the system
  Repeat
    Link objects
  Until the particle and a force (or forces) are linked
  Obtain the equation of motion for the resulting equation of motion object
End_For
```

3 The Object Environment

Objects are embedded in the Mathematica O-O framework developed by Maeder, the principles of which are discussed in Maeder[4]. This framework is presented in the form of a Mathematica package and provides all that we require, integrated within an environment where symbolic manipulation is possible. A discussion of the O-O programming paradigm may be found in Maeder[5] and the implementation used here is from Maeder[6]. The procedure *Class* is used to define an object with its methods and attributes. A constructor procedure, new, creates instance variables, and message passing is implemented as a rule for applying functions to objects.

There is provision for single inheritance only, which causes problems for our purposes. In principle, it would be useful to define a superclass *Coordinates* with two subclasses, *PolarCoordinates* and *CartesianCoordinates*. It would then be possible for a subclass such as *Particle* to inherit methods from both the classes *PolarCoordinates* and *CartesianCoordinates*. If multiple inheritance were available in this way it would simplify the code considerably. In particular, coordinate transformations would have been easy.

4 The Object System

A search through standard texts on mechanics, such as Milne[7] and Dyke and Whitworth[8], reveals consistency in the objects that appear and in the situations in which they appear. Most objects in this context share attributes and methods, so it is reasonable to seek commonalty. The most fundamental shared attribute is that of a coordinate system. The object *CoordinateSystem* forms the superclass for the majority of the other objects in our system. *CoordinateSystem* encapsulates a complete implementation of polar and cartesian coordinates in two dimensions. Transformations between polars and cartesians are implemented in terms of rewrite rules. The principal attribute of *CoordinateSystem* is a coordinate which has five parameters. The first is either cartesian or polar. The second and third are the relevant space coordinates. The fourth determines the position of an initial line relative to a fixed horizontal line which points from left to right. The fifth, *Sense*, specifies the sense of rotation in the case of polar coordinates and either a left handed set or a right handed set in the case of cartesian coordinates. Among the methods of *CoordinateSystem* are *Type*, *Magnitude*, *Direction*, *X*, *Y* and *Displacement*, which serve to export details of the coordinates. The class *CoordinateSystem* is itself a subclass of the class *CoordinateTransformations*, which provides primitives for transformations between cartesian and polar coordinates.

Most other objects are subclasses of *CoordinateSystem*, and have no subclasses themselves. For example, the object *Particle* has the extra attributes *Mass* and *Time*, and has, among other methods, *Velocity* and *Acceleration*. The latter two are obtained by differentiating the space coordinates with respect to time. If an

instance of a *Particle* has polar coordinates, the components of its velocity and acceleration are automatically given in polar form. This mathematical activity makes it a great advantage to maintain the object hierarchy within the Mathematica environment. Particles are fundamental to this discussion because they are part of the vital link which produces an equation of motion.

An example of a subclass of *CoordinateSystem* which has its own subclass is *HorizontalPlane*. This has a subclass *InclinedPlane*, which is specified in terms of the additional attribute, *Slope*. *HorizontalPlane* also has a second subclass, *CircularSurface*, which is used to model problems in which a particle moves on the line of greatest slope of a circular cylinder or a sphere. It is notable that *CircularSurface* differs from *HorizontalPlane* in one respect only: it has the extra attribute *Radius*. The physical objects are quite different!

The object *EquationOfMotion* is unlike the others in that it is not a subclass of *CoordinateSystem*. There is no need for this object to inherit any methods of *CoordinateSystem*, and its own methods are unique. The object hierarchy is shown in Figure 1.

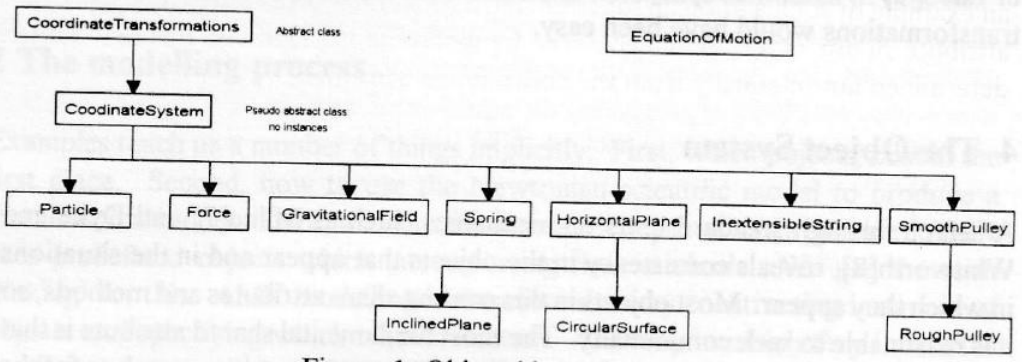

Figure 1: Object hierarchy

The class *CoordinateSystem* has an attribute *Info*, which provides a generic way of inserting information into the system without the need to define a specific method. It is used mainly to provide information, which is not needed initially, for an existing object at a later stage. For example, we might want to use the reserved symbol g for the acceleration due to gravity, and provide a numerical value for g later. This sort of information can be given when an instance of an object is constructed, as below.

```
Clear[P];
P = new[Particle, Cartesian[{{x[t],0},0,1}],
      {InitialVelocity->u},t ,m ]
```

In many cases there is no need to override methods defined in a superclass by an amended method. It is more important to replace irrelevant methods by a *Null* method.

5 Object Links

Formulating and solving problems is done by linking objects and thereby generating new objects. Eventually, a particle can be linked with a sum of forces, and this step generates the equation of motion. The procedure *MakeLink* has a polymorphic definition: one for each meaningful object combination. For example, linking a particle with a spring produces a force - the tension in the spring. There are no links for objects for which the resulting combination is not meaningful, such as a *Spring* with an *EquationOfMotion*. The principal links are:

Particle + GravitationalField = Force; *Particle + Spring = Force;*
Particle + InextensibleString = Force; *Particle + HorizontalPlane = Force;*
Particle + InclinedPlane = Force; *Particle + CircularSurface = Force;*
Particle + Force = EquationOfMotion.

In order to test for the type of objects which are being linked, boolean functions such as *ParticleQ* are implemented in terms of the environmental primitive *isa*. Here is an example of such a boolean function and a *MakeLink* procedure. The use of the attribute *Sense* makes it unnecessary to worry about signs of forces. This is determined automatically from the coordinates specified.

```
ParticleQ[x_]:= isa[x, Particle]
MakeLink[p_?ParticleQ, f_?ForceQ]:=
Which[Sense[p]==Sense[f],
       new[EquationOfMotion, p, f],
      Sense[p]==-Sense[f],
       new[EquationOfMotion, p, -f]
      ]
```

In some cases, linking two objects triggers a response which is appropriate for the situation. For example, if an inextensible string is linked with a particle, the result is a force (the tension in the string). If the coordinates of the particle were given in terms of polar coordinates, the resulting force is also given in terms of polar coordinates. Otherwise, both are given in terms of cartesian coordinates. This trick anticipates the wishes of the user. For example, we would want to stick to polar coordinates for a simple pendulum system.

There is a calculus for combining forces, so there is no need for a *MakeLink* procedure for two *Force* objects. There are polymorphic definitions which determine the *Sense* of the resulting force from the inputs. For example:

```
(f1_?ForceQ + f2_?ForceQ) :=
 new[Force,Cartesian[{{X[f1]+X[f2]},
     Angle[f1],Sense[f1]}]]  /;
          (Length[Displacement[f1]]===
              Length[Displacement[f2]]===1) &&
          (Angle[f1]===Angle[f2]) &&
```

```
(Sense[f1]===Sense[f2])
```

This obviates the need for a *MakeLink* procedure for linking two forces.

Combining forces in this way works well when the forces are simple algebraic expressions. If they are not, the time taken to produce the result is too slow, although the procedures called are identical. For example, consider a system consisting of a particle, free to move in two dimensions, attached to two non-identical springs whose other ends are fixed. The equation of motion is a complicated algebraic expression, although it is not difficult to derive it in principle. If the springs are identical, some simplifications can be made and the equation of motion can be derived much faster.

All the procedures for combining forces assume that the forces concerned are given with respect to the same coordinate axes. If this is not so they cannot be combined unless one coordinate system is rotated such that it coincides with the other. We illustrate this in the example below.

6 Examples: A Particle on a plane

These examples illustrate the important points of this modelling methodology. The inputs consist of constructors for the objects concerned, followed by link statements to produce new objects. Linking continues until an EquationOfMotion object results. The importance of the geometric aspects of problems is paramount. This is particularly so for problems in which objects have coordinates of different types or orientations.

We consider, first, a particle of mass m moving on a rough horizontal plane. It is pulled by a force of magnitude q, inclined at an angle ϕ to the horizontal. The contact between P and Q gives rise to a normal reaction, R, and a frictional component, μR. All are referred to a cartesian coordinate system as in Figure 2.

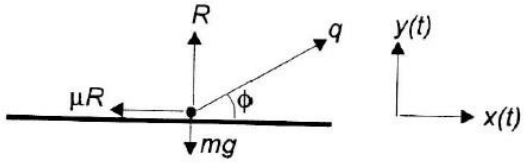

Figure 2: Particle on a horizontal plane

Four objects are constructed: the particle, P, the gravitational field, GF, the plane, HP, and the force which pulls the particle, Q. P and GF are linked to produce the weight, $W (= mg)$. P is linked with HP to produce the contact force, CF, which is a vector $(R, \mu R)$. The sum of forces is then $W + CF + Q$, and this produces the equation of motion, EoM, when linked with P. The *Equation* method of EoM is invoked to produce the standard equations of motion for this system. The required inputs follow.

```
P=new[Particle,Cartesian[{{x[t],0},0,1}],{},t,m]
GF=new[GravitationalField,Cartesian[{{0,g},0,1}],{}]
```

```
HP=new[HorizontalPlane,Cartesian[{{x[t],y[t]},0,1}],
            {NormalReaction->Nr}, mu]
Q=new[Force,Polar[{{q,phi},0,1}]]
CF=LinkObjects[P,HP]
W=LinkObjects[P,GF]
EoM=LinkObjects[P, W+CF+Q]
Equation[EoM]
```

The output corresponding to the last input is the following list.

```
{m x''[t] == -(mu Nr) + q Cos[phi],
    0 == -(g m) + Nr + q Sin[phi]}
```

We now consider what changes must be made to solve the same type of system, but with a plane inclined at an angle α to the horizontal. The natural coordinate system for this situation has axes alligned along the line of and perpendicular to the line of greatest slope of the plane. We assume that the particle moves up the slope. This system is shown in Figure 3.

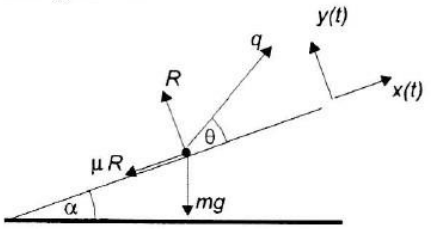

Figure 3: Particle on a slope

The same objects as before are defined, except that the plane, *IncP*, is an instance of the class *InclinedPlane*. The syntax of the term *Cartesian[{{x[t],0},alpha,1}]* means that a right handed set of coordinate axes is rotated clockwise through an angle α. It is natural to use the x and y axes as stated for all objects except the gravitational field, which is expressed in terms of non-rotated axes. In order to combine the weight, W, which results from the gravitational field, with other forces, it is necessary to express them all in terms of a common coordinate system. This is the purpose of the cast *ToCartesian[W, alpha, 1]*, which creates components of weight referred to the axes x and y in Figure 3.

```
P=new[Particle,Cartesian[{{x[t],0},alpha,1}],{},t,m]
GF=new[GravitationalField,Cartesian[{{0,-g},0,1}],{}]
IncP=new[InclinedPlane,Cartesian[{{x[t],y[t]},alpha,1}],
        {NormalReaction->Nr}, lambda,alpha]
Q=new[Force, Polar[{{q,theta},alpha,1}]]
CF=LinkObjects[P,IncP]
W=LinkObjects[P,GF]
W1=ToCartesian[W, alpha,1]
EoM=LinkObjects[P, Q+CF+W1]
Equation[EoM]
```

The output corresponding to the last input is the following list.

```
{m x''[t] ==
    -(lambda Nr) + q Cos[theta]- g m Sin[alpha],
   0 == Nr - g m Cos[alpha] + q Sin[theta]}
```

7 Conclusion

The strength of this system is that relatively complicated mathematical models can be constructed using a small number of generic objects, provided that they are linked in a meaningful way. This method of doing particle mechanics forces the user to think about the geometry of the situation and about the objects in the system. The user links the objects in a way which parallels the way in which a diagram showing forces and coordinates would be drawn. Mathematica deals with the mechanics of producing new objects.

The system works within a wide variety of situations in particle mechanics, without needing a dedicated template for each conceivable situation. It can disguise algebraic manipulations which, although tedious, should be something that the user can do if necessary. Interpretation can be slow at times, particularly when combining two forces which contain structurally complicated expressions. It helps to free memory by exiting Mathematica and restarting.

The present implemention is restricted to given objects within the context of Newton's Second Law of Motion. The conceptual framework of the system can be extended to include momentum, work and energy, which would provide scope to solve a wider class of problems. In addition, a means for the user to define new objects would be advantageous.

References

1. Dubisch, R.J. The toolkit: a notebook subclass, *Mathematica Journal*, 1990, **3**,1, Miller Freeman.
2. Viklund, L. & Fritzson, P. An object oriented language for symbolic computation - Applied to machine element analysis, in Pro. ISSAC 1992 (ed. P. Wang), pp 397-404, Berkeley, CA, USA, 1992.
3. Mitic, P & Thomas, P.G. AMK: An interface for Object-oriented Newtonian particle mechanics, *Open University internal report*, 1995.
4. Maeder, R.E. Polymorphism and Message Passing, *Mathematica Journal*, 1992, **2**, 4, Miller Freeman.
5. Maeder, R.E. Object Oriented Programming, *Mathematica Journal*, 1993, **3**, 1, Miller Freeman.
6. Maeder, R.E. *The Mathematica Programmer*, Academic Press, 1994.
7. Milne,E.A. *Vectorial Mechanics*, Methuen, 1948.
8. Dyke,P & Whitworth,R. *Guide to Mechanics*, MacMillan, 1992.