

Elements—An Object-Oriented Approach to Industrial Software Development

**Gerd Baumann
Michal Mruk**

This article discusses an object-oriented approach to industrial software development using *Mathematica*. We present the package *Elements* for structured representation of physical, engineering, and mathematical objects. This package introduces object-oriented paradigms into *Mathematica* and is used to develop a modeling environment built on a knowledge base where class and object properties and relations are maintained in a consistent, transparent, and extensible way. We show how this tool can be applied to design models parametrized by structured objects instead of just simple values.

■ Introduction

Models in engineering as well as in physics, chemistry, finance, mathematics, and economics are characterized by the following two main features.

- First, any model is based on methods representing an algorithm with which the problem is solved. These methods usually depend on each other or are at best completely independent.
- The second feature of any model is the occurrence of parameters determining physical, engineering, economic, or other properties and thus the behavior of the system.

The parameters are of central importance when variations of the model are examined. For example these quantities determine bifurcations to different states of the system. By *variations* we mean not only numeric changes but also symbolic ones.

However, modeling is a sophisticated endeavor of traversing between reality, imagination, and feasibility. It eventually converges to a mathematical descrip-

tion which captures the most essential aspects important for a particular purpose and, at the same time, is simple enough to be treated by formal methods.

We start with an observation that every modeling process involves working with objects and its primary goal is to select features relevant to a specific problem together with relations between them. Different problems in one specific area usually involve a rather limited number of objects. The problem itself is then reflected in the relations between them.

From this point of view it is natural to build a knowledge base containing descriptions of basic objects which often appear in modeling. Such a database would significantly speed up the modeling and provide a basis for rapid prototyping. It is not only a tedious and error-prone task to type in things over and over again, but, in many cases, parametrized solutions for recurring problems could be computed and stored in advance.

Our primary motivation is to facilitate the modeling process in engineering by providing a transparent and easy-to-use knowledge base of objects. In this article we stipulate and analyze requirements on such a system and present a *Mathematica* package to store and retrieve information about objects.

The proposed system is designed to be, in its nature, primarily static. It is not supposed to include “automatic solvers”, although internal and external algorithms will be utilized to perform computation whenever this appears appropriate. Extensions of the knowledge base will always be checked by a human. The reason is that the vast majority of practical problems do not provide easy answers. Even though today’s computer systems are very powerful, solutions obtained automatically still need to be post-processed by experts. This makes it impossible to simply add them to an existing knowledge base since sooner or later its consistency would inevitably be destroyed, making it useless.

□ Objects in Modeling

When we analyze requirements on the design of a system for object representation, we naturally come across some principles already used for years in software engineering: object-oriented programming. It is not surprising that the same or very similar principles will govern the design here.

Object-oriented programming introduced the concepts of classes and inheritance. It can be observed that solutions are often simpler when a hierarchical structure is imposed on the problems.

Another aspect of object-oriented programming may be even more important. Classes and their objects are encapsulated, stand-alone, and well-specified entities that behave the same way in any program—this is a key property of reusable components. This makes it possible to publish code which can be used in various settings without change.

It is clear that these paradigms apply to modeling as well. A hierarchical structure is already in the nature of all mathematical and physical objects while the closeness of used entities is a prerequisite for the use of objects in different environ-

ments. These two concepts will play a primary role in the design of a modeling environment.

□ Object-Oriented Methods in Computer Algebra

The fact that principles of object-oriented programming are compatible with structures in scientific computing led to adopting those methods into computer algebra systems. A proper combination of these two fields yields the key to a solution of our problem. On the one hand, object-oriented design provides for encapsulation and reuse of objects. On the other hand, computer algebra offers a variety of algorithms to work with them.

However, up to now there is no single system that contains these features in a well-balanced symbiosis. We encounter either systems offering implementation of the class concept (C++, Java, SML, Aldor, and so forth), or single- or multi-purpose computer algebra systems providing more or less large sets of algorithms (Maple, *Mathematica*, GAP, and so forth).

In spite of this fact, a lot of successful work has already been done using object-oriented methods in computer algebra systems. There is an extensive literature on object-oriented numerical methods [1, 2, 3, 4]. Numerical algorithms benefit from the fact that most of the data types they work on are largely supported by classical programming languages so there are quite a few choices. However, numerics do not belong to areas that would primarily benefit from object-oriented paradigms.

Another approach integrates object-oriented programming languages and computer algebra in one application [5, 6, 7, 8]. However, these solutions are mostly aimed at a few specialized fields.

The last option is to stay solely within a computer algebra system. For a long time, Maple lacked a proper concept for data encapsulation. It was finally included in Version 7. An example of object-oriented programming using numerical computation in Maple can be found in [9]. In spite of this, Maple's imperative nature is not well suited for the complex manipulations on data and code necessary when implementing object-oriented features.

Probably the first implementation of classes in *Mathematica* was done by R. Maeder in [10]. Even though this code was just a proof of concept, it demonstrated the strength of the language and proved that *Mathematica* provides powerful means for building structures and for data encapsulation.

Applications in engineering require an extensive collection of algorithms which are currently provided only by general computer algebra systems like *Mathematica* or Maple. In this article we present the package *Elements*, which provides an implementation of object-oriented features to facilitate modeling directly within *Mathematica* [11, 12]. The package, which is available from the authors' company [13], goes beyond Maeder's work and has proved to be well suited for the purpose of building a knowledge base of objects from which models are built. In the next section, we postulate and analyze requirements for a working environment that allows rapid development of models in engineering.

■ Examples

□ Car on a Bumpy Road

The problem considered here is a practical application originating from the early phase of a car design process. Engineers in the automotive business have to know in advance how a new car will react under certain environmental conditions. In the sequel, we discuss a procedure for building a model that provides basic information on the behavior of a car at the very beginning of its design. In this early stage of the process, very little information on the new car is available and thus it is sufficient to reduce the model to the most essential properties of the car. These properties can include the mass distribution on a chassis, that is, the location of the engine and other heavy components like the fuel tank, batteries, gear box, and so on. Another essential feature in this early design phase is the undercarriage. In our model, these components are combined into a multibody system interacting by forces with each other and with the environment. Knowing the preliminary behavior of the new car, the next step is to improve the model over several iterations to incorporate more details about the interacting components. Here, we restrict our considerations to the very beginning of the design process and demonstrate how an object-oriented approach can be used to model different components in a transparent way.

Figure 1 shows the simplified one-dimensional mechanical model of a car moving on a bumpy road. In principle, the car consists of three main components: the two axes and the car body. We assume that the two axes are coupled to the chassis by a dashpot and a spring. In addition to the translation, we allow a narrow rotation of the body. The wheels are in contact with the road (this interaction is modeled by introducing springs and dampers representing the two tires). The symbols shown in the figure have the following meaning: k_i , d_i with ($i = 1, 2, 3, 4$), m_k with ($k = 1, 2, 3$), a , and b are the physical and geometric parameters of the system. The k_i and d_i denote force and damping constants, m_k are the wheel and chassis masses, and a and b determine the center of the mass of the chassis. $J_{C2} = I$ is the inertia moment of the chassis. The dynamical coordinates denoted by z_1 , z_2 , z_3 , and β describe the vertical and angular motion of the car. The acting external forces on the car are due to bumps in the road and gravity. Now, the whole system can be described by Newton's equations.


```

In[1]:= Setup = Class["Setup", Class["Element"],
  {a = 1.2, Description → "distance front axis"},
  {b = 1.2, Description → "distance rear axis"},
  {v = 5, Description → "velocity of the car"},
  {stepWidth = 0.1, Description → "step width"},
  {stepHeight = 0.07, Description → "step height"}},
  {extForce[t_] :=
    Fold[Plus, 0, Table[(Tanh[100*v*(t - n stepWidth)] -
      Tanh[100*(v*(t - n stepWidth) - stepWidth)])*
      stepHeight/2, {n, 1, 4}]]}
]

```

Out[1]= <Class Setup>

Body

The class Body consists solely of geometric and physical parameters of the car body. This class also includes the initial conditions for displacement, angle, and velocity.

```

In[2]:= Body = Class["Body", Setup,
  {description = "Body",
  {m = 1200, Description → "mass"},
  {z0 = 0, Description → "initial displacement"},
  {zp0 = 0, Description → "initial velocity"},
  {h = 0.75, Description → "geometric quantity"},
  {β0 = 0, Description → "initial value for β"},
  {βp0 = 0, Description → "initial value for β prime"},
  {zFunName = z, Description → "name of the z function"},
  {betaFunName = β, Description → "name of the β function"}},
  {}]

```

Out[2]= <Class Body>

Axes

The common properties of the two axes are collected in the class CarAxis. This class contains only parameters: force and damping constants together with initial conditions for the equations of motion.

```

In[3]:= CarAxis = Class["CarAxis", Setup,
  {description = "Axis",
  {m = 42.5, Description → "mass of the axis"},
  {k1 = 150000, Description → "wheel-body force constant"},
  {d1 = 700, Description → "wheel-body damping constant"},
  {k2 = 4000, Description → "wheel-road force constant"},
  {d2 = 1800, Description → "wheel-road damping constant"},
  {z0 = 0, Description → "initial displacement"},
  {zp0 = 0, Description → "initial velocity"},
  {zFunName = z, Description → "name of the function z"}},
  {}]

```

Out[3]= <Class CarAxis>

In the next step we define classes for the front and rear axes. The front axis consists solely of a method to generate the equation of motion for the axis by considering the acting forces.

```
In[4]:= FrontAxis = Class["FrontAxis", CarAxis,
  {},
  {getEquations[body_ /; ObjectOfClassQ[body, Body]] :=
    Block[{f1 = k1 * (zFunName[t] - extForce[t]) +
      d1 * (zFunName'[t] - D[extForce[t], t]),
      f3 = k2 * ((body ° zFunName)[t] - b * (body ° betaFunName)[t] -
        zFunName[t]) + d2 * (D[(body ° zFunName)[t] -
          b * (body ° betaFunName)[t], t] - zFunName'[t])},
    {m * zFunName''[t] + (-f1 + f3) * -1 == 0,
      zFunName[0] == z0, zFunName'[0] == zp0}]
  }
]
```

```
Out[4]= <Class FrontAxis>
```

The rear axis is generated in the same way by taking the relations specific to this component into account.

```
In[5]:= RearAxis = Class["RearAxis", CarAxis,
  {},
  {getEquations[body_ /; ObjectOfClassQ[body, Body]] :=
    Block[{f2 = k1 * (zFunName[t] - extForce[t - (a + b) / v]) +
      d1 * ((body ° zFunName)'[t] - D[extForce[t - (a + b) / v], t]),
      f4 = k2 * ((body ° zFunName)[t] + a * (body ° betaFunName)[t] -
        zFunName[t]) + d2 * (D[(body ° zFunName)[t] +
          a * (body ° betaFunName)[t], t] - zFunName'[t])},
    {m * zFunName''[t] + (-f2 + f4) * -1 == 0,
      zFunName[0] == z0, zFunName'[0] == zp0}]
  }
]
```

```
Out[5]= <Class RearAxis>
```

Car

Finally, after setting up classes for the body and axes, we define a class to describe the car. This class incorporates parameters as well as methods to combine information from different components.

```
In[6]:= Car = Class["Car", Setup,
  {description = "Car",
    {body = Null, Description → "body"},
    {frontAxis = Null, Description → "front axis"},
    {rearAxis = Null, Description → "rear axis"}},
  {Ic[a0_, b0_, h0_, m03_] := Block[{}, m03 / 12 * ((a0 + b0)^2 + h0^2)],
  getEquations[] := Block[
    {f3 = frontAxis ° k2 *
      ((body ° zFunName)[t] -
        b * (body ° betaFunName)[t] - (frontAxis ° zFunName)[t]) +
      frontAxis ° d2 *
      (D[(body ° zFunName)[t] - b * (body ° betaFunName)[t], t] -
```

```

      (frontAxis ◦ zFunName) ' [t]),
    f4 = rearAxis ◦ k2 *
      ((body ◦ zFunName) [t] +
        a * (body ◦ betaFunName) [t] - (rearAxis ◦ zFunName) [t]) +
      rearAxis ◦ d2 *
      (D[(body ◦ zFunName) [t] + a * (body ◦ betaFunName) [t], t] -
        (rearAxis ◦ zFunName) ' [t])),
    Union[{body ◦ m * (body ◦ zFunName) ' ' [t] + 2 f3 == 0,
      (body ◦ zFunName) [0] == body ◦ z0,
      (body ◦ zFunName) ' [0] == body ◦ zp0, Ic[a, b, body ◦ h, body ◦ m] *
        (body ◦ betaFunName) ' ' [t] + (b * f3 - a * f4) * -1 == 0,
      (body ◦ betaFunName) [0] == body ◦ β0,
      (body ◦ betaFunName) ' [0] == body ◦ βp0},
      frontAxis ◦ getEquations[body],
      rearAxis ◦ getEquations[body]]
  ],
  getVariables[] := {frontAxis ◦ zFunName, rearAxis ◦ zFunName,
    body ◦ zFunName, body ◦ betaFunName}}
]

```

Out[6]= <Class Car>

Simulation

Having the classes for all physical components available, we need a tool to carry out the simulation. The following classes provide an efficient simulation of the model and generate a graphical representation of the results.

```

In[7]:= MakeSimulation = Class["MakeSimulation", Class["Element"],
  {description = "Simulation of the motion",
    {car = Null, Description → "Car"},
    {maxSteps = 100000,
      Description → "Maximum number of steps in NDSolve"}},
  {simulate[tstart_, tend_] :=
    Block[{nsol = getNSolution[tstart, tend]},
      $TextStyle = {FontFamily → "Arial", FontSize → 12};
      MapThread[Plot[Evaluate[{{car ◦ extForce[t]}, #1 /. nsol}],
        {t, tstart, tend}, PlotRange → All,
        PlotStyle → {RGBColor[0.25098, 0, 0.25098],
          {RGBColor[0.9, 0, 0], AbsoluteThickness[3]}},
        AxesLabel → {"t [s]", ""}, PlotLabel → #2,
        GridLines → Automatic, Frame → True] &,
        {Map[Apply[#, {t}] &, car ◦ getVariables[]], {"Front Axis",
          "Rear Axis", "Body Movement", "Rotation Angle"}}];],
    getNSolution[tstart_, tend_] :=
      NDSolve[car ◦ getEquations[],
        car ◦ getVariables[], {t, tstart, tend},
        MaxSteps → maxSteps]}
]

```

Out[7]= <Class MakeSimulation>

If we wish to see the motion of the car, it is convenient to declare a special class for making animations.

```

In[8]:= << Graphics `Shapes `

In[9]:= MakeAnimation = Class["MakeAnimation", Class["Element"],
  {description = "Animate the motion of the car"
   {path = "C:/Mma/WORK/TUMObjects/Elements05/Library/Cars/
    EEVCMedium/Generic", Description → "Path of the car model"
   }
  },
  {animate[sim1_, tstart_, tend_, δt_] :=
   Block[{names, l1, tire1, tire2, window1, window2, window3, body,
    sol = sim1`getNSolution[tstart, tend]},
    SetDirectory[path];
    names = FileNames["*. *"];
    l1 = Map[ReadList[#, Number, RecordLists → True,
      RecordSeparators →
        {If[StringMatchQ[$0operatingSystem, "Windows*"],
          "\n", "\r"}]] &, names];
    tire1 = {GrayLevel[0.752941], Polygon[
      l1[[1]] /. {x_, y_} → {x, y + z1[t]}}];
    tire2 = {GrayLevel[0.752941], Polygon[
      l1[[2]] /. {x_, y_} → {x, y + z2[t]}}];
    window1 = {GrayLevel[0.752941], Polygon[
      l1[[3]] /. {x_, y_} → {x Cos[-β[t]] + y Sin[-β[t]],
        y Cos[-β[t]] - x Sin[-β[t]] + z3[t]}}];
    window2 = {GrayLevel[0.752941], Polygon[
      l1[[4]] /. {x_, y_} → {x Cos[-β[t]] + y Sin[-β[t]],
        y Cos[-β[t]] - x Sin[-β[t]] + z3[t]}}];
    window3 = {GrayLevel[0.752941], Polygon[
      l1[[5]] /. {x_, y_} → {x Cos[-β[t]] + y Sin[-β[t]],
        y Cos[-β[t]] - x Sin[-β[t]] + z3[t]}}];
    body = {RGBColor[0, 0, 0.25098], Polygon[
      l1[[6]] /. {x_, y_} → {x Cos[-β[t]] + y Sin[-β[t]],
        y Cos[-β[t]] - x Sin[-β[t]] + z3[t]}}];
    Table[Show[Graphics[{body, tire1, tire2, window1, window2,
      window3} /. sol], AspectRatio → Automatic, Axes → False,
      Frame → True, PlotRange → {{-.1, 4.4}, {0, 1.4}},
      FrameTicks → None], {t, tstart, tend, δt}];
  ]}
]

Out[9]:= <Class MakeAnimation>

```

Objects

Up to this point, nothing more than a framework was defined in which the computation can be carried out. The classes represent essentially only templates for creating the real and calculation objects. In order to set up the simulation, we have to generate objects and specify their properties. The car is generated by a modular process incorporating the distinguished objects. At this point, it is apparent that we have a great advantage in designing a specific car compared to traditional approaches in simulation. For example, we can define different objects for axes and use them in different simulations as parameters of the underlying

model. Components (objects) of the same type can be exchanged without causing any harm to the system. In the sequel, we demonstrate how this process is carried out.

Body

First, we create the body by calling the method `new` of the class `Body`. All properties except `zFunName` and `betaFunName` will have their default valued specified in the class declaration.

```
In[10]:= b = Body ◦ new[{zFunName → z3, betaFunName → β}]
```

```
Out[10]:= <Object of Body>
```

The actual properties of the body can be checked by

```
In[11]:= GetPropertiesForm[b]
```

```
Out[11]//DisplayForm=
```

Property	Value
description	Body
a	1.2
b	1.2
betaFunName	β
h	0.75
m	1200
stepHeight	0.07
stepWidth	0.1
v	5
z0	0
zFunName	z3
zp0	0
β 0	0
β p0	0

Axes

Next, objects for the axes are created. The front axis is defined with a mass of 45.5 kg; the function representing the displacement of the center of the wheel attached to this axis is denoted by z_1 .

```
In[12]:= fa = FrontAxis ◦ new[{m → 45.5, zFunName → z1}]
```

```
Out[12]:= <Object of FrontAxis>
```

The properties of the axis can be printed by

```
In[13]:= GetPropertiesForm[fa]
```

```
Out[13]//DisplayForm=
```

Property	Value
description	Axis
a	1.2
b	1.2
d1	700
d2	1800
k1	150000
k2	4000
m	45.5
stepHeight	0.07
stepWidth	0.1
v	5
z0	0
zFunName	z1
zp0	0

Since no values for the damping and the force constant were given in the call to `new`, these properties are initialized with values specified in the class declaration.

A second version of the front axis with a stiffer damping is derived in the following line. Here, k_1 and k_2 were given different initial values.

```
In[14]:= fas = FrontAxis ◦ new[{m → 45.5,  

          d1 → 5000, d2 → 5000,  

          k1 → 150000, k2 → 150000,  

          zFunName → z1}]
```

```
Out[14]= <Object of FrontAxis>
```

For the rear axis, we just use the default settings. The coordinate of elongation is denoted by z_2 .

```
In[15]:= ra = RearAxis ◦ new[{zFunName → z2}]
```

```
Out[15]= <Object of RearAxis>
```

A check of parameters shows the default values.

```
In[16]:= GetPropertiesForm[ra]
```

```
Out[16]//DisplayForm=
```

Property	Value
description	Axis
a	1.2
b	1.2
d1	700
d2	1800
k1	150000
k2	4000
m	42.5
stepHeight	0.07
stepWidth	0.1
v	5
z0	0
zFunName	z2
zp0	0

Car

Having all necessary components available, it is easy to generate the car by incorporating the defined objects into a common model. The car object is defined by

```
In[17]:= c = Car◦new[{frontAxis → fas, rearAxis → ra, body → b}];
```

It consists of the front axis, the rear axis, and the body. We want to stress that object-oriented design allows the behavior of the model to be modified very easily by changing different components or their properties. Hence, if an engineer has access to a collection of different bodies, axes, and other objects, he would be able to create and test many different car designs without needing to adapt the model.

Simulation

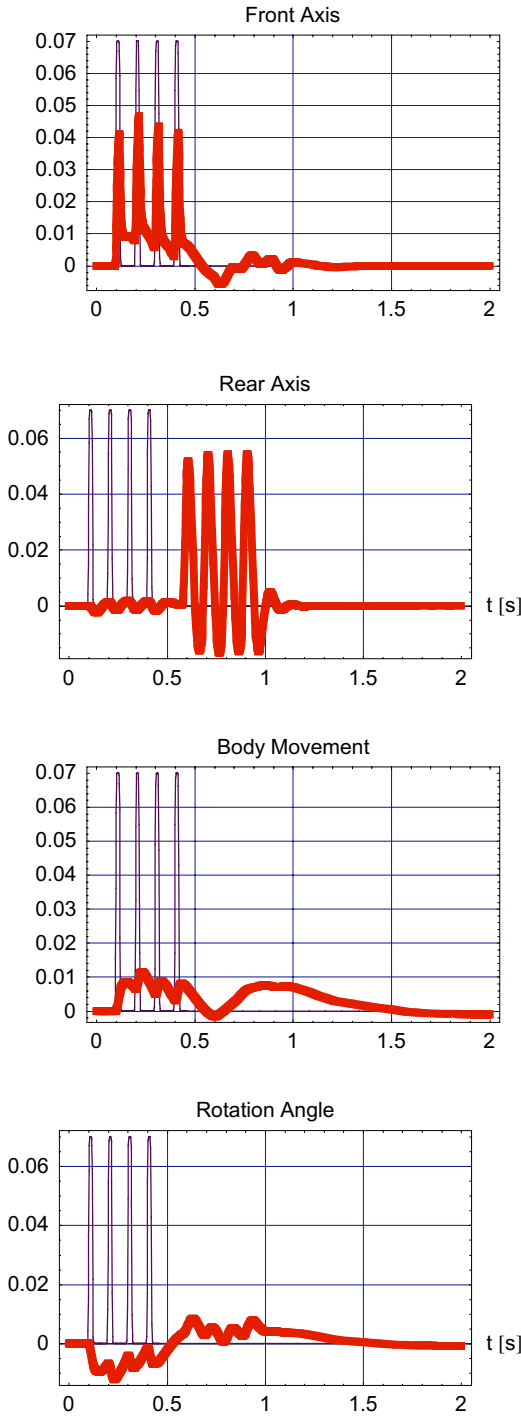
A simulation object is created from the class `MakeSimulation`. The simulation can be performed with any object of the class `Car` that is assigned to the property `car` in the call to the method `new`.

```
In[18]:= sim = MakeSimulation◦new[{car → c}]
```

```
Out[18]= <Object of MakeSimulation>
```

Finally, the actual simulation is initiated by calling the `simulate` method of the class `MakeSimulation`. The inputs for this function are just the starting and end points of the simulation interval. The result is four plots for the coordinates of the axes, the body, and the rotation angle of the body. All quantities are plotted versus the simulation time.

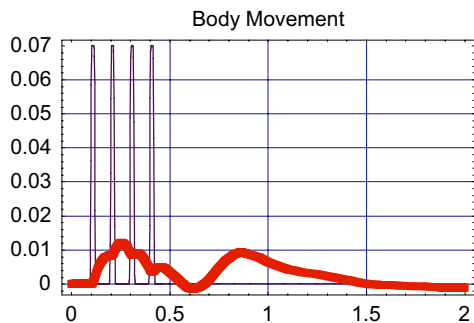
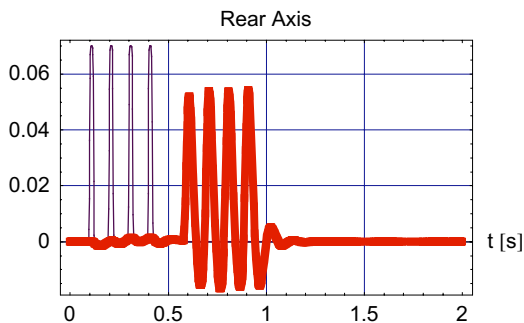
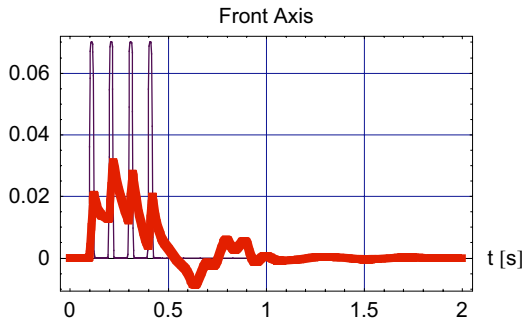
```
In[19]:= sim◦simulate[0, 2]
```

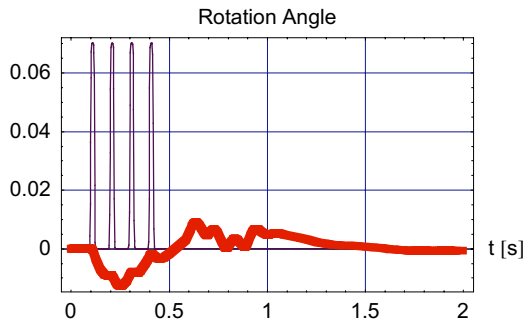


Now the hard work for an engineer starts. If he is interested in an optimization of the elongations, he must change the properties of the car in the simulation.

How to achieve the best result and how to select the best strategy depends on the experience of the individual engineer. However, within an object-oriented simulation environment, he has the flexibility to incorporate his thoughts in a quick and transparent way. For example, he can change some parameters of the front axis to achieve a better performing car.

```
In[20]:= sim ◦ car ◦ frontAxis ◦ d1 = 700;  
         sim ◦ car ◦ frontAxis ◦ k1 = 100000;  
  
In[22]:= sim ◦ simulate [0, 2]
```





Or he can replace the whole front axis by one with a smoother damper.

```
In[23]:= sim ◦ car ◦ frontAxis = fa
```

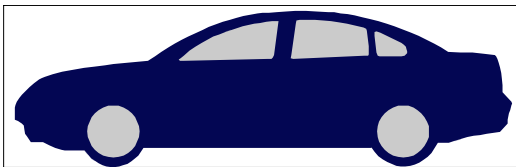
```
Out[23]= <Object of FrontAxis>
```

```
In[24]:= sim ◦ simulate[0, 2]
```

And so on (the output of the previous line was suppressed due to space limits). However, a clever engineer will resort to some optimization strategies that can be carried out by an additional class in the development environment (not shown here). In this article, we considered only manual computations to demonstrate the simulation steps.

Finally, if the results are to be presented at meetings, the behavior of the optimized car can be shown online as an animation using different simulation models or strategies. The generation of an animation object and the animation itself is started with

```
In[25]:= an = MakeAnimation ◦ new[] ;
          an ◦ animate[sim, 0, 2, 0.05]
```



The flip chart movie is suppressed in the printed version. However, the notebook version of this article shows you the movement of the car body and the wheels.

This example demonstrated that the object-oriented approach to modeling is very efficient and transparent to the user. It divides the process of modeling into two phases. In the first phase, the model and its components are implemented. In the optimization phase, the parameters of the model satisfying certain criteria are found. Both phases benefit from the availability of predefined components (objects). Using the classical approach, minor structural changes (e.g., using a different axis) require modifications by the model builder. This, in turn, may have made a subsequent verification step necessary. In contrast, the object-

oriented approach allows this to be done much more easily. A new component (e.g., an axis) can be added or exchanged without affecting the soundness of the model. This fact is of crucial importance in the industrial environment where models contain a huge number of parameters. It is necessary to have a clear and well-founded procedure to handle them. If the process to manage and work with models is clearly specified, results can be obtained more quickly, they are more predictable, and comprehensible.

□ Dynamics in Poisson–Hamilton Manifolds

Another example of an application of *Elements* in a mathematical context is the definition of operators and manifolds for Poisson–Hamiltonian systems. The problem here can be stated as follows: A set of generalized coordinates and momenta in connection with a Hamiltonian defines a Hamiltonian manifold. This manifold is also known as the phase space governed by the Hamiltonian. We assume that in addition to coordinates and momenta an additional function generating the dynamics exists. This function is usually the Hamiltonian of the system. Having the coordinates and the generating function available, the manifold acquires an algebraic structure. The algebra on the Hamiltonian manifold is generated by the Poisson’s bracket. The total Poisson–Hamilton manifold is thus equipped with coordinates, momenta, the Hamiltonian, and the Poisson’s bracket.

The question is: How do these entities relate to objects and classes in an object-oriented environment? From a practical point of view, it is necessary to define the Poisson manifold given by the coordinates and momenta associated with the algebraic structure. Thus, one class in the Poisson–Hamilton manifold represents the Poisson bracket with its properties for a given set of coordinates and momenta. The basic algebraic properties which a Poisson bracket has to satisfy are discussed next.

Let us consider two functions f and g depending on the phase space variables. Using these functions in the Poisson bracket, we can derive some general properties for this kind of bracket defined by

$$\{f, g\}_{(q,p)} = \sum_{k=1}^s \frac{\partial f}{\partial q_k} \frac{\partial g}{\partial p_k} - \frac{\partial f}{\partial p_k} \frac{\partial g}{\partial q_k}. \quad (5)$$

In the following, we use the shorthand notation $\{f, g\}$ to represent the Poisson bracket $\{f, g\}_{(q,p)}$. This notation is used whenever no confusion about the phase space variables can arise. The Poisson bracket possesses the following properties:

$$\{f, g\} = -\{g, f\} \quad \text{antisymmetry.} \quad (6)$$

If one of the functions f or g is a constant, the Poisson bracket vanishes

$$\{f, c\} = 0 = \{c, g\}. \quad (7)$$

If we have three functions f , g , and h living in the phase space, then the following properties hold:

$$\{f + b, g\} = \{f, g\} + \{b, g\} \quad \text{linearity} \quad (8)$$

$$\{f b, g\} = f \{b, g\} + b \{f, g\} \quad \text{Leibniz's rule} \quad (9)$$

$$\frac{\partial}{\partial t} \{f, g\} = \left\{ \frac{\partial f}{\partial t}, g \right\} + \left\{ f, \frac{\partial g}{\partial t} \right\} \quad \text{differentiation rule.} \quad (10)$$

If one of the two functions f or g reduces to a phase space variable, the Poisson bracket reduces to the partial derivative of the function with respect to the conjugate coordinate. For example, if g is equal to the generalized coordinate q_k or the generalized momentum p_k , the result of the Poisson bracket is

$$\{f, q_k\} = - \frac{\partial f}{\partial p_k} \quad (11)$$

$$\{f, p_k\} = \frac{\partial f}{\partial q_k}. \quad (12)$$

If we chose the coordinates of the phase space for both f and g , we obtain the fundamental Poisson brackets

$$\{q_i, q_j\} = 0 \quad (13)$$

$$\{p_i, p_j\} = 0 \quad (14)$$

$$\{q_i, p_i\} = \sum_{k=1}^s \frac{\partial q_i}{\partial q_k} \frac{\partial p_j}{\partial p_k} - \frac{\partial q_i}{\partial p_k} \frac{\partial p_j}{\partial q_k} = \sum_{k=1}^s \delta_{ik} \delta_{jk} = \delta_{ij}. \quad (15)$$

These relations of the fundamental Poisson brackets represent the algebraic structure and are the basis of quantum mechanics. Any three functions of the phase space satisfy a special relation, the so-called Jacobi identity

$$\{f, \{g, b\}\} + \{g, \{b, f\}\} + \{b, \{f, g\}\} = 0. \quad (16)$$

These properties determine the algebraic quality of the Poisson bracket. Especially, linearity, antisymmetry, and the Jacobi identity define the related Lie algebra of the bracket.

Another important property of the Poisson bracket is the ability to derive from two conserved quantities \mathcal{J}_1 and \mathcal{J}_2 another conserved quantity

$$\{\mathcal{J}_1, \mathcal{J}_2\} = \text{const.} \quad (17)$$

This behavior is known as Poisson's theorem. A direct proof is feasible if we assume that \mathcal{J}_1 and \mathcal{J}_2 are independent of time. By replacing the third function in the Jacobi identity by the Hamiltonian of the system, we get

$$\{H, \{\mathcal{J}_1, \mathcal{J}_2\}\} + \{\mathcal{J}_1, \{\mathcal{J}_2, H\}\} + \{\mathcal{J}_2, \{H, \mathcal{J}_1\}\} = 0. \quad (18)$$

Since $\{\mathcal{J}_2, H\} = 0$ and $\{H, \mathcal{J}_1\} = 0$, we find

$$\{H, \{\mathcal{J}_1, \mathcal{J}_2\}\} = 0. \quad (19)$$

Thus, the bracket $\{f_1, f_2\}$ is also a conserved quantity. We note that the application of Poisson's theorem will not always provide new conserved quantities because the number of conserved quantities of a standard mechanical system is finite. It is known that the total number of conserved quantities is given by $2n - 1$, if n is the degree of freedom in the phase space. Thus, Poisson's theorem sometimes delivers trivial constants or the resulting conserved quantity is a function of the original conserved quantities f_1 and f_2 . If both cases fail, we gain a new conserved quantity.

It is obvious from the previous definitions and properties that the pure algebraic structure is independent of the phase space variables. Thus we can separate the problem into the algebraic properties (equations (6) through (10)) and the coordinate-specific definition of the PB given by equation (5). This separation is the basis of our class for the Poisson bracket PB.

Poisson Brackets

The following definition of the class PB represents a general definition of the Poisson bracket. This class is determined by the algebraic properties and does not incorporate the coordinates on which it operates. The class definition expresses properties, such as bilinearity, the product relation for each argument of the Poisson bracket, and the calculation of the Poisson expression. The properties implemented represent the algebraic structure of a Lie algebra.

```

In[1]:= PB = Class["PB", Class["Element"],
  {description = "Poisson Bracket",
    {P = Null, Description → "momenta"},
    {Q = Null, Description → "coordinates"},
    {T = t, Description → "independent variable"}},
  {(* --- constant factor extraction --- *)PoissonBracket[a_f_, g_] :=
    a PoissonBracket[f, g] /; (Apply[And, Map[FreeQ[a, #] &, Q]] ∧
      Apply[And, Map[FreeQ[a, #] &, P]]),
    (* --- constant factor extraction --- *)
    PoissonBracket[f_, a_g_] :=
      a PoissonBracket[f, g] /; (Apply[And, Map[FreeQ[a, #] &, Q]] ∧
        Apply[And, Map[FreeQ[a, #] &, P]]),
    (* --- Linearity --- *)
    PoissonBracket[a_ + f_, g_] :=
      PoissonBracket[a, g] + PoissonBracket[f, g],
    (* --- Linearity --- *)
    PoissonBracket[f_, a_ + g_] :=
      PoissonBracket[f, a] + PoissonBracket[f, g],
    (* --- product relation --- *)
    PoissonBracket[f_h_, g_] :=
      f PoissonBracket[h, g] + h PoissonBracket[f, g],
    (* --- product relation --- *)
    PoissonBracket[g_, f_h_] :=
      f PoissonBracket[g, h] + h PoissonBracket[g, f],
    (* --- Calculation of the bracket --- *)
    PoissonBracket[f_, g_] := Block[{},
      Fold[Plus, 0, MapThread[(∂#1 f ∂#2 g - ∂#2 f ∂#1 g) &, {Q, P}]]],
  }

```

```
PB[Pnew_List, Qnew_List, Tnew_Symbol] :=
  Block[{f, P = Pnew; Q = Qnew; T = Tnew}
  ]
```

Out[1]= <Class PB>

Our aim here is to use a Poisson bracket within *Mathematica* in the same way as in a textbook. This requires that we first define the Poisson manifold as an object and use this object in our calculations. The following line generates a template for the Poisson bracket combining the Poisson manifold as an object and the algebraic properties of the bracket.

```
In[2]:= Notation[{f_, g_]_obj_ <=> obj_ ◦ PoissonBracket[f_, g_]};
```

This allows us to proceed in the same way as in mechanics textbooks.

A Simple Poisson Bracket

In the following, we give a few examples of Poisson manifolds and their application to a function defined on them.

The first example describes a two-dimensional manifold in generalized coordinates p and q . The manifold (object pk) is derived from the class PB.

```
In[3]:= pk = PB ◦ new[{P → {p[t]}, Q → {q[t]}}]
```

Out[3]= <Object of PB>

The properties of this manifold can be printed using

```
In[4]:= GetPropertiesForm[pk]
```

Out[4]//DisplayForm=

Property	Value
description	Poisson Bracket
P	{p[t]}
Q	{q[t]}
T	t

We get the coordinates q and p depending by default on time t .

Now, let us apply the Poisson manifold pk to two functions f and g depending on p and q .

```
In[5]:= {f[p[t], q[t]], g[p[t], q[t]}}_pk
```

Out[5]= $-g^{(0,1)} [p[t], q[t]] f^{(1,0)} [p[t], q[t]] + f^{(0,1)} [p[t], q[t]] g^{(1,0)} [p[t], q[t]]$

The result is the definition of the Poisson bracket given in equation (5) for $s = 1$.

A second example is concerned with the Hamiltonian of a single particle moving in a potential $V = V(q)$. The application to this Hamiltonian provides us with the momentum p .

```
In[6]:= {  $\frac{p[t]^2}{2} + v[q[t]]$ , p[t] }_pk
```

Out[6]= $V' [q[t]]$

Another example of a general function H depending on both coordinates yields for the q coordinate

$$\text{In}[7]:= \{H[p[t], q[t]], q[t]\}_{pk}$$

$$\text{Out}[7]= -H^{(1,0)}[p[t], q[t]]$$

An additional example deals with a general Hamiltonian H and an arbitrary function f depending on both coordinates of the manifold.

$$\text{In}[8]:= \{\alpha H[q[t], p[t]], f[q[t], p[t]]\}_{pk}$$

$$\text{Out}[8]= \alpha (-H^{(0,1)}[q[t], p[t]] f^{(1,0)}[q[t], p[t]] + f^{(0,1)}[q[t], p[t]] H^{(1,0)}[q[t], p[t]])$$

This result demonstrates that linearity is recognized by the manifold. The next relation represents Jacobi's identity for three functions H , f , and g depending on both coordinates.

$$\begin{aligned} \text{In}[9]:= & \{\alpha H[q[t], p[t]], \{f[q[t], p[t]], g[q[t], p[t]]\}_{pk}\}_{pk} + \\ & \{f[q[t], p[t]], \{g[q[t], p[t]], \alpha H[q[t], p[t]]\}_{pk}\}_{pk} + \\ & \{g[q[t], p[t]], \{\alpha H[q[t], p[t]], f[q[t], p[t]]\}_{pk}\}_{pk} // \text{Simplify} \end{aligned}$$

$$\text{Out}[9]= 0$$

Higher-Dimensional Manifolds

The class PB is defined in such a way that it allows creating a general Poisson manifold in $2n$ -dimensions. To make things easier, let us define a four-dimensional Poisson manifold with coordinates q_i and p_i ($i = 1, 2$). The manifold is derived from the class PB and stored in the object pb2.

$$\text{In}[10]:= \text{pb2} = \text{PB} \circ \text{new}\{\{P \rightarrow \{p1[t], p2[t]\}, Q \rightarrow \{q1[t], q2[t]\}\}$$

$$\text{Out}[10]= \langle \text{Object of PB} \rangle$$

The properties of this Poisson manifold are

$$\text{In}[11]:= \text{GetPropertiesForm}[\text{pb2}]$$

$$\text{Out}[11]//\text{DisplayForm} =$$

Property	Value
description	Poisson Bracket
P	{p1[t], p2[t]}
Q	{q1[t], q2[t]}
T	t

Again, the momenta and coordinates are defined in terms of the independent variable t .

Global relations incorporating all coordinates for a two-particle Hamiltonian can be obtained by the following line

$$\text{In[12]:= Map}\left[\left\{\frac{\mathbf{p1}[\mathbf{t}]^2}{2} + \frac{\mathbf{p2}[\mathbf{t}]^2}{2} + \mathbf{v}[\mathbf{q1}[\mathbf{t}], \mathbf{q2}[\mathbf{t}]], \#\right\}_{\text{pb2}}, \&, \{\mathbf{p1}[\mathbf{t}], \mathbf{p2}[\mathbf{t}], \mathbf{q1}[\mathbf{t}], \mathbf{q2}[\mathbf{t}]\}\right]$$

$$\text{Out[12]= } \{V^{(1,0)}[\mathbf{q1}[\mathbf{t}], \mathbf{q2}[\mathbf{t}]], V^{(0,1)}[\mathbf{q1}[\mathbf{t}], \mathbf{q2}[\mathbf{t}]], -\mathbf{p1}[\mathbf{t}], -\mathbf{p2}[\mathbf{t}]\}$$

The function used as the first argument of the Poisson bracket is a Hamiltonian consisting of two terms for the kinetic energy and a general interaction potential V . For another two-dimensional Hamiltonian with a different potential V , we find

$$\text{In[13]:= Map}\left[\left(\left\{\frac{\mathbf{p1}[\mathbf{t}]^2}{2} + \frac{\mathbf{p2}[\mathbf{t}]^2}{2} + \mathbf{v}[\mathbf{q1}[\mathbf{t}]], \#\right\}_{\text{pb2}}\right) \&, \{\mathbf{p1}[\mathbf{t}], \mathbf{p2}[\mathbf{t}], \mathbf{q1}[\mathbf{t}], \mathbf{q2}[\mathbf{t}]\}\right]$$

$$\text{Out[13]= } \{V[\mathbf{q1}[\mathbf{t}]], 0, -\mathbf{p1}[\mathbf{t}], -\mathbf{p2}[\mathbf{t}]\}$$

The following example demonstrates that the Poisson bracket with two identical arguments, here a Hamiltonian, vanishes

$$\text{In[14]:= } \left\{\frac{\mathbf{p1}[\mathbf{t}]^2}{2} + \frac{\mathbf{p2}[\mathbf{t}]^2}{2} + \mathbf{v}[\mathbf{q1}[\mathbf{t}]], \frac{\mathbf{p1}[\mathbf{t}]^2}{2} + \frac{\mathbf{p2}[\mathbf{t}]^2}{2} + \mathbf{v}[\mathbf{q1}[\mathbf{t}]]\right\}_{\text{pb2}}$$

$$\text{Out[14]= } 0$$

So far, we defined the algebraic structure and derived certain types of Poisson manifolds by specifying the phase space variables. However, the knowledge of these manifolds allows us to go a step further to a real application and to introduce Hamilton's equations. Knowing the Poisson manifold, it is straightforward to write down these equations.

Definition of Hamilton's Equations

For a set of generalized coordinates (q_k, p_k) , Hamilton's equations are defined by

$$\dot{q}_k = \frac{\partial H}{\partial p_k} \tag{20}$$

$$\dot{p}_k = -\frac{\partial H}{\partial q_k}. \tag{21}$$

This set of equations is related to Poisson's bracket by considering a function f in the phase space depending on the phase space coordinates q_k, p_k , and t as

$$f = f(q_k, p_k, t). \tag{22}$$

The structure of the phase space is governed by Poisson's bracket which can be seen by differentiating f with respect to t

$$\frac{df}{dt} = \frac{\partial f}{\partial t} + \sum_{k=1}^p \frac{\partial f}{\partial q_k} \dot{q}_k + \frac{\partial f}{\partial p_k} \dot{p}_k. \tag{23}$$

By inserting Hamilton's equations of motion into this expression we gain

$$\frac{df}{dt} = \frac{\partial f}{\partial t} + \sum_{k=1}^{\rho} \frac{\partial f}{\partial q_k} \frac{\partial H}{\partial p_k} - \frac{\partial f}{\partial p_k} \frac{\partial H}{\partial q_k}. \quad (24)$$

Let us abbreviate by

$$\sum_{k=1}^{\rho} \frac{\partial f}{\partial q_k} \frac{\partial H}{\partial p_k} - \frac{\partial f}{\partial p_k} \frac{\partial H}{\partial q_k} = \{f, H\}_{(q,p)} \quad (25)$$

the Poisson's bracket defined on the phase space spanned by the coordinates q_k and p_k . The subscript $\{q, p\}$ denotes the set of variables of the phase space. Using this definition, the temporal change of f becomes

$$\frac{df}{dt} = \frac{\partial f}{\partial t} + \{f, H\}_{(q,p)}. \quad (26)$$

Note that this relation allows us to calculate the temporal changes of any function depending on the phase space variables.

We already encountered conserved quantities which have the property that their temporal change vanishes. This vanishing can be expressed by the Poisson bracket in a very convenient way. The conservation of the quantity f implies

$$\frac{df}{dt} = 0, \quad (27)$$

which is identical with

$$\frac{\partial f}{\partial t} + \{f, H\}_{(q,p)} = 0. \quad (28)$$

The previous relations define a connection between a Poisson manifold and a Hamilton system. Thus, the two properties connected by the manifold are denoted as a Poisson–Hamilton (PH) manifold. The equations of motion for any kind of function in this manifold can be derived by means of the Poisson bracket via equation (26).

It is convenient here to define a class for Hamilton's equations which inherits the properties of the Poisson bracket. The properties of the Poisson manifold are equivalent to the properties of the Hamilton manifold.

The following lines define the class `HamiltonEquations` based on equation (28).

```
In[15]:= HamiltonEquations = Class["HamiltonEquations", PB,
  {description = "Hamilton's equations"},
  {HamEqs[H_, V_ /; FreeQ[V, List]] := ∂ₜ V == PoissonBracket[H, V],
  HamEqs[H_, V_List] := Map[HamEqs[H, #] &, V],
  HamEqs[H_] := Map[HamEqs[H, #] &, Flatten[{P, Q}]],
  HamiltonEquations[Pnew_List, Qnew_List, Tnew_Symbol] :=
  Block[{P = Pnew; Q = Qnew; T = Tnew}]
]
```

```
Out[15]= <Class HamiltonEquations>
```

To handle the class for Hamiltonian equations and the derived objects in the same way as in textbooks or in the case of Poisson brackets, we introduce the notation

In[16]:= **Notation** [\mathcal{H}_{Eq}^{obj-} [f_] \Leftrightarrow obj_ \circ **HamEqs** [f_]]

and define the corresponding palette.

In[17]:=

$\{\square, \square\}_\square$
$\mathcal{H}_{Eq}^\square [\square]$

Out[17]= $\{\{\square, \square\}_\square, \{\mathcal{H}_{Eq}^\square [\square]\}\}$

Having these tools available, we can apply the classes to specific problems.

Hamilton's Equations

As a first example, let us examine a PH manifold with a single coordinate and a single momentum. The object defining the PH manifold is created by

In[18]:= **ham1** = **HamiltonEquations** \circ **new** [{P \rightarrow {p[t]}, Q \rightarrow {q[t]}}]

Out[18]= <Object of HamiltonEquations>

Specifying a single particle Hamiltonian by kinetic and potential energies, we can derive the set of Hamilton's equations by applying the manifold to the Hamiltonian.

In[19]:= $\mathcal{H}_{Eq}^{ham1} \left[\frac{p[t]^2}{2} + v[q[t]] \right]$

Out[19]= {p'[t] == v'[q[t]], q'[t] == -p[t]}

The result is a system of equations defining the dynamic motion of this particle.

A second example is concerned with a four-dimensional PH manifold. The generalized coordinates and the momenta are primarily given by $q_1, q_2, p_1,$ and p_2 .

In[20]:= **ham2** = **HamiltonEquations** \circ **new** [{P \rightarrow {p1[t], p2[t]}, Q \rightarrow {q1[t], q2[t]}}]

Out[20]= <Object of HamiltonEquations>

As an example, let us consider the double pendulum. The Hamiltonian for this system reads

In[21]:= **HamDoublePendulum** =

$$\frac{1}{2 l_1^2 l_2^2 m_2 (m_1 + m_2 \sin[\theta_1[t] - \theta_2[t]]^2)} (l_2^2 m_2 p_1[t]^2 + l_1^2 (m_1 + m_2) p_2[t]^2 - 2 m_2 l_1 l_2 p_1[t] p_2[t] \cos[\theta_1[t] - \theta_2[t]]) - m_2 g l_2 \cos[\theta_2[t]] - (m_1 + m_2) g l_1 \cos[\theta_1[t]]$$

Out[21]= $-g \cos[\theta_2[t]] l_2 m_2 - g \cos[\theta_1[t]] l_1 (m_1 + m_2) + \frac{l_2^2 m_2 p_1[t]^2 - 2 \cos[\theta_1[t] - \theta_2[t]] l_1 l_2 m_2 p_1[t] p_2[t] + l_1^2 (m_1 + m_2) p_2[t]^2}{2 l_1^2 l_2^2 m_2 (m_1 + \sin[\theta_1[t] - \theta_2[t]]^2 m_2)}$

where p_i ($i = 1, 2$) are the generalized momenta, l_i , m_i are the inertia momenta and the masses of the particles, and θ_i are the angles of deviation. The PH manifold `ham2` defined earlier does not exactly correspond to the variables used in the Hamiltonian. However, we are able to change the coordinate names by setting the properties of the PH manifold using

```
In[22]:= SetProperty[ham2, {P -> {p1[t], p2[t]}, Q -> {\theta1[t], \theta2[t]}}
```

Now the PH manifold is defined for the coordinates

```
In[23]:= GetPropertiesForm[ham2]
```

```
Out[23]//DisplayForm=
```

Property	Value
description	Hamilton's equations
P	{p1[t], p2[t]}
Q	{\theta1[t], \theta2[t]}
T	t

The four equations of motion can then be gained by

```
In[24]:= equationsOfMotion = N_{Eq}^{ham2} [HamDoublePendulum]
```

```
Out[24]= {p1'[t] == g Sin[\theta1[t]] l1 (m1 + m2) +
          1
          2 l1^2 l2^2 m2 ( -
          2 Sin[\theta1[t] - \theta2[t]] l1 l2 m2 p1[t] p2[t] -
          (2 Cos[\theta1[t] - \theta2[t]] Sin[\theta1[t] - \theta2[t]] m2 (l2^2 m2 p1[t]^2 -
          2 Cos[\theta1[t] - \theta2[t]] l1 l2 m2 p1[t] p2[t] + l1^2 (m1 + m2) p2[t]^2) ) /
          (m1 + Sin[\theta1[t] - \theta2[t]]^2 m2) )^2 }, p2'[t] == g Sin[\theta2[t]] l2 m2 +
          1
          2 l1^2 l2^2 m2 ( -
          2 Sin[\theta1[t] - \theta2[t]] l1 l2 m2 p1[t] p2[t] +
          (2 Cos[\theta1[t] - \theta2[t]] Sin[\theta1[t] - \theta2[t]] m2
          (l2^2 m2 p1[t]^2 - 2 Cos[\theta1[t] - \theta2[t]] l1 l2 m2 p1[t] p2[t] +
          l1^2 (m1 + m2) p2[t]^2) ) / (m1 + Sin[\theta1[t] - \theta2[t]]^2 m2) )^2 },
          \theta1'[t] ==
          -2 l1^2 m2 p1[t] + 2 Cos[\theta1[t] - \theta2[t]] l1 l2 m2 p2[t]
          2 l1^2 l2^2 m2 (m1 + Sin[\theta1[t] - \theta2[t]]^2 m2) },
          \theta2'[t] ==
          2 Cos[\theta1[t] - \theta2[t]] l1 l2 m2 p1[t] - 2 l1^2 (m1 + m2) p2[t]
          2 l1^2 l2^2 m2 (m1 + Sin[\theta1[t] - \theta2[t]]^2 m2) }
```

They represent the dynamics of the double pendulum in the PH manifold. This example demonstrates the fact that using an object-oriented approach, the knowledge of the Hamiltonian, and the coordinates for the manifold suffice to derive the equations of motion in a portion of a second. The ease of use and the close connection to textbook presentations allows for fast manipulation and

reliable calculation of results. Besides these examples, many other applications of *Elements* to similar subjects are ahead.

■ Conclusions

In this article we discussed basic guidelines for building a sophisticated object-oriented modeling environment based on a knowledge base containing predefined components. We presented the package *Elements* that implements the core functionality of such a system that is suitable to be used for both industrial and mathematical applications. We demonstrated that combining the computational power of computer algebra with object-oriented paradigms provides a firm foundation for speeding up the work flow in industrial modeling processes. The soundness and consistency of the stored information ensures its practical applicability in a variety of settings.

■ References

- [1] P. F. Dubois, *Object Technology for Scientific Computing: Object-Oriented Numerical Software in Eiffel and C*, Englewood Cliffs, NJ: Prentice Hall, 1997.
- [2] K. Dowd, *High Performance Computing*, A Nutshell Handbook, Sebastopol, CA: O'Reilly & Associates, Inc., 1993.
- [3] G. Buzzi-Ferraris, *Scientific C++: Building Numerical Libraries the Object-Oriented Way*, Englewood Cliffs, NJ: Prentice Hall, 1993.
- [4] D. Yang, *C++ and Object-Oriented Numeric Computing for Scientists and Engineers*, New York: Springer-Verlag, 2000.
- [5] M. Berth, F.-M. Moser, and A. Triulzi, "Implementing Computational Services Based on OpenMath," in Ganzha et al., pp. 49–60.
- [6] M. Göbel, W. Küchlin, S. Müller, and A. Weber, "Extending a Java Based Framework for Scientific Software-Components," in *Computer Algebra in Scientific Computing (CASC'99), Proceedings of the Second International Workshop on Computer Algebra in Scientific Computing*, Munich (V. G. Ganzha, E. W. Mayr, and E. V. Vorozhtsov, eds.), New York: Springer-Verlag, 1999 pp. 207–223.
- [7] *Modelica—A Unified Object-Oriented Language for Physical Systems Modeling Language Specification Version 2.0*, Linköping, Sweden: Modelica Association, (Feb 2002) www.modelica.org.
- [8] P. Fritzson, J. Gunnarson, and M. Jirstrand, "MathModelica: An Extensible Modeling and Simulation Environment with Integrated Graphics and Literate Programming," in *The Proceedings of the Second International Modelica Conference*, DLR, Oberpfaffenhofen, Germany (M. Otter, ed.), 2002 pp. 41–54 www.mathcore.com/documents.
- [9] V. G. Ganzha, D. Chibisov, and E. V. Vorozhtsov, "GROOME—Tool Supported Graphical Object-Oriented Modeling for Computer Algebra and Scientific Computing," in *Computer Algebra in Scientific Computing (CASC 2001), Proceedings of the Fourth International Workshop on Computer Algebra in Scientific Computing*, Konstanz (V. G. Ganzha, E. W. Mayr, and E. V. Vorozhtsov, eds.), New York: Springer-Verlag, 2001 pp. 213–23.

- [10] R. Maeder, *The Mathematica Programmer*, Boston: AP Professional, 1994.
- [11] M. Mruk and G. Baumann, "Elements: A Package for Object Oriented Programming in Mathematica," in *Computer Algebra in Scientific Computing (CASC 2002), Proceedings of the Fifth International Workshop on Computer Algebra in Scientific Computing*, Yalta, Ukraine (V. G. Ganzha, E. W. Mayr, and E. V. Vorozhtsov, eds.), Munich: Technical University, 2002 pp. 227–236.
- [12] A. Weber, G. Simon, W. Küchlin, and J. Hoss, "Lessons Learned from Using CORBA for Components in Scientific Computing," in Ganzha et al., pp. 409–422.
- [13] *Elements*, KnowledgeGeneration, Edelweiss-St. 1, D-82211 Herrsching a.A., Germany (contact Gerd.Baumann@KnowledgeGeneration.de).
- [14] V. G. Ganzha, E. W. Mayr, and E. V. Vorozhtsov, eds., *Computer Algebra in Scientific Computing (CASC 2000), Proceedings of the Third International Workshop on Computer Algebra in Scientific Computing*, Samarkand, New York: Springer-Verlag, 2000.

About the Authors

Gerd Baumann is a professor in the Mathematical Physics department at the University of Ulm. He is Head of the Mathematics Department at the German University in Cairo (GUC) and also associated with the faculty of Informatics at the Technical University of Munich where he lectures on symbolic computing. Baumann is the author of *Mathematica in Theoretical Physics* and *Symmetry Analysis of Differential Equations with Mathematica*.

Michal Mruk is associated with the faculty of Informatics at the Technical University of Munich. He holds a Ph.D. degree in mathematics from the Research Institute for Symbolic Computation, Hagenberg, and is currently completing his Habilitation at the Technical University of Munich.

Gerd Baumann

*Department of Mathematical Physics
University of Ulm
Albert-Einstein-Allee 11
D-89069 Ulm
Gerd.Baumann@KnowledgeGeneration.de*

Michal Mruk

*Institute of Informatics
Technical University of Munich
Boltzmannstrasse 3
85748 Garching b. München
mruk@scimod.de*