Programming in Mathematica using object based paradigm

Nasser M. Abbasi

January 29, 2024 Compiled on January 29, 2024 at 3:08am

Contents

1 Introduction

Mathematica can be effectively used for object based programming. It is well known that using object based programming helps in managing the complexity of large programs. Using Object based programming in Mathematica can lead to the best of both worlds: object based combined with functional programming.

Object based can be used to help in organizing the program in the large and functional programming is used in the actual implementation of the Classes methods.

The idea is simple. A Module acts as what is the Class in standard OO languages. Inside this module will be additional inner Modules. These inner Modules act as the Class methods. Inner Modules can be made public or private.

By adding the name of an inner Module in the list of the local variables of the outer Module, the inner Module becomes private and is seen and only be called from other inner Modules.

The outer Module local variables are the Class private variables and these variables can be accessed only by the Class inner modules.

An object is first created as an instance of the outer Module and from then on this object can be used in the same way as an object is used standard OO by using the notation object@method[parameters] where the `@` here acts as the dot "." acts in common OO languages.

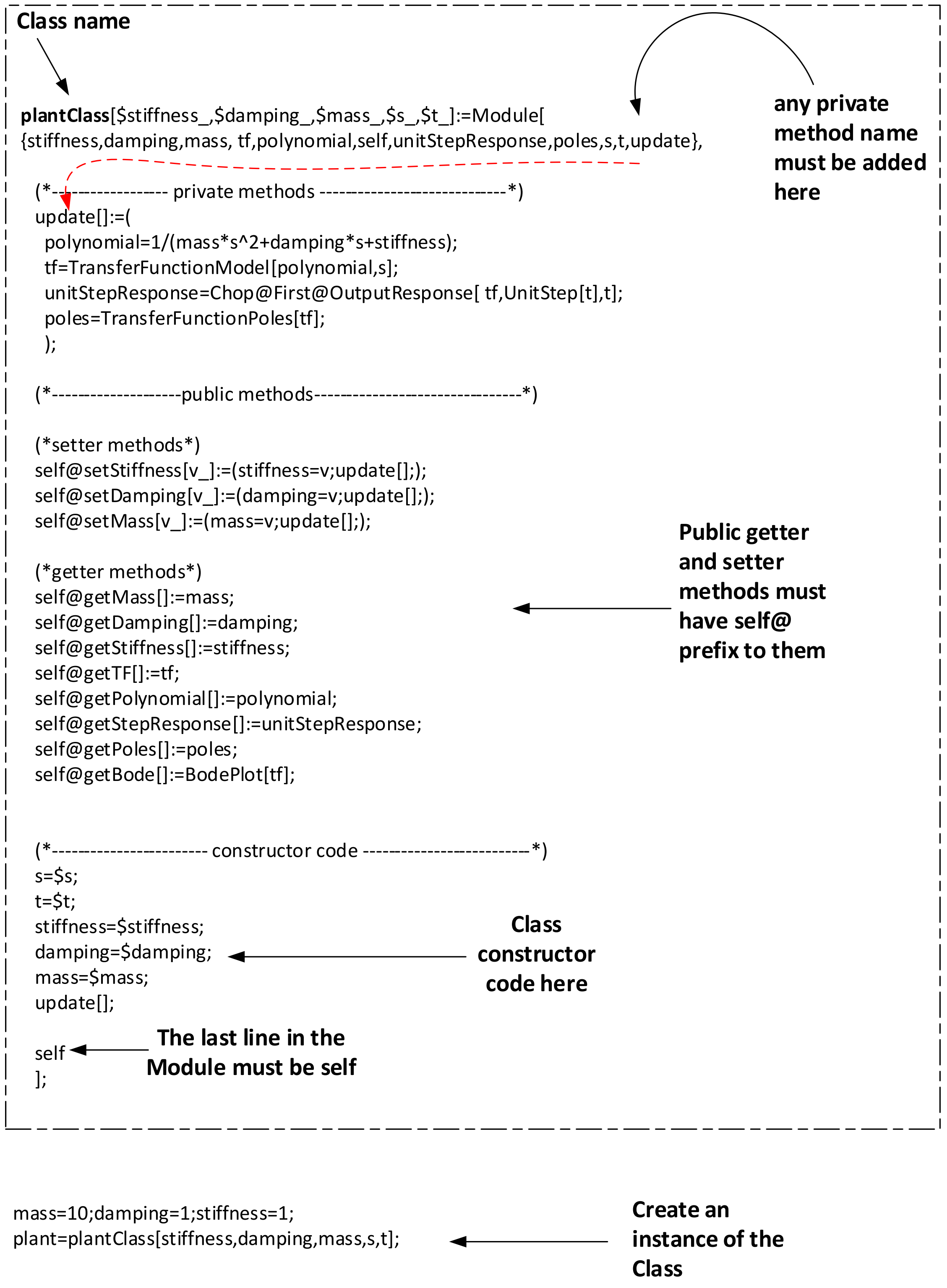
In other words, the dot is replaced by `@` and almost everything else remains the same. This makes the notation easier to use for someone who is more familiar with common OO notations.

The following example illustrates this idea where a Class that represents a second order system is defined and used to make an instance of a second order system (such as a spring-mass-damper) and the object methods are used for some basic control system operations to illustrate how to use it.

The Class is called plantClass (which is the outer Module name). To create a specific instance of this Class, the constructor is first called using the call

```
plant=plantClass[parameters]
```

The following diagram summarizes this setup, followed by the Mathematica code itself


Class name
any private method name must be added here
Public getter and setter methods must have self@ prefix to them
Class constructor code here
The last line in the Module must be self
Create an instance of the Class

```
plantClass[$stiffness_,$damping_,$mass_,$s_,$t_]:=Module[
{stiffness,damping,mass, tf,polynomial,self,unitStepResponse,poles,s,t,update},

  (*------------------ private methods ----------------------------*)
  update[]:=(
   polynomial=1/(mass*s^2+damping*s+stiffness);
   tf=TransferFunctionModel[polynomial,s];
   unitStepResponse=Chop@First@OutputResponse[ tf,UnitStep[t],t];
   poles=TransferFunctionPoles[tf];
   );

  (*------------------public methods------------------------------*)

  (*setter methods*)
  self@setStiffness[v_]:=(stiffness=v;update[];);
  self@setDamping[v_]:=(damping=v;update[];);
  self@setMass[v_]:=(mass=v;update[];);

  (*getter methods*)
  self@getMass[]:=mass;
  self@getDamping[]:=damping;
  self@getStiffness[]:=stiffness;
  self@getTF[]:=tf;
  self@getPolynomial[]:=polynomial;
  self@getStepResponse[]:=unitStepResponse;
  self@getPoles[]:=poles;
  self@getBode[]:=BodePlot[tf];


  (*---------------------- constructor code -----------------------*)
  s=$s;
  t=$t;
  stiffness=$stiffness;
  damping=$damping;
  mass=$mass;
  update[];

  self
  ];


mass=10;damping=1;stiffness=1;
plant=plantClass[stiffness,damping,mass,s,t];
```

Figure 1: Basic class layout

2 Code implementation of the Class and how to use it

In the following, the complete code of the Class and example of using it are given. A plant is created, then the step response is plotted, then a Manipulate is made that uses this Class where the plant's mass, damping and stiffness are used as Manipulate sliders control variables to be changed and the plant step response is updated each time.

```
plantClass[$stiffness_, $damping_, $mass_, $s_, $t_] :=
  Module[{stiffness, damping, mass, tf, polynomial, self,
    unitStepResponse, poles, s, t, update},
   SetAttributes[self, HoldAll];
   (*------------------ private methods ----------------------------*)
   update[] := (
     polynomial = 1/(mass*s^2 + damping*s + stiffness);
     tf = TransferFunctionModel[polynomial, s];
     unitStepResponse = Chop@First@OutputResponse[ tf, UnitStep[t], t];
     poles = TransferFunctionPoles[tf];
     );

   (*--------------------public methods-------------------------------*)

   (*setter methods*)
   self@setStiffness[v_] := (stiffness = v; update[];);
   self@setDamping[v_] := (damping = v; update[];);
   self@setMass[v_] := (mass = v; update[];);

   (*getter methods*)
   self@getMass[] := mass;
   self@getDamping[] := damping;
   self@getStiffness[] := stiffness;
   self@getTF[] := tf;
   self@getPolynomial[] := polynomial;
   self@getStepResponse[] := unitStepResponse;
   self@getPoles[] := poles;
   self@getBode[] := BodePlot[tf];

   (*----------------------- constructor code -------------------------*)
   s = $s;
   t = $t;
   stiffness = $stiffness;
   damping = $damping;
   mass = $mass;
   update[];

   self
   ];
```

2.1 Create an instance of the class

```
mass = 10; damping = 1; stiffness = .5;
plant = plantClass[stiffness, damping, mass, s, t];
```

2.2 Use it to make a step response plot

```
Plot[plant@getStepResponse[], {t, 0, 50},
 FrameLabel -> {{y[t], None}, {t,
    Row[{"Step response of a plant represented as transfer function ",
       plant@getPolynomial[]}]}}, Frame -> True, PlotRange -> All]
```

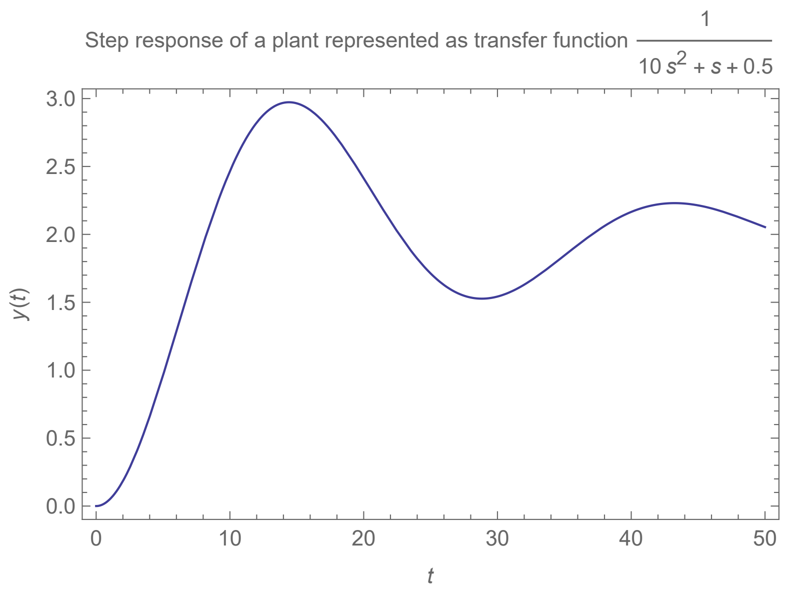


Figure 2: Plot generated from the above command

2.3 Change the plant damping ratio and update the response plot

```
plant@setDamping[5];
Plot[plant@getStepResponse[], {t, 0, 20},
 FrameLabel -> {{y[t], None}, {t,
    Row[{"Step response of a plant represented as transfer function ",
       plant@getPolynomial[]}]}}, Frame -> True, PlotRange -> All]
```

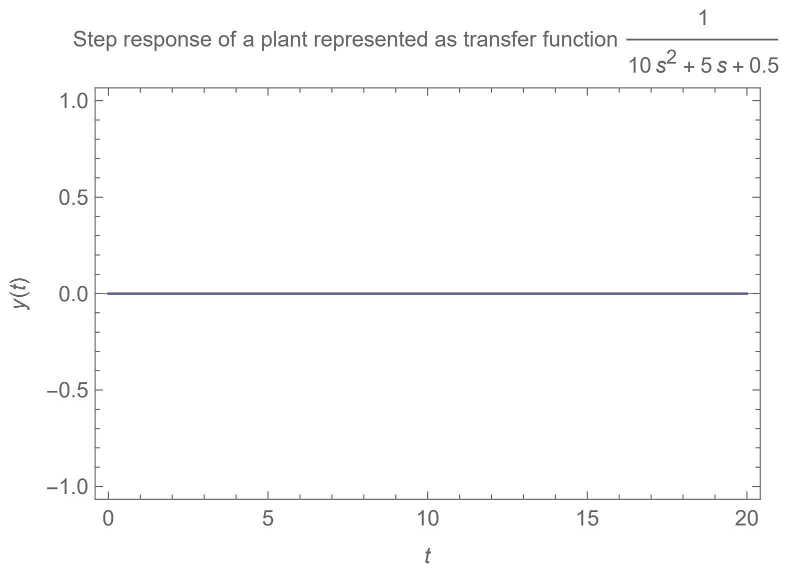
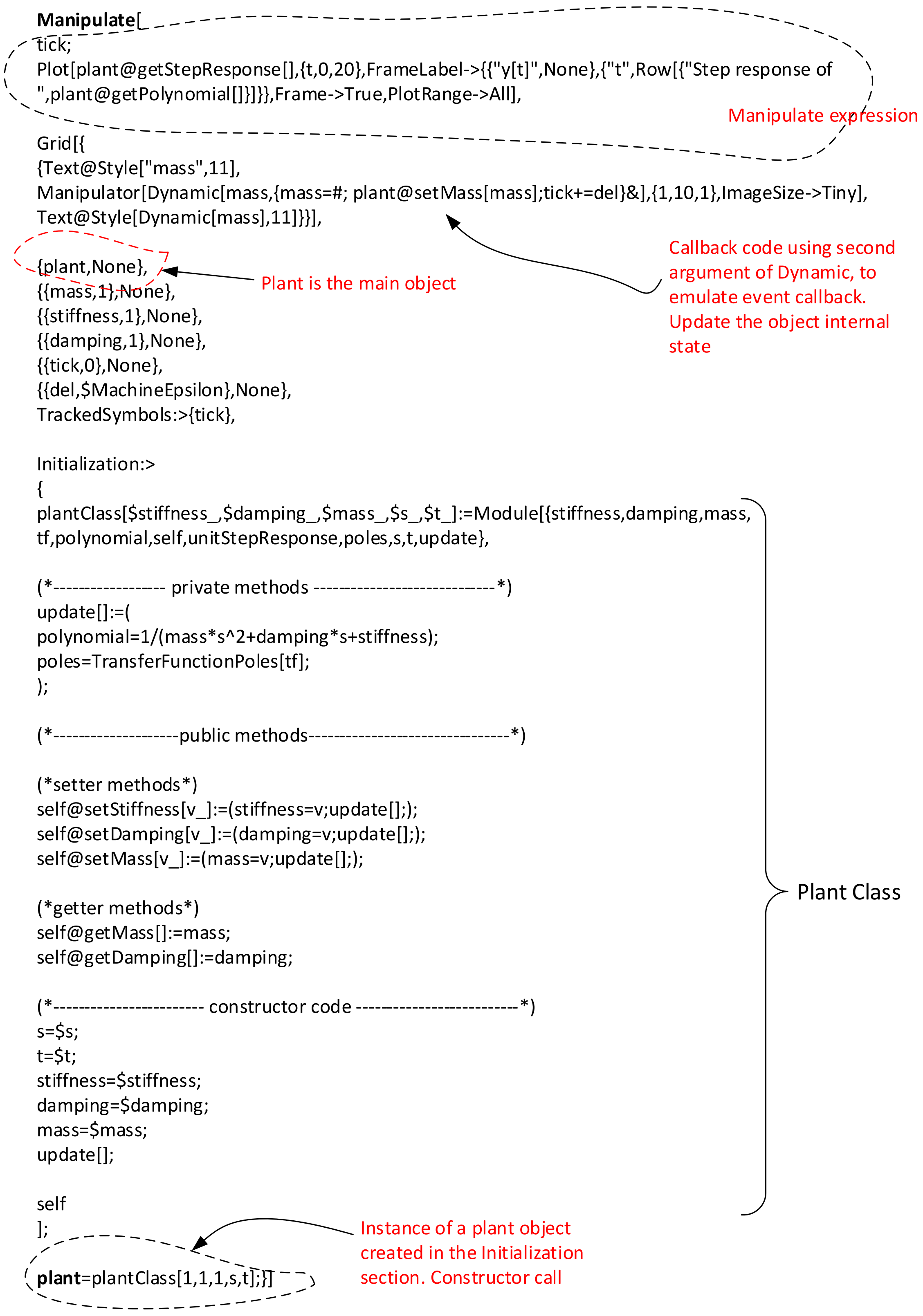


Figure 3: Plot generated from the above command

3 Making a Manipulate to use the above Class to simulate plant response to different parameters

The above Class is now used inside Manipulate. It is important that the object instantiation occur in the Manipulate Initialization section, and after the Class code and not before it.

In this example, the object is the second order plant, and one instance is created in the Manipulate Initialization section. Each time a slider changes, the object internal state is updated using a setter method. Here is a diagram to help illustrate the layout



```
Manipulate[
tick;
Plot[plant@getStepResponse[],{t,0,20},FrameLabel->{{"y[t]",None},{"t",Row[{"Step response of
",plant@getPolynomial[]}]}},Frame->True,PlotRange->All],

Grid[{
{Text@Style["mass",11],
Manipulator[Dynamic[mass,{mass=#; plant@setMass[mass];tick+=del}&],{1,10,1},ImageSize->Tiny],
Text@Style[Dynamic[mass],11]}}],

{plant,None},
{{mass,1},None},
{{stiffness,1},None},
{{damping,1},None},
{{tick,0},None},
{{del,$MachineEpsilon},None},
TrackedSymbols:>{tick},

Initialization:>
{
plantClass[$stiffness_,$damping_,$mass_,$s_,$t_]:=Module[{stiffness,damping,mass,
tf,polynomial,self,unitStepResponse,poles,s,t,update},

(*------------------ private methods ----------------------------*)
update[]:=(
polynomial=1/(mass*s^2+damping*s+stiffness);
poles=TransferFunctionPoles[tf];
);

(*--------------------public methods--------------------------------*)

(*setter methods*)
self@setStiffness[v_]:=(stiffness=v;update[];);
self@setDamping[v_]:=(damping=v;update[];);
self@setMass[v_]:=(mass=v;update[];);

(*getter methods*)
self@getMass[]:=mass;
self@getDamping[]:=damping;

(*------------------------ constructor code --------------------------*)
s=$s;
t=$t;
stiffness=$stiffness;
damping=$damping;
mass=$mass;
update[];

self
];
plant=plantClass[1,1,1,s,t];}]
```

Manipulate expression
Callback code using second argument of Dynamic, to emulate event callback. Update the object internal state
Plant is the main object
Plant Class
Instance of a plant object created in the Initialization section. Constructor call

Figure 4: Using the class inside Manipulate

3.1 Code in Mathematica

```
Manipulate[
 tick;
 Plot[plant@getStepResponse[], {t, 0, 20},
  FrameLabel -> {{"y[t]", None}, {"t",
     Row[{"Step response of ", plant@getPolynomial[]}]}},
  Frame -> True, PlotRange -> All
  ],

 (*controls and event callbacks*)
 Grid[{
   {Text@Style["mass", 11],
    Manipulator[
     Dynamic[mass, {mass = #; plant@setMass[mass];
        tick += del} &], {1, 10, 1}, ImageSize -> Tiny],
    Text@Style[Dynamic[mass], 11]
    },
   {Text@Style["damping", 11],
    Manipulator[
     Dynamic[damping, {damping = #; plant@setDamping[damping];
        tick += del} &], {1, 10, 1}, ImageSize -> Tiny],
    Text@Style[Dynamic[damping], 11]
    },
   {Text@Style["stiffness", 11],
    Manipulator[
     Dynamic[stiffness, {stiffness = #; plant@setStiffness[stiffness];
         tick += del} &], {1, 10, 1}, ImageSize -> Tiny],
    Text@Style[Dynamic[stiffness], 11]
    }
   }],

 {plant, None},
 {{mass, 1}, None},
 {{stiffness, 1}, None},
 {{damping, 1}, None},
 {{tick, 0}, None},
 {{del, $MachineEpsilon}, None},
 TrackedSymbols :> {tick},

 SynchronousUpdating -> False,
ContinuousAction -> False,
SynchronousInitialization -> True,
Initialization :>
 {
  plantClass[$stiffness_, $damping_, $mass_, $s_, $t_] :=
   Module[{stiffness, damping, mass, tf, polynomial, self,
     unitStepResponse, poles, s, t, update},
    SetAttributes[self, HoldAll];
    (*------------------
    private methods -----------------------------*)
    update[] := (
      polynomial = 1/(mass*s^2 + damping*s + stiffness);
      tf = TransferFunctionModel[polynomial, s];
      unitStepResponse =
       Chop@First@OutputResponse[ tf, UnitStep[t], t];
      poles = TransferFunctionPoles[tf];
      );

    (*--------------------public methods--------------------------------*)

    (*setter methods*)
    self@setStiffness[v_] := (stiffness = v; update[];);
    self@setDamping[v_] := (damping = v; update[];);
    self@setMass[v_] := (mass = v; update[];);

    (*getter methods*)
    self@getMass[] := mass;
    self@getDamping[] := damping;
    self@getStiffness[] := stiffness;
    self@getTF[] := tf;
    self@getPolynomial[] := polynomial;
    self@getStepResponse[] := unitStepResponse;
    self@getPoles[] := poles;
    self@getBode[] := BodePlot[tf];

    (*----------------------- constructor code -------------------------*)
    s = $s;
    t = $t;
    stiffness = $stiffness;
    damping = $damping;
    mass = $mass;
    update[];

    self
    ];

  plant = plantClass[1, 1, 1, s, t];
  }
]
```

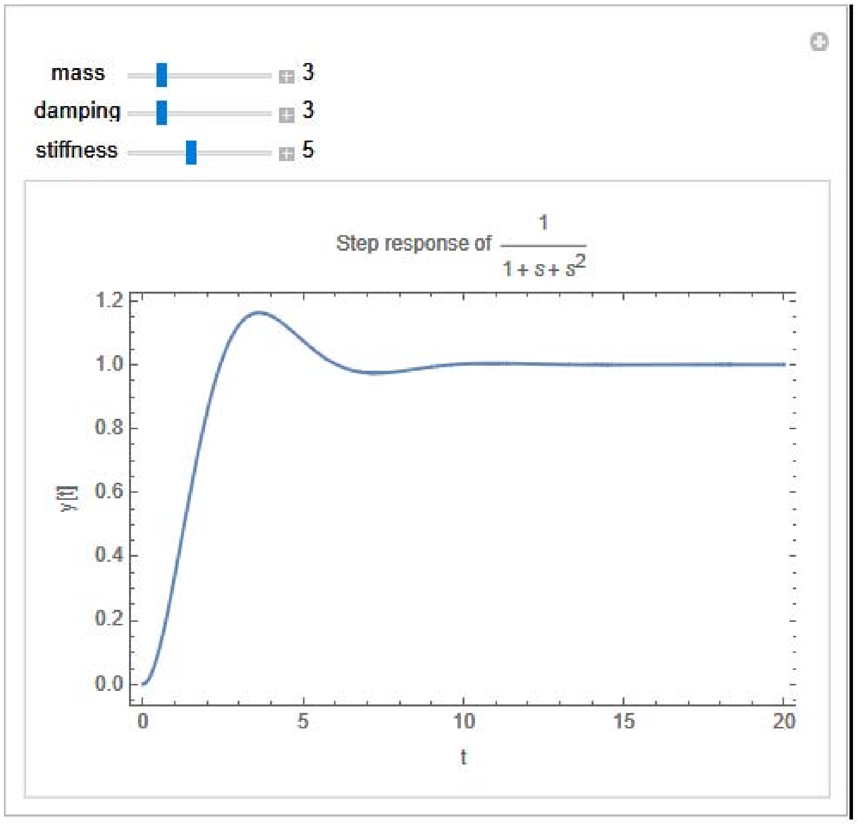

mass 3
damping 3
stiffness 5

Figure 5: Screen shot of the Manipulate

4 Conclusion

It is well known that object based programming help to improve the design of software and managing the complexity of large applications. Mathematica can be used effectively as object based and combined with functional programming, which leads to better overall software. I have used this setup for first time in an actual demonstration for the simulation of control system successfully, and I have found that it helped better organize my demonstration code and shortened the development time. The demonstration can be downloaded from `https://demonstrations.wolfram.com/SimulationOfFeedbackControlSystemWithControllerAndSecondOrde/` or from `https://12000.org/my_notes/mma_demos/PID/index.htm`

4.1 References

1. `http://mathematica.stackexchange.com/questions/586/how-can-you-give-a-module-a-context-and-have-its-local-variables-and-modules-bel`