Callbacks and Events

Chris Rackauckas

February 25, 2019

In working with a differential equation, our system will evolve through many states. Particular states of the system may be of interest to us, and we say that an ***"event"*** is triggered when our system reaches these states. For example, events may include the moment when our system reaches a particular temperature or velocity. We ***handle*** these events with ***callbacks***, which tell us what to do once an event has been triggered.

These callbacks allow for a lot more than event handling, however. For example, we can use callbacks to achieve high-level behavior like exactly preserve conservation laws and save the trace of a matrix at pre-defined time points. This extra functionality allows us to use the callback system as a modding system for the DiffEq ecosystem's solvers.

This tutorial is an introduction to the callback and event handling system in DifferentialEquations.jl, documented in the Event Handling and Callback Functions page of the documentation. We will also introduce you to some of the most widely used callbacks in the Callback Library, which is a library of pre-built mods.

0.1 Events and Continuous Callbacks

Event handling is done through continuous callbacks. Callbacks take a function, `condition`, which triggers an `affect!` when `condition == 0`. These callbacks are called "continuous" because they will utilize rootfinding on the interpolation to find the "exact" time point at which the condition takes place and apply the `affect!` at that time point.

Let's use a bouncing ball as a simple system to explain events and callbacks. Let's take Newton's model of a ball falling towards the Earth's surface via a gravitational constant `g`. In this case, the velocity is changing via `-g`, and position is changing via the velocity. Therefore we receive the system of ODEs:

```
using DifferentialEquations, ParameterizedFunctions
ball! = @ode_def BallBounce begin
  dy =  v
  dv = -g
end g
```

We want the callback to trigger when `y=0` since that's when the ball will hit the Earth's surface (our event). We do this with the condition:

```
function condition(u,t,integrator)
```

```
  u[1]
end
```

```
condition (generic function with 1 method)
```

Recall that the `condition` will trigger when it evaluates to zero, and here it will evaluate to zero when `u[1] == 0`, which occurs when `v == 0`. *Now we have to say what we want the callback to do.* Callbacks make use of the Integrator Interface. Instead of giving a full description, a quick and usable rundown is:

- Values are strored in `integrator.u`
- Times are stored in `integrator.t`
- The parameters are stored in `integrator.p`
- `integrator(t)` performs an interpolation in the current interval between `integrator.tprev` and `integrator.t` (and allows extrapolation)
- User-defined options (tolerances, etc.) are stored in `integrator.opts`
- `integrator.sol` is the current solution object. Note that `integrator.sol.prob` is the current problem

While there's a lot more on the integrator interface page, that's a working knowledge of what's there.

What we want to do with our `affect!` is to "make the ball bounce". Mathematically speaking, the ball bounces when the sign of the velocity flips. As an added behavior, let's also use a small friction constant to dampen the ball's velocity. This way only a percentage of the velocity will be retained when the event is triggered and the callback is used. We'll define this behavior in the `affect!` function:

```
function affect!(integrator)
    integrator.u[2] = -integrator.p[2] * integrator.u[2]
end
```

```
affect! (generic function with 1 method)
```

`integrator.u[2]` is the second value of our model, which is `v` or velocity, and `integrator.p[2]`, is our friction coefficient.

Therefore `affect!` can be read as follows: `affect!` will take the current value of velocity, and multiply it `-1` multiplied by our friction coefficient. Therefore the ball will change direction and its velocity will dampen when `affect!` is called.

Now let's build the `ContinuousCallback`:

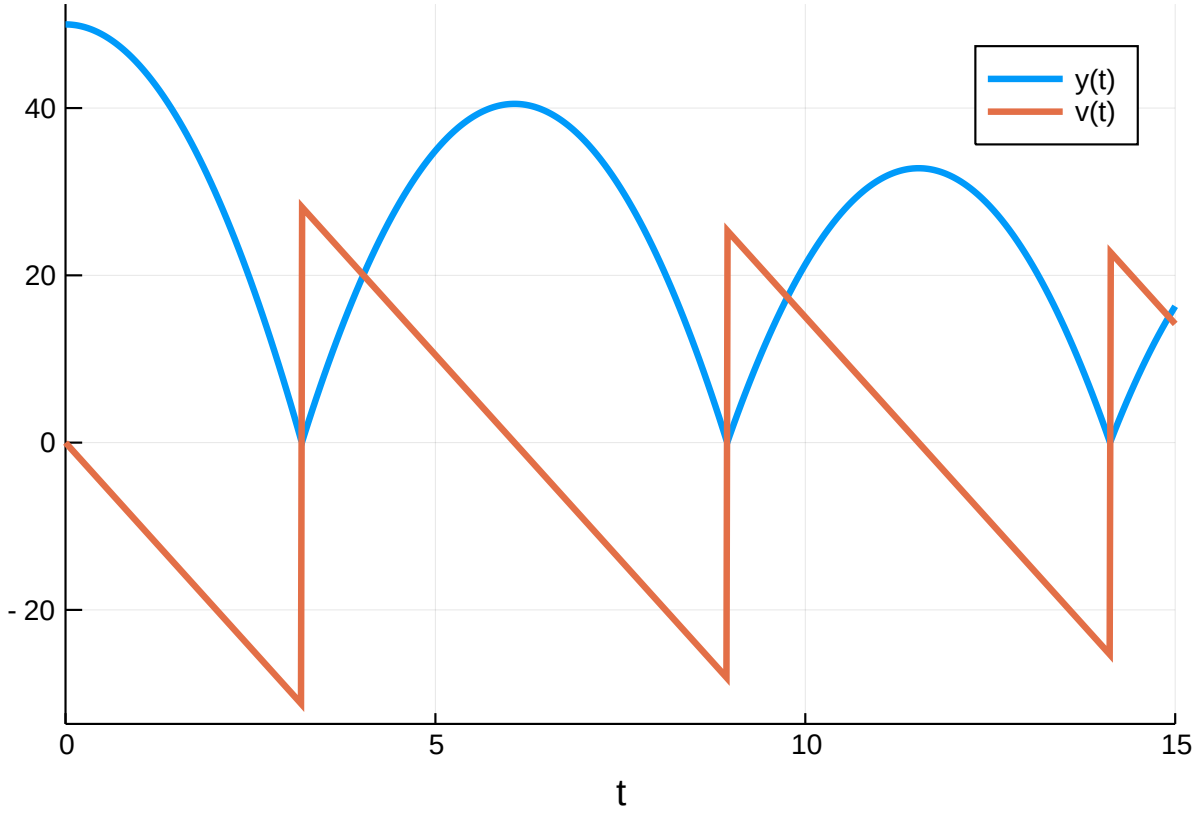
```
bounce_cb = ContinuousCallback(condition,affect!)
```

Now let's make an `ODEProblem` which has our callback:

```
u0 = [50.0,0.0]
tspan = (0.0,15.0)
p = (9.8,0.9)
prob = ODEProblem(ball!,u0,tspan,p,callback=bounce_cb)
```

Notice that we chose a friction constant of `0.9`. Now we can solve the problem and plot the solution as we normally would:

```
sol = solve(prob,Tsit5())
using Plots; gr()
plot(sol)
```



and tada, the ball bounces! Notice that the `ContinuousCallback` is using the interpolation to apply the effect "exactly" when `v == 0`. This is crucial for model correctness, and thus when this property is needed a `ContinuousCallback` should be used.

Exercise 1 In our example we used a constant coefficient of friction, but if we are bouncing the ball in the same place we may be smoothing the surface (say, squishing the grass), causing there to be less friction after each bounce. In this more advanced model, we want the friction coefficient at the next bounce to be `sqrt(friction)` from the previous bounce (since `friction < 1`, `sqrt(friction) > friction` and `sqrt(friction) < 1`).

Hint: there are many ways to implement this. One way to do it is to make `p` a `Vector` and mutate the friction coefficient in the `affect!`.

0.2 Discrete Callbacks

A discrete callback checks a `condition` after every integration step and, if true, it will apply an `affect!`. For example, let's say that at time `t=2` we want to include that a kid kicked the ball, adding 20 to the current velocity. This kind of situation, where we want to add a specific behavior which does not require rootfinding, is a good candidate for a `DiscreteCallback`. In this case, the `condition` is a boolean for whether to apply the `affect!`, so:

```
function condition_kick(u,t,integrator)
    t == 2
end
```

```
condition_kick (generic function with 1 method)
```

We want the kick to occur at `t=2`, so we check for that time point. When we are at this time point, we want to do:

```
function affect_kick!(integrator)
    integrator.u[2] += 50
end
```

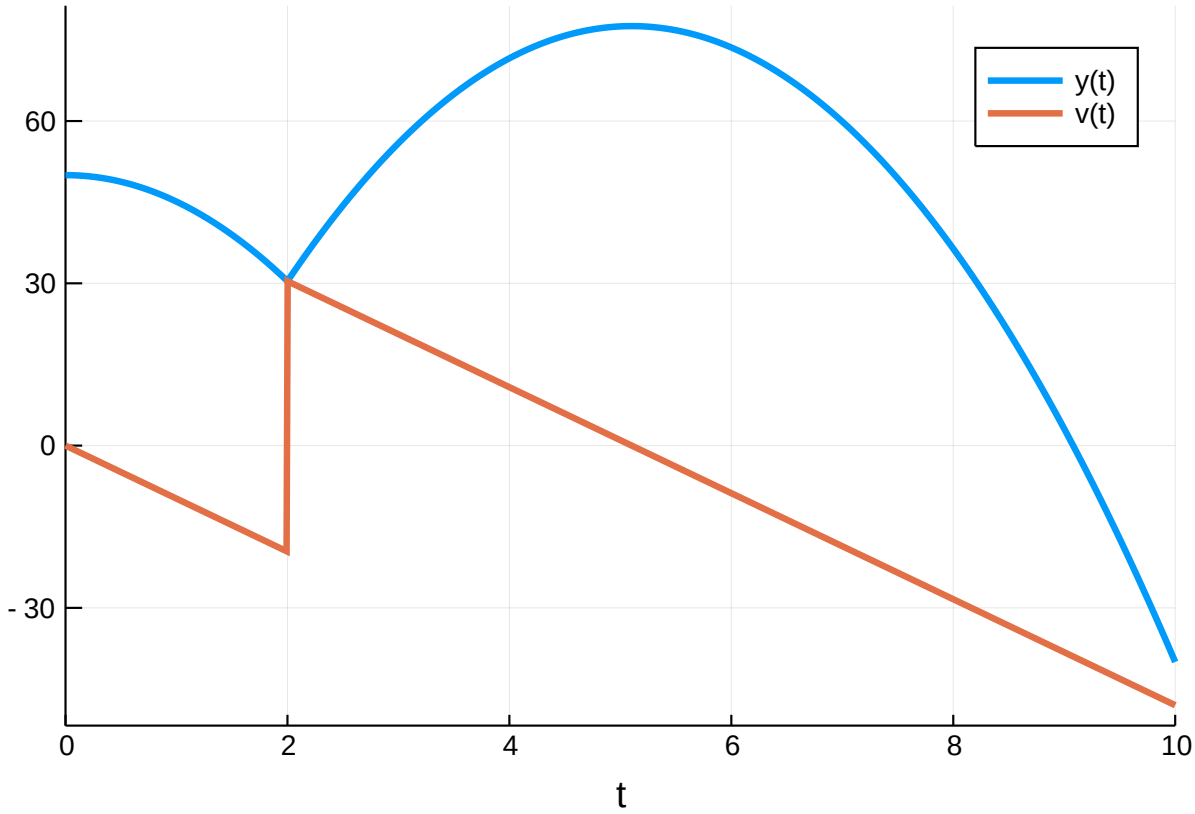
```
affect_kick! (generic function with 1 method)
```

Now we build the problem as before:

```
kick_cb = DiscreteCallback(condition_kick,affect_kick!)
u0 = [50.0,0.0]
tspan = (0.0,10.0)
p = (9.8,0.9)
prob = ODEProblem(ball!,u0,tspan,p,callback=kick_cb)
```

Note that, since we are requiring our effect at exactly the time `t=2`, we need to tell the integration scheme to step at exactly `t=2` to apply this callback. This is done via the option `tstops`, which is like `saveat` but means "stop at these values".

```
sol = solve(prob,Tsit5(),tstops=[2.0])
plot(sol)
```



Note that this example could've been done with a `ContinuousCallback` by checking the condition `t-2`.

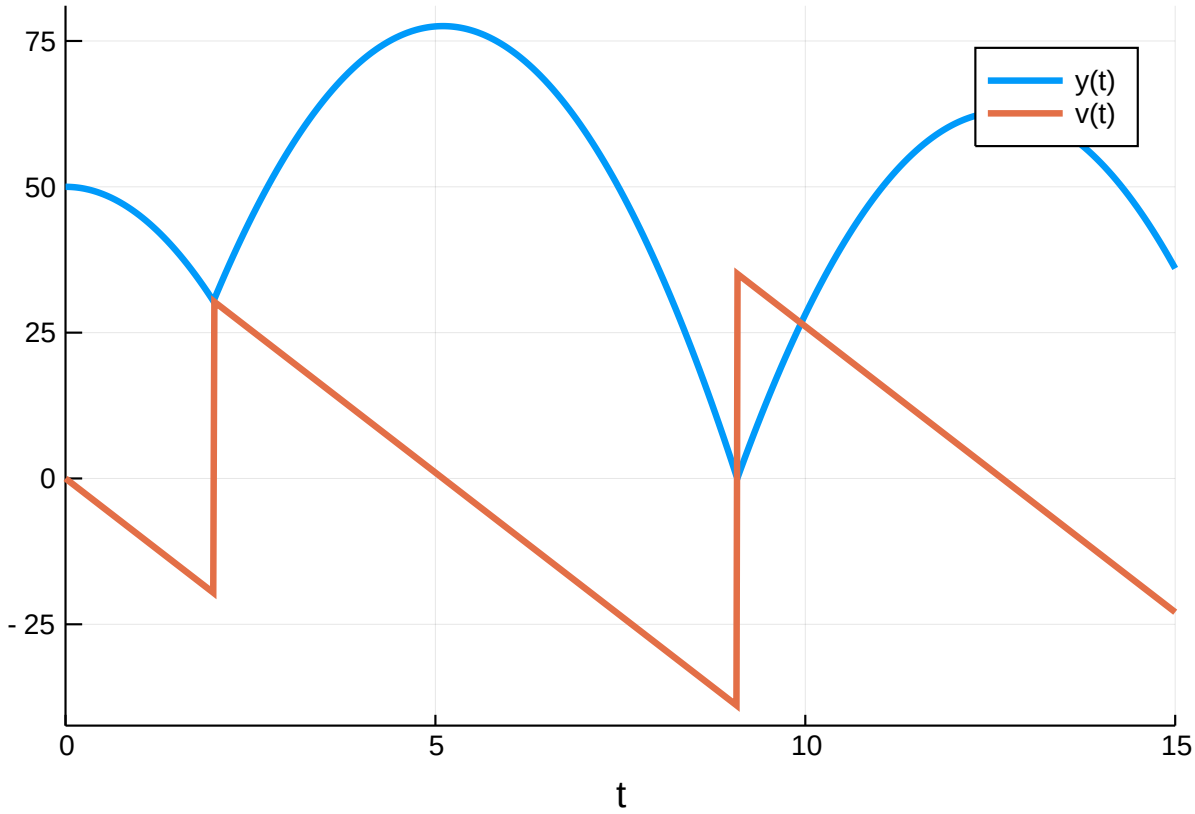
0.3 Merging Callbacks with Callback Sets

In some cases you may want to merge callbacks to build up more complex behavior. In our previous result, notice that the model is unphysical because the ball goes below zero! What we really need to do is add the bounce callback together with the kick. This can be achieved through the `CallbackSet`.

```
cb = CallbackSet(bounce_cb,kick_cb)
```

A `CallbackSet` merges their behavior together. The logic is as follows. In a given interval, if there are multiple continuous callbacks that would trigger, only the one that triggers at the earliest time is used. The time is pulled back to where that continuous callback is triggered, and then the `DiscreteCallbacks` in the callback set are called in order.

```
u0 = [50.0,0.0]
tspan = (0.0,15.0)
p = (9.8,0.9)
prob = ODEProblem(ball!,u0,tspan,p,callback=cb)
sol = solve(prob,Tsit5(),tstops=[2.0])
plot(sol)
```



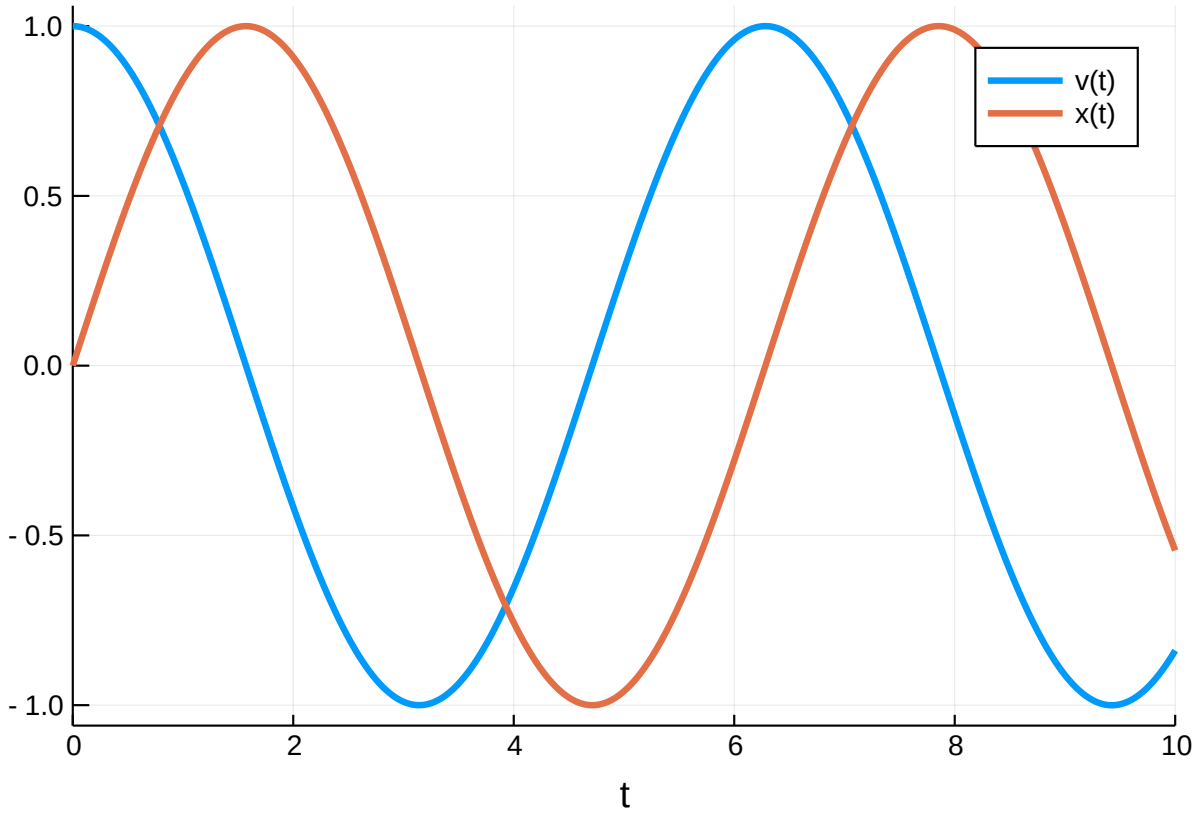
Notice that we have now merged the behaviors. We can then nest this as deep as we like.

Exercise 2 Add to the model a linear wind with resistance that changes the acceleration to `-g + k*v` after `t=10`. Do so by adding another parameter and allowing it to be zero until a specific time point where a third callback triggers the change.

0.4 Integration Termination and Directional Handling

Let's look at another model now: the model of the Harmonic Oscillator. We can write this as:

```
u0 = [1.,0.]
harmonic! = @ode_def HarmonicOscillator begin
   dv = -x
   dx = v
end
tspan = (0.0,10.0)
prob = ODEProblem(harmonic!,u0,tspan)
sol = solve(prob)
plot(sol)
```



Let's instead stop the integration when a condition is met. From the Integrator Interface stepping controls we see that `terminate!(integrator)` will cause the integration to end. So our new `affect!` is simply:

```
function terminate_affect!(integrator)
    terminate!(integrator)
end
```

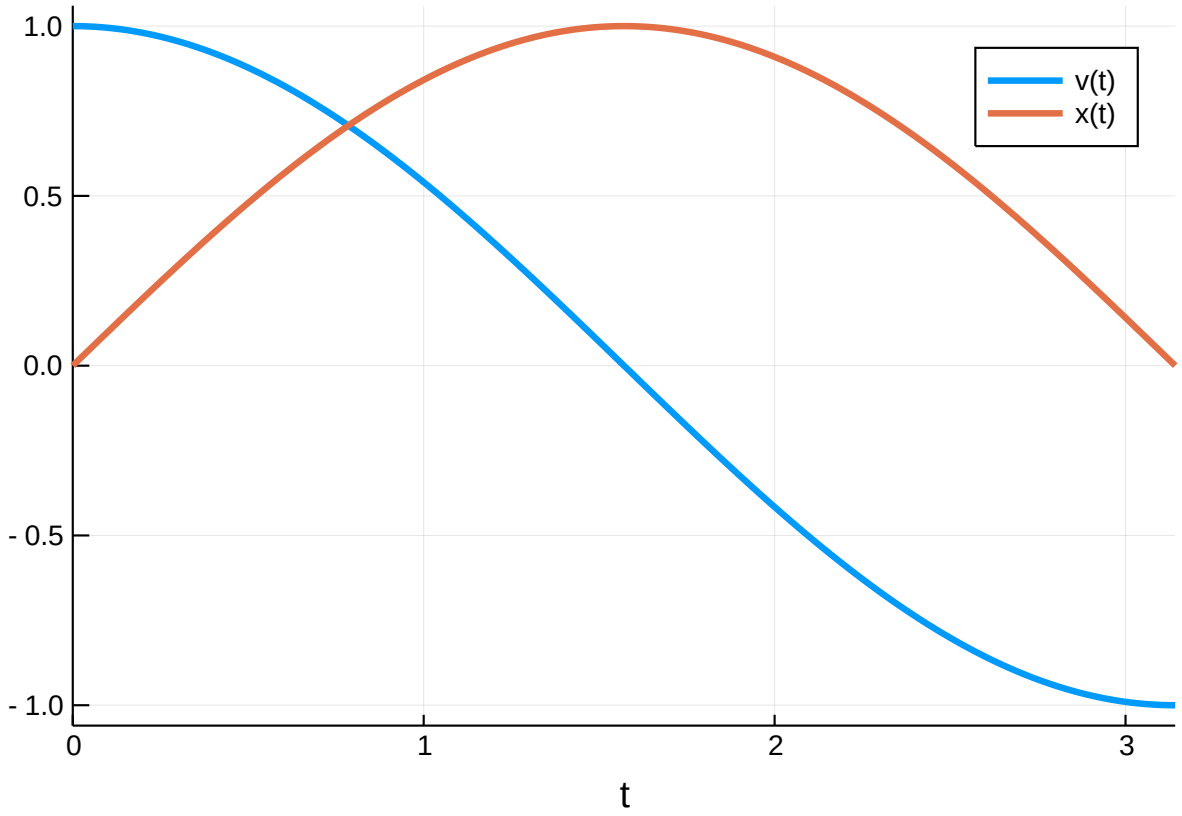
```
terminate_affect! (generic function with 1 method)
```

Let's first stop the integration when the particle moves back to `x=0`. This means we want to use the condition:

```
function terminate_condition(u,t,integrator)
    u[2]
end
terminate_cb = ContinuousCallback(terminate_condition,terminate_affect!)
```

Note that instead of adding callbacks to the problem, we can also add them to the `solve` command. This will automatically form a `CallbackSet` with any problem-related callbacks and naturally allows you to distinguish between model features and integration controls.

```
sol = solve(prob,callback=terminate_cb)
plot(sol)
```



Notice that the harmonic oscilator's true solution here is `sin` and `cosine`, and thus we would expect this return to zero to happen at `t=`π:

```
sol.t[end]
```

```
3.1415902498226123
```

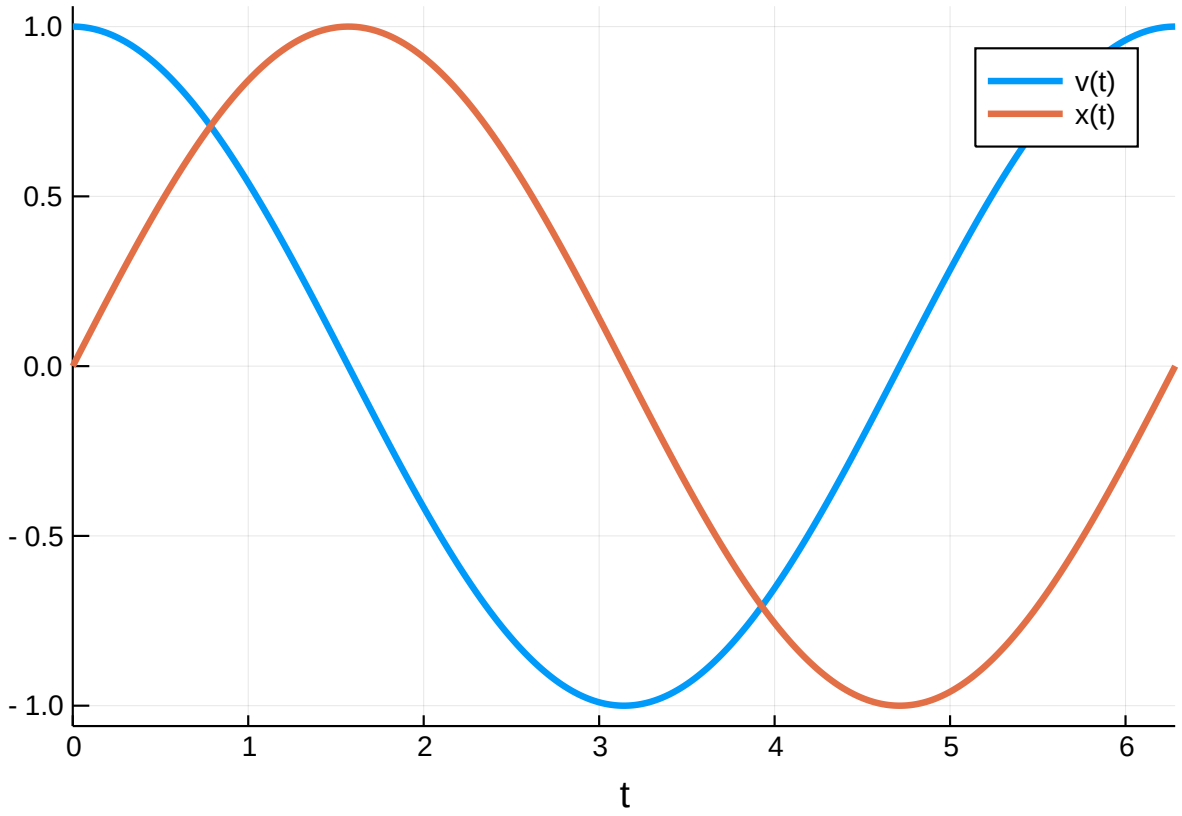
This is one way to approximate π! Lower tolerances and arbitrary precision numbers can make this more exact, but let's not look at that. Instead, what if we wanted to halt the integration after exactly one cycle? To do so we would need to ignore the first zero-crossing. Luckily in these types of scenarios there's usually a structure to the problem that can be exploited. Here, we only want to trigger the `affect!` when crossing from positive to negative, and not when crossing from negative to positive. In other words, we want our `affect!` to only occur on upcrossings.

If the `ContinuousCallback` constructor is given a single `affect!`, it will occur on both upcrossings and downcrossings. If there are two `affect!`s given, then the first is for upcrossings and the second is for downcrossings. An `affect!` can be ignored by using `nothing`. Together, the "upcrossing-only" version of the effect means that the first `affect!` is what we defined above and the second is `nothing`. Therefore we want:

```
terminate_upcrossing_cb =
    ContinuousCallback(terminate_condition,terminate_affect!,nothing)
```

Which gives us:

```
sol = solve(prob,callback=terminate_upcrossing_cb)
plot(sol)
```



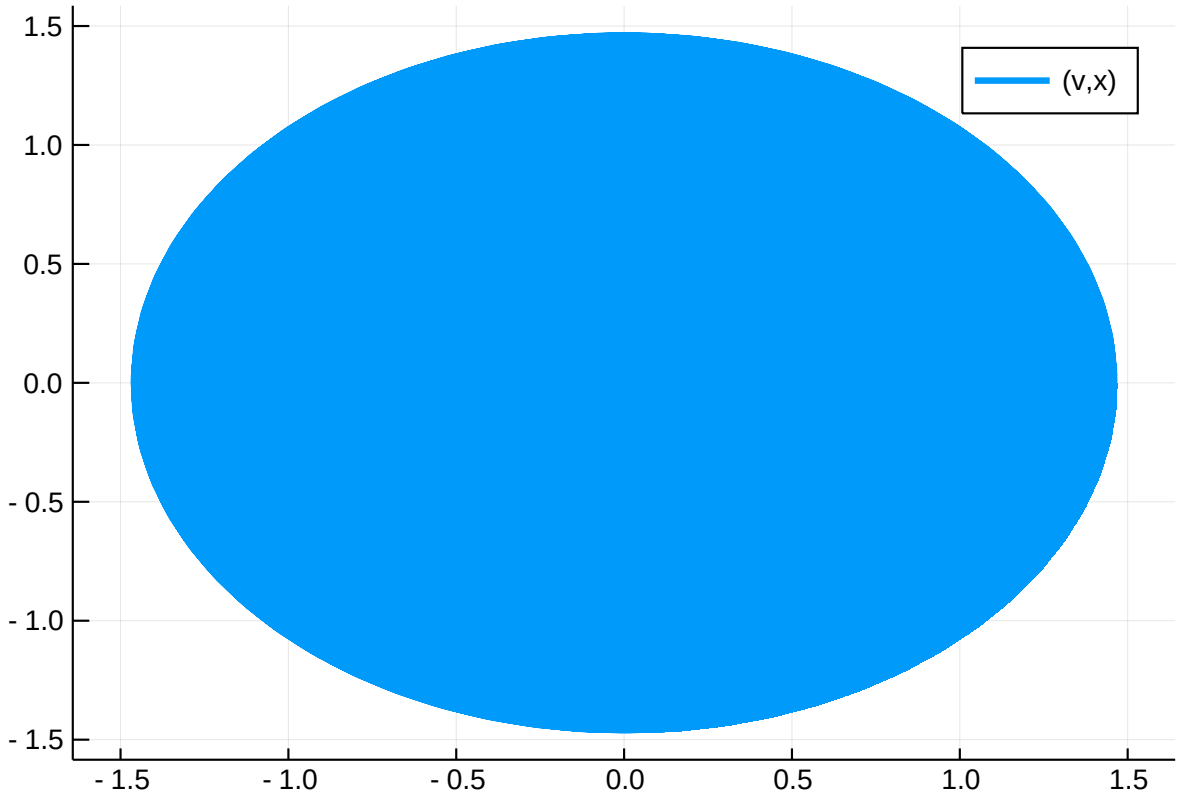
0.5 Callback Library

As you can see, callbacks can be very useful and through `CallbackSets` we can merge together various behaviors. Because of this utility, there is a library of pre-built callbacks known as the Callback Library. We will walk through a few examples where these callbacks can come in handy.

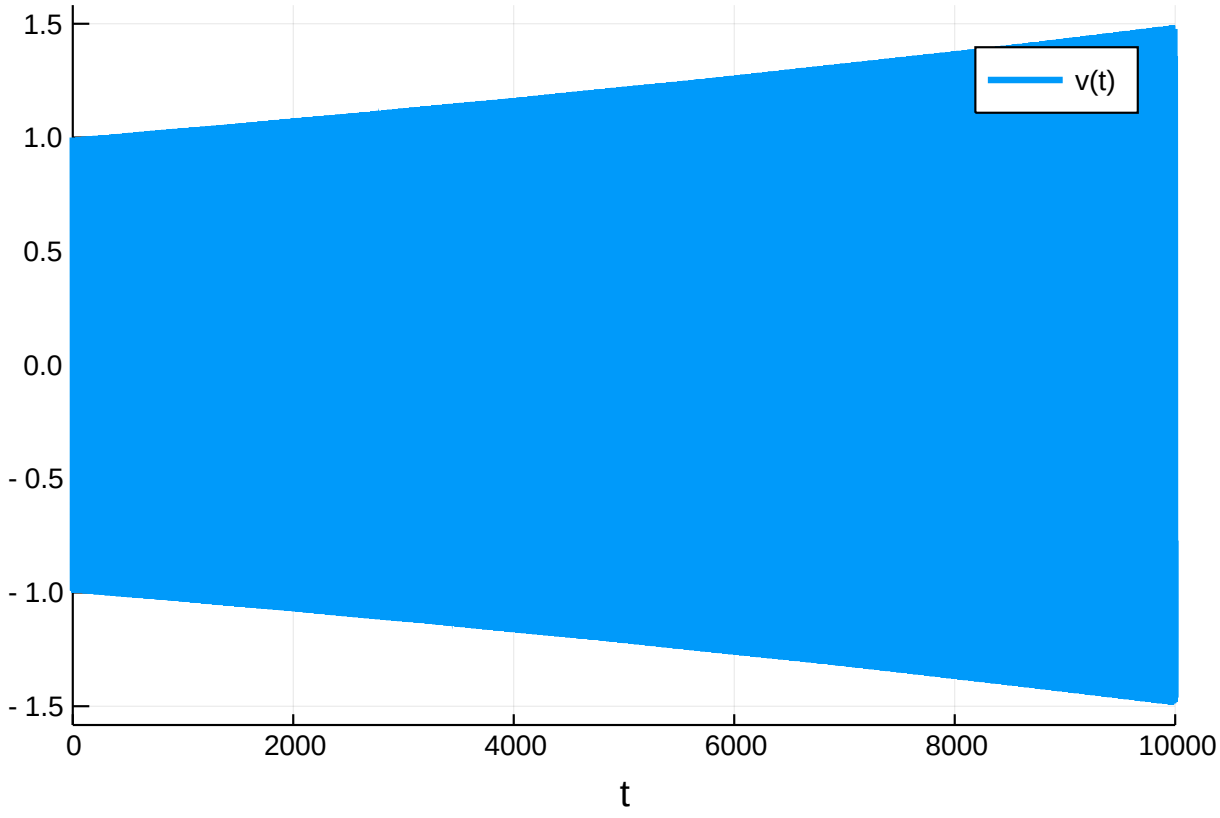
0.5.1 Manifold Projection

One callback is the manifold projection callback. Essentially, you can define any manifold `g(sol)=0` which the solution must live on, and cause the integration to project to that manifold after every step. As an example, let's see what happens if we naively run the harmonic oscillator for a long time:

```
tspan = (0.0,10000.0)
prob = ODEProblem(harmonic!,u0,tspan)
sol = solve(prob)
gr(fmt=:png) # Make it a PNG instead of an SVG since there's a lot of points!
plot(sol,vars=(1,2))
```

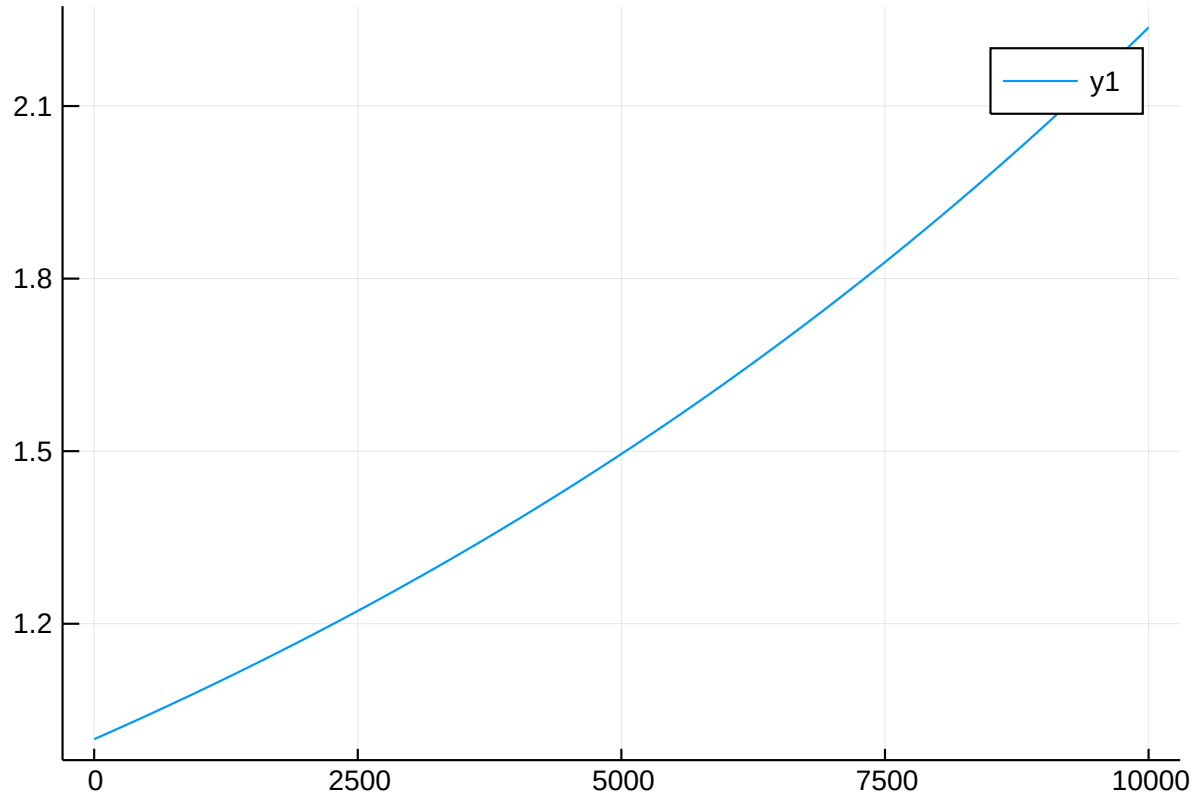



```
plot(sol,vars=(0,1),denseplot=false)
```



Notice that what's going on is that the numerical solution is drifting from the true solution over this long time scale. This is because the integrator is not conserving energy.

```
plot(sol.t,[u[2]^2 + u[1]^2 for u in sol.u]) # Energy ~ x^2 + v^2
```



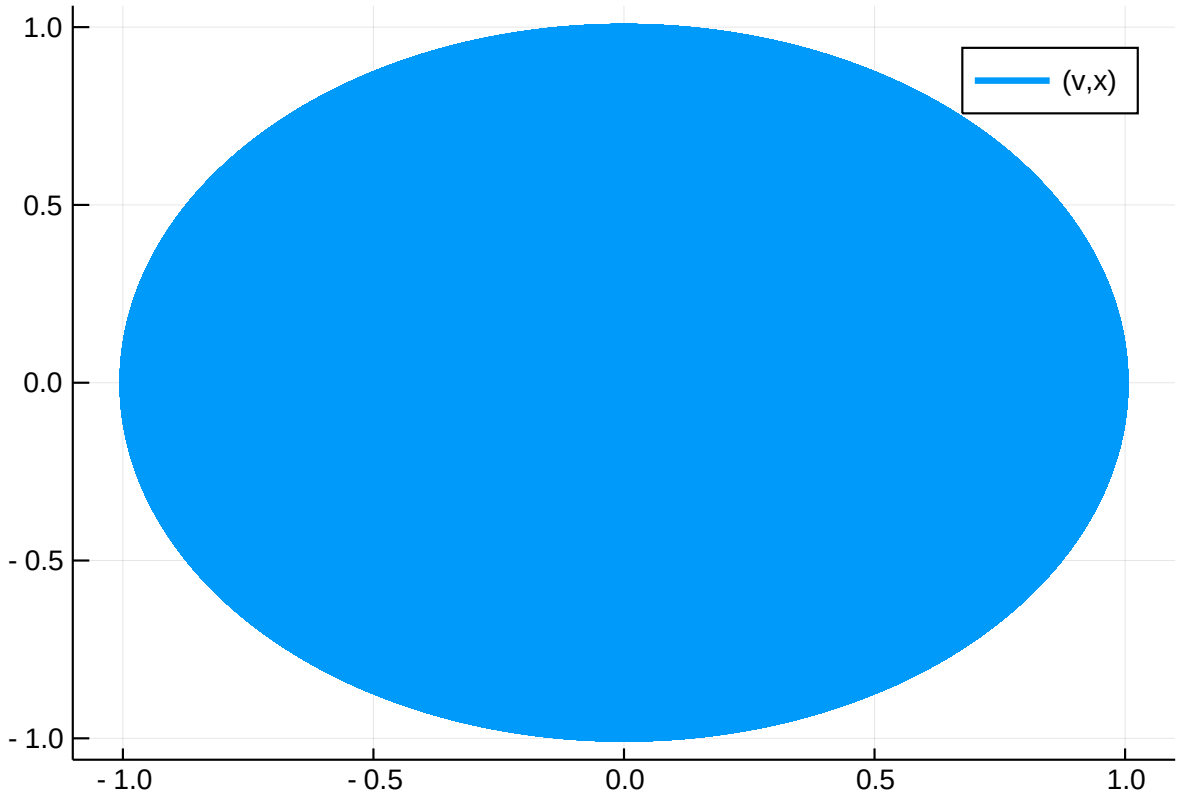
Some integration techniques like symplectic integrators are designed to mitigate this issue, but instead let's tackle the problem by enforcing conservation of energy. To do so, we define our manifold as the one where energy equals 1 (since that holds in the initial condition), that is:

```
function g(resid,u,p,t)
  resid[1] = u[2]^2 + u[1]^2 - 1
  resid[2] = 0
end
```

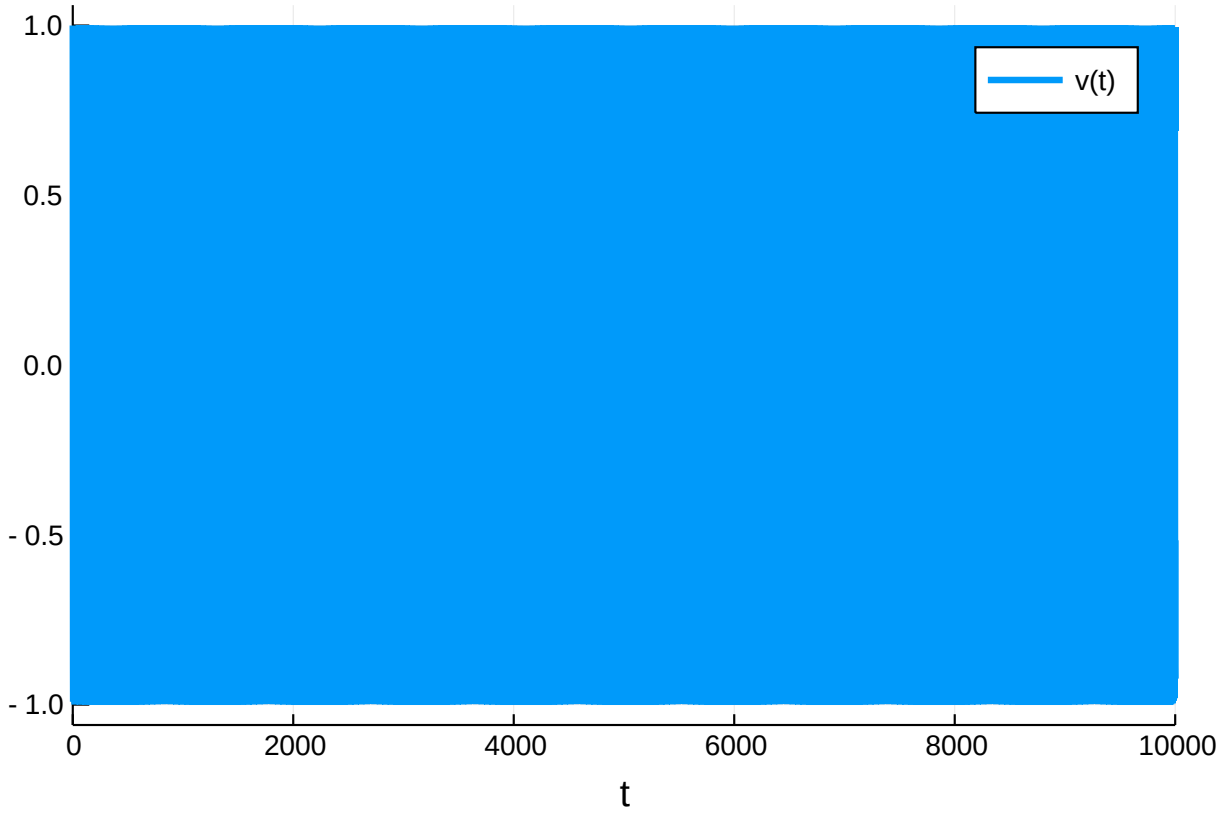
```
g (generic function with 1 method)
```

Here the residual measures how far from our desired energy we are, and the number of conditions matches the size of our system (we ignored the second one by making the residual 0). Thus we define a `ManifoldProjection` callback and add that to the solver:

```
cb = ManifoldProjection(g)
sol = solve(prob,callback=cb)
plot(sol,vars=(1,2))
```




```
plot(sol,vars=(0,1),denseplot=false)
```



Now we have "perfect" energy conservation, where if it's ever violated too much the solution will get projected back to `energy=1`.

```
u1,u2 = sol[500]
u2^2 + u1^2
```

```
1.0000425845647514
```

While choosing different integration schemes and using lower tolerances can achieve this effect as well, this can be a nice way to enforce physical constraints and is thus used in many disciplines like molecular dynamics. Another such domain constraining callback is the `PositiveCallback()` which can be used to enforce positivity of the variables.

0.5.2 SavingCallback

The `SavingCallback` can be used to allow for special saving behavior. Let's take a linear ODE define on a system of 1000x1000 matrices:

```
prob = ODEProblem((du,u,p,t)->du.=u,rand(1000,1000),(0.0,1.0))
```

In fields like quantum mechanics you may only want to know specific properties of the solution such as the trace or the norm of the matrix. Saving all of the 1000x1000 matrices can be a costly way to get this information! Instead, we can use the `SavingCallback` to save the `trace` and `norm` at specified times. To do so, we first define our `SavedValues` cache. Our time is in terms of `Float64`, and we want to save tuples of `Float64`s (one for the `trace` and one for the `norm`), and thus we generate the cache as:

```
saved_values = SavedValues(Float64, Tuple{Float64,Float64})
```

Now we define the `SavingCallback` by giving it a function of `(u,p,t,integrator)` that returns the values to save, and the cache:

```
using LinearAlgebra
cb = SavingCallback((u,t,integrator)->(tr(u),norm(u)), saved_values)
```

Here we take `u` and save `(tr(u),norm(u))`. When we solve with this callback:

```
sol = solve(prob, Tsit5(), callback=cb, save_everystep=false, save_start=false,
    save_end = false) # Turn off normal
    saving
```

Our values are stored in our `saved_values` variable:

```
saved_values.t
```

```
5-element Array{Float64,1}:
 0.0
```

```
 0.10012919123099687
 0.3483908642670527
 0.6837371824388825
 1.0
```

```
saved_values.saveval
```

```
5-element Array{Tuple{Float64,Float64},1}:
 (507.3097845885668, 577.3688967213816)
 (560.736457948385, 638.1737547989151)
 (718.7493551165406, 818.0081180957256)
 (1005.1167536168117, 1143.9226459678168)
 (1379.0109082387173, 1569.4513112974228)
```

By default this happened only at the solver's steps. But the `SavingCallback` has similar controls as the integrator. For example, if we want to save at every `0.1` seconds, we do can so using `saveat`:

```
saved_values = SavedValues(Float64, Tuple{Float64,Float64}) # New cache
cb = SavingCallback((u,t,integrator)->(tr(u),norm(u)), saved_values, saveat =
    0.0:0.1:1.0)
sol = solve(prob, Tsit5(), callback=cb, save_everystep=false, save_start=false,
    save_end = false) # Turn off normal
    saving
```

```
saved_values.t
```

```
11-element Array{Float64,1}:
 0.0
 0.1
 0.2
 0.3
 0.4
 0.5
 0.6
 0.7
 0.8
 0.9
 1.0
```

```
saved_values.saveval
```

```
11-element Array{Tuple{Float64,Float64},1}:
 (507.3097845885668, 577.3688967213816)
 (560.6640203943352, 638.0913136713561)
 (619.6295922797631, 705.1999881307798)
 (684.7965396621105, 779.3664435311259)
 (756.8174035577464, 861.3333363283327)
```

```
(836.4123896651672, 951.9203321038325)
(924.3783371615668, 1052.0343129454154)
(1021.596451876869, 1162.6781785669505)
(1129.0387788881305, 1284.958212763371)
(1247.7803234705375, 1420.0978782561997)
(1379.0109082387173, 1569.4513112974228)
```

Exercise 3 Go back to the Harmonic oscillator. Use the `SavingCallback` to save an array for the energy over time, and do this both with and without the `ManifoldProjection`. Plot the results to see the difference the projection makes.

0.6 Appendix

```
using DiffEqTutorials
DiffEqTutorials.tutorial_footer(WEAVE_ARGS[:folder],WEAVE_ARGS[:file])
```

```
These benchmarks are part of the DiffEqTutorials.jl repository, found at:

https://github.com/JuliaDiffEq/DiffEqTutorials.jl

To locally run this tutorial, do the following commands:

using DiffEqTutorials
DiffEqTutorials.weave_file("introduction","callbacks_and_events.jmd")

Computer Information:

Julia Version 1.1.0
Commit 80516ca202 (2019-01-21 21:24 UTC)
Platform Info:
  OS: Windows (x86_64-w64-mingw32)
  CPU: Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz
  WORD_SIZE: 64
  LIBM: libopenlibm
  LLVM: libLLVM-6.0.1 (ORCJIT, skylake)
Environment:
  JULIA_EDITOR = "C:\Users\accou\AppData\Local\atom\app-1.34.0\atom.exe" -a
  JULIA_NUM_THREADS = 6

Package Information:

    Status `C:\Users\accou\.julia\environments\v1.1\Project.toml`
  [7e558dbc] ArbNumerics v0.3.6
  [c52e3926] Atom v0.7.14
  [6e4b80f9] BenchmarkTools v0.4.2
  [336ed68f] CSV v0.4.3
  [3895d2a7] CUDAapi v0.5.4
  [be33ccc6] CUDAnative v1.0.1
  [3a865a2d] CuArrays v0.9.1
  [a93c6f00] DataFrames v0.17.1
  [55939f99] DecFP v0.4.8
  [abce61dc] Decimals v0.4.0
  [39dd38d3] Dierckx v0.4.1
  [459566f4] DiffEqCallbacks v2.5.2
```

```
  [f3b72e0c] DiffEqDevTools v2.6.1
  [aae7a2af] DiffEqFlux v0.2.0
  [c894b116] DiffEqJump v6.1.0+ [`C:\Users\accou\.julia\dev\DiffEqJump`]
  [1130ab10] DiffEqParamEstim v1.6.0+ [`C:\Users\accou\.julia\dev\DiffEqPar
amEstim`]
  [055956cb] DiffEqPhysics v3.1.0
  [225cb15b] DiffEqTutorials v0.0.0 [`C:\Users\accou\.julia\external\DiffEq
Tutorials.jl`]
  [0c46a032] DifferentialEquations v6.3.0
  [497a8b3b] DoubleFloats v0.7.5
  [587475ba] Flux v0.7.3
  [f6369f11] ForwardDiff v0.10.3+ [`C:\Users\accou\.julia\dev\ForwardDiff`]
  [28b8d3ca] GR v0.38.1
  [7073ff75] IJulia v1.17.0
  [c601a237] Interact v0.9.1
  [b6b21f68] Ipopt v0.5.4
  [4076af6c] JuMP v0.19.0
  [e5e0dc1b] Juno v0.5.4
  [eff96d63] Measurements v2.0.0
  [76087f3c] NLopt v0.5.1
  [429524aa] Optim v0.17.2
  [1dea7af3] OrdinaryDiffEq v5.2.1+ [`C:\Users\accou\.julia\dev\OrdinaryDif
fEq`]
  [65888b18] ParameterizedFunctions v4.1.1
  [91a5bcdd] Plots v0.23.0
  [71ad9d73] PuMaS v0.0.0 [`C:\Users\accou\.julia\dev\PuMaS`]
  [d330b81b] PyPlot v2.7.0
  [731186ca] RecursiveArrayTools v0.20.0
  [90137ffa] StaticArrays v0.10.2
  [789caeaf] StochasticDiffEq v6.1.1+ [`C:\Users\accou\.julia\dev\Stochasti
cDiffEq`]
  [c3572dad] Sundials v3.0.0
  [1986cc42] Unitful v0.14.0
  [2a06ce6d] UnitfulPlots v0.0.0 #master (https://github.com/ajkeller34/Uni
tfulPlots.jl)
  [44d3d7a6] Weave v0.7.1 [`C:\Users\accou\.julia\dev\Weave`]
```