Julia Tutorial

Author: Hasan Poonawala

Contents

1 Introduction

The `Julia` programming language is meant to be an open-source and efficient way to implement computing algorithms. You may start by visiting this homepage that mentions advantages. For robotics, a specific advantage is the work by groups at MIT and elsewhere that implements robotics-relevant algorithms as `Julia` packages. A description of these packages can be found on the Julia Robotics webpage.

For our purposes, using `Julia` involves creating `Julia` code that imports a variety of packages and combines them to achieve some goal. Using the right packages and understanding how to combine them depends on understanding the theory behind the corresponding implementations. Below, we briefly mention installation and first runs, and then get to the part about adding packages.

1.1 Online Course

See MIT's Fall 2020 Course on `Julia`

2 Installing Julia

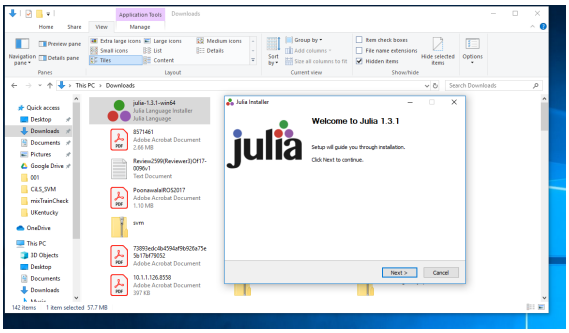
Visit this page for platform-specific instructions.

2.1 MacOS

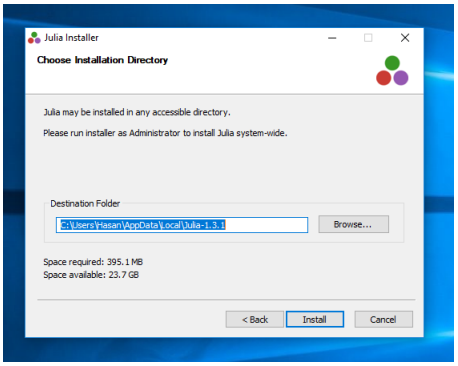
The installation includes downloading the `.dmg` file, double-clicking it, and then moving the `Julia` image to the `Applications` folder. I directly edited the file `/etc/paths` to add the `Julia` binaries folder to the path.

2.2 Windows

The installation on Windows is far smoother than I expected.



(a) Double-click to run installation file, then click on 'Next'



(b) Copy and store the installation directory that you choose, even if you use the default. Click 'Install'.

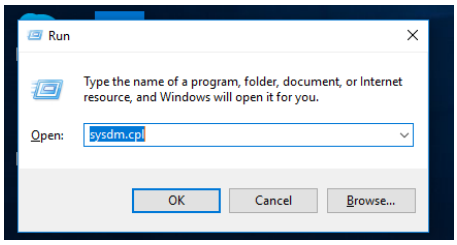
Figure 1: Installing Julia on Windows.

And you're done, sort of. You must navigate to a specific folder to run `Julia`, which turns out to be the location you saved with `\bin` appended to it:

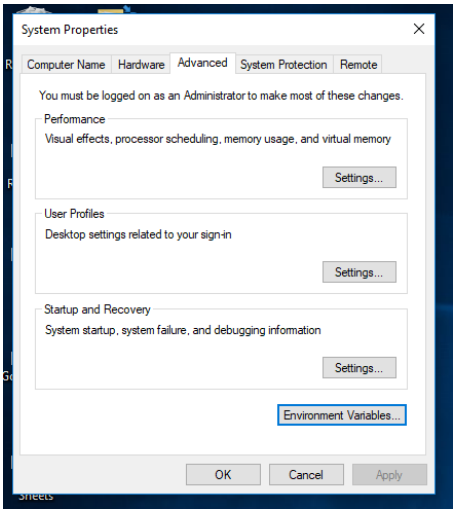
```
Microsoft Windows [Version 10.0.]
(c) 2018 Microsoft Corporation. All rights reserved.

c:\Users\Hasan>
c:\Users\Hasan> cd AppData\Local\Julia-1.3.1\bin
c:\Users\Hasan\AppData\Local\Julia-1.3.1\bin>
c:\Users\Hasan\AppData\Local\Julia-1.3.1\bin> julia
```

This approach is limiting, and so we tell Windows where to find the `Julia` executable, as described on this page. There is one error, in that this link does not mention the possible need to append `\bin`. The windows you should expect to see while adding the path is below.
Now, you should be able to start `Julia` from any folder:

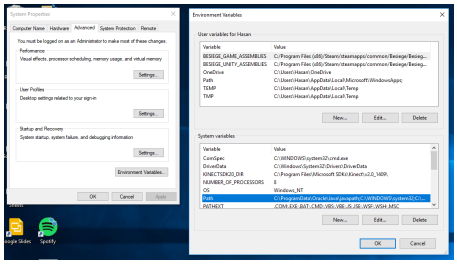
```
Microsoft Windows [Version 10.0.]
(c) 2018 Microsoft Corporation. All rights reserved.

c:\Users\Hasan>
c:\Users\Hasan> julia
```



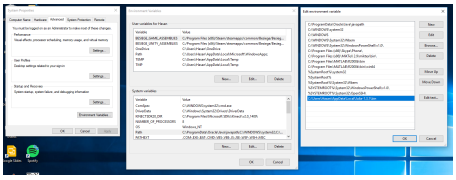
(a) Run `sysdm.cpl` through the Windows Run window



(b) Get to the 'advanced' tab, click on the 'Environment Variables' button.



(c) Highlight the 'Path' entry and **hit 'Edit'**.



(d) Click on 'New' and enter the folder location with the `\bin` appended as shown.

Figure 2: Adding `Julia` to the `PATH`.

3 Basic Usage

3.1 Recommended Workflow

My `Julia` involves using a Unix terminal to enter the `Julia` REPL (see Section 3.2) and run scripts from within that REPL. I edit scripts using a text editor. This setup is identical to starting `MATLAB` and calling code from the command window, after modifying it in the text editor window.

Previously this workflow meant switching between two windows. I now use `VS Code` as the editor, which allows me to pull up a terminal window within the editor window.

A **BAD** way to use `Julia` is to use it like you would use `Python` or `C++`, namely to call scripts from the command line by excuting '`julia <filename>`', without entering the REPL.

Another choice is to always use local environments for projects (see Section 3.3.1). This approach is similar to using `venv`/`vitualenv`/`poetry` in `Python`.

3.2 First Run

At the command prompt, after adding the `Julia` directory to the path, run:

```
$ julia
```

which takes you into an interactive shell (**REPL**, or read-eval-print-loop). More details on first steps in that shell can be found here.

To figure out which directory you are in:

```
julia> pwd()
```

To run a file `test.jl`, you can use either the command prompt

```
$ julia test.jl
```

or the `Julia` REPL:

```
julia> include("test.jl")
```

Again, I recommend avoiding the first method at least during code development.

3.3 Package Management For Projects

When you start `Julia`, you are in the default project/environment, which has a list of included packages stored in the files `Manifest.toml` and `Project.toml`. These files describe the installed packages, including a description of dependencies. See here for details. To run the code in Problem **??**, you must install some packages that aren't included in the install.

To add these packages, `Julia` offers a package REPL that can be accessed from the `Julia` shell by pressing `]` :

```
julia> ]
```

results in

```
(v1.3) pkg>
```

You can now add a package `PackageName` by running:

```
(v1.3) pkg> add PackageName
```

Every file you run after starting `Julia` has access to this package, because `Julia` starts in the default environment. When you always run a `.jl` file using the default `Julia` environment, the folder containing that file won't get its own `Manifest.toml` and `Project.toml` files.

3.3.1 Local Environments

To create local projects different from the default, you use environments. Environments provide a way to isolate package dependencies for specific code from each other. To do so, run

```
(v1.3) pkg> activate .
```

including the period, and the prompt changes to

```
(foldername) pkg>
```

Adding packages now will only add them to this project. In turn, this folder gets its own `Manifest.toml` and `Project.toml` files.

To see this, you can run the `st` command in the package REPL:

```
(v1.3) pkg> st
```

will return something different from

```
(foldername) pkg> st
```

if you added something to the specific project.

On my machine, for folder `dp_cl` I get

```
        (v1.3) pkg> st
    Status `~/.julia/environments/v1.3/Project.toml`
        [a2e0e22d] CalculusWithJulia v0.0.1
  [7876af07] Example v0.5.3
  [283c5d60] MeshCat v0.9.1
  [6ad125db] MeshCatMechanisms v0.6.0
  [366cf18f] RigidBodyDynamics v2.2.0
  [e61f16d8] RigidBodySim v1.3.0
```

versus

```
(v1.3) pkg> activate .
Activating environment at `~/Teaching/julia/dp_cl/Project.toml`

(dp_cl) pkg> st
    Status `~/Teaching/julia/dp_cl/Project.toml`
  [7073ff75] IJulia v1.20.2
  [283c5d60] MeshCat v0.5.0
  [6ad125db] MeshCatMechanisms v0.2.1
  [91a5bcdd] Plots v0.22.2
  [366cf18f] RigidBodyDynamics v1.3.0
  [e61f16d8] RigidBodySim v1.0.0
  [90137ffa] StaticArrays v0.10.2
```

The project I created has older versions of the same packages, which is one reason you might like to use environments.

4 Jupyter Notebooks

`Jupyter` is a nice browser-based environment in which to run `Julia` code as a notebook. Some students had issues getting `Jupyter` to work, which may be an issue with Windows.

You need the `IJulia` package installed in `Julia`, see this page for details. You don't have to install `Jupyter` first, but I did. I used `python3` and `pip3`:

```
$ pip3 install jupyter
```

Add the `IJulia` package:

```
julia> ]
(v1.3) pkg> add IJulia
```

To run jupyter in some folder:

```
$ jupyter notebook
```

at which point your browser launches a window showing files in that folder. You may then create a new notebook to run `Julia` code.

5 Creating Packages

The full guide to creating packages can be found here.

The relationship between `import` and `using` is described here. That `wikibook` also describes how modules and packages may be used. An example for a package directory can be found here.

A package is a project with a name, `uuid` and version entry in the `Project.toml` file, and a `src/PackageName.jl` file that defines the module `PackageName`. This file is executed when the package is loaded.

In some folder `baseFolder`, create a subfolder corresponding to `PackageName` using the command, in the `Pkg` REPL:

```
(v1.3) pkg> generate PackageName
```

Return to `Julia` REPL, enter the subfolder

```
julia> cd("PackageName/")
```

Enter `Pkg` REPL, activate this folder

```
julia> ]
(v1.3) pkg> activate .
(PackageName) pkg>
```

Return to the `Julia` REPL. When you execute

```
julia> import PackageName
```

you should see a successful precompilation the first time, and no errors on subsequent uses.

Now, try running

```
(julia)> PackageName.greet()
```

This function is predefined when you generate the package, via the **module** (a file) `src/PackageName.jl`:

```
module PackageName

greet() = print("Hello World!")

end # module
```

Your goal is to add functionality by modifying this file.

For example:

```
module PackageName

import Random
import JSON

greet() = print("Hello World!")
greet_alien() = print("Hello ", Random.randstring(8))

end # module
```

For this new version of the package to work, you will need to add `Random, JSON` to the project.

6 Incorporating 'Local' Packages

Suppose you are in a project with name/folder name `MyProject`. Someone shares their package `externalProject` with you, as a folder named `externalProject`. Assume that this package's `src` folder contains `externalProject.jl`:

```
module externalProject

greet() = print("externalProject Hello World!")
end # module
```

You can incorporate this package into your project in two ways. Note that a registered project (like `RigidBodySim`) does not require these steps.

6.1 Add `externalProject` folder as a subfolder of `MyProject`

To correctly use `externalProject` – defined in the subfolder `externalProject` – to your project in `MyProject`, you could run

```
julia>]
(v1.3) pkg> activate .
(MyProject) pkg> dev --local externalProject
julia> import externalProject
julia>externalProject.greet()
Hello World!
```

6.2 Add source files such as `externalProject.jl`

We can include the module defined in `externalProject.jl` to the folder `MyProject/src/`, and then modify the file `MyProject/src/MyProject.jl`:

```
module PackageName

greet() = print("Hello World!")
```

```
include("externalProject.jl")

end # module
```

We access the functions in `externalProject` as follows:

```
julia> PackageName.externalProject.greet()
externalProject Hello World!
julia> PackageName.greet()
Hello World!
```

7 Simulating Dynamical Systems And Robots

We focus on solving ODE's with continuous right-hand side. Some book-keeping by the programmer may allow simulation of discontinuous systems. A detailed description is available here.

The main idea is to combine solvers for Ordinary Differential Equations with pacakges that convert Universal Robot Description Formats (URDFs) into a set of simulatable ODEs.

We use the following packages, but not their latest versions due to compatibility issues:

```
[0c46a032] DifferentialEquations v6.12.0
[90137ffa] StaticArrays v0.12.1
[283c5d60] MeshCat v0.9.1
[6ad125db] MeshCatMechanisms v0.6.0
[366cf18f] RigidBodyDynamics v2.2.0
[e61f16d8] RigidBodySim v1.3.0
```

- The `RigidBodyDynamics` package allows specification of articulated robots, either explicitly, or through parsing of URDFs. Specification means link lengths and inertias, and joint information of how these links (rigid bodies) are joined to one another.
- `RigidBodySim` provides Julia tools for simulation and visualization of systems of interconnected rigid bodies (both passive and controlled), built on top of RigidBodyDynamics, DifferentialEquations, and RigidBodyTreeInspector.
- The packages `MeshCat` and `MeshCatMechanisms` allow visualization of mechanisms, and is used by `RigidBodySim`.

7.1 Example: Simulation Of Double Pendulum

We may manually define a (planar) double pendulum using the code. Note, this code **doesn't show the packages** you need to use/import:

```
g = -9.81 # gravitational acceleration in z-direction
world = RigidBody{Float64}("world")
doublependulum = Mechanism(world; gravity = SVector(0, 0, g))

axis = SVector(0., 1., 0.) # joint axis
I_1 = 0.333 # moment of inertia about joint axis
c_1 = -0.5 # center of mass location with respect to joint axis
m_1 = 1. # mass
```

```
frame1 = CartesianFrame3D("upper_link") # the reference frame in which the
    ↪  spatial inertia will be expressed
inertia1 = SpatialInertia(frame1, moment=I_1 * axis * axis', com=SVector
    ↪ (0, 0, c_1), mass=m_1)

upperlink = RigidBody(inertia1)

shoulder = Joint("shoulder", Revolute(axis))

before_shoulder_to_world = one(Transform3D, frame_before(shoulder),
    ↪ default_frame(world))

attach!(doublependulum, world, upperlink, shoulder, joint_pose =
    ↪ before_shoulder_to_world)

l_1 = -1. # length of the upper link
I_2 = 0.333 # moment of inertia about joint axis
c_2 = -0.5 # center of mass location with respect to joint axis
m_2 = 1. # mass
inertia2 = SpatialInertia(CartesianFrame3D("lower_link"), moment=I_2 *
    ↪ axis * axis', com=SVector(0, 0, c_2), mass=m_2)
lowerlink = RigidBody(inertia2)
elbow = Joint("elbow", Revolute(axis))
before_elbow_to_after_shoulder = Transform3D(frame_before(elbow),
    ↪ frame_after(shoulder), SVector(0, 0, l_1))
attach!(doublependulum, upperlink, lowerlink, elbow, joint_pose =
    ↪ before_elbow_to_after_shoulder)
```

The double pendulum is the `mechanism` named `doublependulum`.

Alternatively, if a URDF is available, like the one that comes with `RigidBodyDynamics`:

```
urdflocation = joinpath(dirname(pathof(RigidBodyDynamics)), "..", "test",
    ↪ "urdf", "Acrobot.urdf")
doublependulum = parse_urdf(urdflocation)
```

Once you have a mechanism, you can simulate it using:

```
state = MechanismState(doublependulum)
# there are multiple ways to set the full state config,velocity. one is:
set_configuration!(state, shoulder, 0.3)
set_configuration!(state, elbow, 0.4)
set_velocity!(state, shoulder, 1.)
set_velocity!(state, elbow, 2.);
# Call a simulator for initial condition started
# set total time and timesteps of solution:
ts, qs, vs = simulate(state, 5., ?? = 1e-3); #?? is Delta t,doesn't render
    ↪  here, see Julia files)
```

Now the solution qs with times ts may be animated:

```
vis = Visualizer(); open(vis); #Call this once, not for every simulation
mvis = MechanismVisualizer(doublependulum,URDFVisuals(urdflocation),vis)
setanimation!(mvis, ts, qs)
```

7.2 Example: PID Control Double Pendulum

A `control` function may be specified to set joint torques in the simulation. This function is pretty rigid, you only have the time and state to work with. The rest of the control function needs to be hard-coded.

```
urdfloc = joinpath(dirname(pathof(RigidBodySim)), "..", "test", "urdf", "
    ↪ Acrobot.urdf")
mechanism = parse_urdf(Float64, urdfloc)
state = MechanismState(mechanism)
set_configuration!(state, [0.1; 0.2])

function control!(tau, t, state)
    # Do some PD. Note tau is symbolic in actual Julia code
    tau .= -20 .* velocity(state) - 100*(configuration(state) - [pi;0.0])
end

problem = ODEProblem(Dynamics(mechanism,control!), state, (0., 10.))
sol = solve(problem, Vern7()) # Can replace Vern7() with other integration
    ↪  schemes
mvis = MechanismVisualizer(mechanism, URDFVisuals(urdfloc),vis)
setanimation!(mvis, sol; realtime_rate = 1.0);
```

7.3 Example: Neural Network Control Of Double Pendulum

If your control torque computations need more than just time t and state, we need a workaround to the rigid nature of the `control` function. For example, let's say that a structure `nncontrol` contains neural networks that we wish to use to compute torques. (Note: I use Julia package `Flux` to work with neural networks.) Here's one way that I know of, using the `setparams!` feature in the `Dynamics` function, where we define a new intermediate control function `mysp!` which has more freedom.

```
function control!(tau, t, state)
    tau .= p
end

function mysp!(state,p,nncontrol)
        qgoal=[-0.6;1.3]
        p[:] = -12*velocity(state)-50*(configuration(state)-qgoal) +
            ↪ nncontrol.gravity_net(qgoal)
end

problem = ODEProblem(Dynamics(mechanism,control!;setparams!=(state,p)->
    ↪ mysp!(state,p,nncontrol), state, (0., 10.),p)
sol = solve(problem, Vern7())
mvis = MechanismVisualizer(mechanism, URDFVisuals(urdfloc),vis)
setanimation!(mvis, sol; realtime_rate = 1.0);
```

There may be other and better ways to do the same thing. The function calls for the method above may also be overly complicated.

7.4 Contacts

Most robot arms have a base link that is rigidly attached to the world frame. For a robot like a quadruped robot, none of its links are rigidly attached to the world frame. Its base link has a 'floating' joint with respect to the world frame. If we simulate this quadruped, it falls through the 'floor 'under gravity. Simulators like `Gazebo` can take in meshes and create collisions. Unfortunately, `Julia` falls short on this aspect.

What we need to do is to add contacts manually.

This section describes describes the process. Let's repeat the usual steps: Load packages:

```
using RigidBodyDynamics, MeshCat, MeshCatMechanisms
using LinearAlgebra, Printf
```

Load mechanism:

```
# load mechanism
mechanism=parse_urdf("Cheetah.urdf";scalar_type=Float64,floating=true)
remove_fixed_tree_joints!(mechanism)
```

Define the ground that the robot's feet will collide with:

```
# create a half space representing ground
hs_p = Point3D(root_frame(mechanism),0.0,0.0,0.0)
hs_v = FreeVector3D(root_frame(mechanism),0.0,0.0,1.0)
hs = RigidBodyDynamics.Contact.HalfSpace3D(hs_p,hs_v)
ce = RigidBodyDynamics.Contact.ContactEnvironment{Float64}()
# add half space to mechanism
push!(ce.halfspaces,hs)
mechanism.environment = ce
```

Define the contact model:

```
# create a soft contact model conmod3
import RigidBodyDynamics.Contact.ViscoelasticCoulombModel
import RigidBodyDynamics.Contact.HuntCrossleyModel
import RigidBodyDynamics.Contact.SoftContactModel
conmod1 = HuntCrossleyModel(50e3,1.5*0.2*50e3,1.5)
conmod2 = ViscoelasticCoulombModel(0.5,10e5,10e3)
conmod3 = SoftContactModel(conmod1,conmod2)
```

Create contact points on bodies in the Mechanism

```
cp1 = RigidBodyDynamics.Contact.ContactPoint(Point3D(default_frame(bodies(
    ↪ mechanism)[7]),0.0,0.0,-1.0),conmod3)
add_contact_point!(bodies(mechanism)[7],cp1)
cp1 = RigidBodyDynamics.Contact.ContactPoint(Point3D(default_frame(bodies(
    ↪ mechanism)[8]),0.0,0.0,-1.0),conmod3)
add_contact_point!(bodies(mechanism)[8],cp1)
cp1 = RigidBodyDynamics.Contact.ContactPoint(Point3D(default_frame(bodies(
    ↪ mechanism)[9]),0.0,0.0,-1.0),conmod3)
add_contact_point!(bodies(mechanism)[9],cp1)
cp1 = RigidBodyDynamics.Contact.ContactPoint(Point3D(default_frame(bodies(
    ↪ mechanism)[10]),0.0,0.0,-1.0),conmod3)
add_contact_point!(bodies(mechanism)[10],cp1)
```

Define a control that regulates the quadruped towards some target pose:

```
# PD control on shoulders and joints
function mytorque!(torques::AbstractVector, t, state::MechanismState)
    desvec = [1.0;0;0;0;0;0;0;.1;.1;-.1;-.1;-1;-1;1.0;1.0]
    torques .= 0
    for i=1:8
    torques[6+i]=-5*state.v[6+i] - 50*(state.q[7+i] - desvec[7+i])
    end
end
```

Simulate as usual:

```
# set the robot state
state=MechanismState(mechanism)
zero_velocity!(state)
set_configuration!(state,[1.0;0;0;0;0;0;2.0;.5;.5;-.5;-.5;-1;-1;1;1])

# define simulation time
final_time = 5.00

# simulate
ts, qs, vs = simulate(state, final_time,mytorque!);

# display
mvis2 = MechanismVisualizer(mechanism, URDFVisuals("Cheetah.urdf"),vis2)
setanimation!(mvis2,ts,qs)
```

You should now see the Cheetah drop from a height, and land on its feet with a small bounce in the body due to leg compliance. The feet don't move much at all.