NTRTSim Documentation

Perry Bhandal

Dec 19, 2018

Contents

The NASA Tensegrity Robotics Toolkit (NTRT) is a collection of C++ and MATLAB software modules for the modeling, simulation, and control of Tensegrity Robots. The NTRT Simulator is a tensegrity-specific simulator built to run ontop of the Bullet Physics Engine, version 2.82.

NTRT's source code is available on GitHub:

https://github.com/NASA-Tensegrity-Robotics-Toolkit/NTRTsim

CHAPTER 1

Contents

1.1 Setup/Installation

1.1.1 Installing NTRT Without a Virtual Machine

See this page for more information:

https://raw.githubusercontent.com/NASA-Tensegrity-Robotics-Toolkit/NTRTsim/master/INSTALL

1.1.2 Installing NTRT in a Virtual Machine

Currently the NTRT OS X build does not work, and Windows is not supported. If you use either operating system you will need to run NTRT in VirtualBox (or some other virtualization software) in the meantime. Here's how you do that:

For now, here is how to install it and get it going...

1. Install VirtualBox on your computer. https://www.virtualbox.org/. Theoretically you could use whatever virtualization program you wanted, but virtualbox is easy to learn.
2. Download the .ova file: http://ntrt.perryb.ca/storage/vm/ntrt_vb_1-1.ova
3. Open Virtualbox. "File -> Import Appliance" and select the .ova file.
4. Adjust the number of processors, RAM, and video memory you'd like to allocate to the guest. I'd suggest 2 processors, 4GB of RAM, and max out the video memory. If your computer is a bit slower, you might be able to get away with one one processor and down to 2 or 3GB of RAM.
5. Once it's done importing, click on "Settings" to confirm things are OK. If you see an error in the bottom of the Settings window that says "Invalid Settings Detected," change your settings according to the error's recommendation.
6. Check the "Enable 3D Acceleration" box under "Display." This makes the simulation visualization run at a reasonable speed.
7. Run the virtual machine, and enjoy! You should be able to run the SUPERball demo by opening up a Terminal window and typing the commands:

8. Once your virtual machine boots up, you'll be prompted for a password the "tensegribuntu" user. The password is tensegrity.

Once the virtual machine is online, you can test it by opening a terminal and running the following:

cd /home/tensegribuntu/NTRTsim/bin ./build.sh cd /home/tensegribuntu/NTRTsim/build/examples/SUPERball ./AppSuperBall

Known Issues: - Virtualbox's LibGL 3D drivers are wonky and throw an error when they're loaded: it's actually fine and working though. https://www.virtualbox.org/ticket/12941 (Links to an external site.) - The screen resolution options don't work from within Xubuntu: just drag/resize the window and it should automatically change the resolution, though.

Once you have NTRT installed and running, open your web browser and it will open three NTRT related pages in separate tabs: tutorials, doxygen and NTRT's github page. The tutorials are a good place to start in gaining familiarity with the sytstem, or in order to learn how to contribute changes you make into the NTRT master branch.

1.1.3 General Linux Help

In case it helps, here is a brief list of commands you can run in the terminal: http://www.pixelbeat.org/cmdline.html (Links to an external site.). I know Linux is not intuitive at first, but I'm here to help!

1.2 YAML Model Builder

The YAML model builder was built to allow users to easily create, modify and run tensegrity structures using NTRT's simulator. Contrary to building structures in C++, using the YAML model builder does not require any programming knowledge, it does not require any knowledge of NTRT's libraries and does not require a compilation step each time a modification to a structure is made. Structures defined in YAML are also considerably shorter and simpler to create than their C++ equivalents. YAML is a human-readable data serialization format. It was chosen because of its simplicity and clean syntax.

1.2.1 Building Simple Structures

Shown below is a 3-Prism structure defined in YAML. It is important to note that spacing and indentation are important in YAML and inconsistent spacing will cause errors when parsing the file. Using a text editor such as Notepad++ or Sublime Text is recommended when defining structures. The site http://codebeautify.org/yaml-validator can be used to make sure the structure you have defined is valid YAML before running it in the simulator.

```
nodes:
  bottom1: [-5, 0, 0]
  bottom2: [5, 0, 0]
  bottom3: [0, 0, 8.66]

  top1: [-5, 5, 0]
  top2: [5, 5, 0]
  top3: [0, 5, 8.66]

pair_groups:
  rod:
    - [bottom1, top2]
    - [bottom2, top3]
    - [bottom3, top1]
```

(continues on next page)

(continued from previous page)

```
  string:
    - [bottom1, bottom2]
    - [bottom2, bottom3]
    - [bottom1, bottom3]

    - [top1, top2]
    - [top2, top3]
    - [top1, top3]

    - [bottom1, top1]
    - [bottom2, top2]
    - [bottom3, top3]
```

Running Structures in the Simulator

Before running a structure in the simulator, make sure you have completed all the setup/installation steps for NTRT. The YAML model builder is run by running the BuildModel executable (located in the yamlbuilder directory) and providing it with the path to a YAML structure as a command line argument. To run the structure above in the simulator go to the root NTRT directory and type in:

```
build/yamlbuilder/BuildModel resources/YAMLStructures/BaseStructures/3Prism.yaml
```

Adding Nodes

Tensegrity structures are made up of rods and strings. The vertices of the rods and strings are defined using the keyword "nodes". Each node is defined by a key-value pair where the key is a unique name and the value is a 3-element array with coordinates of the node. Names should not contain spaces, periods, or slash characters. The x and z coordinate axes are horizontal and run parallel to the ground (with the z axis representing depth). The y coordinate axis is vertical and perpendicular to the ground.

```
nodes:
  bottom1: [-5, 0, 0]
```

Adding Pairs

Rods and strings are defined using the "pair_groups" keyword and are grouped by tags. Tags determine how each pair is built in the simulator. There are different types of rods and strings that can be used to build any given pair. These are defined using builders which will be discussed in more detail in the next section. Default builders, that define a default rod and string, are added automatically to match the tags "rod" and "string" so that a tensegrity structure can be defined without having to worry about builders. Each tag holds a list of one or more pairs, where each pair is defined by the two nodes it connects. If a tag has a space such as "prism string" it is treated as two different tags that are assigned to a given pair. Tags should not use any period or slash characters.

```
pair_groups:
  rod:
    - [bottom1, top2]
```

Adding Builders

Shown below is an example of builders that can be used to modify how rods and strings are built in the simulator.

```
builders:
  rod:
    class: tgRodInfo
    parameters:
      density: 0.688
      radius: 0.31
  string:
    class: tgBasicActuatorInfo
    parameters:
      stiffness: 1000
      damping: 10
      pretension: 1000
```

The "builders" keyword is used to define one or more builders. Builders work through tag matching. For example, the builder "rod" will match any pairs with the tag "rod". The builder "rod" will also match any pairs with the tag "prism rod" since "prism rod" is treated as two different tags and the "rod" builder is looking for any pairs that include the tag "rod". As was mentioned earlier, the YAML model builder automatically adds a default "rod" and "string" builder to make building structures even faster. If a "rod "or "string" builder is defined by the user, it will override the default "rod" and "string" builder. Builders defined inside one file should never overlap (eg. using a "muscle" and "leg muscle" builder in the same file).

Each builder tag needs to be given a class using the "class" keyword. The class determines the properties of the rod or string. The basic rod and string classes are tgRodInfo and tgBasicActuatorInfo. More information about different string/cable classes can be found in the motors and cables section. Each builder takes a number of parameters which are specified using the "parameters" keyword. All parameters are optional (if they are not specified they will take on default values). Some of the most common parameters for tgRodInfo and tgBasicActuatorInfo are shown in the example above. More information about the parameters used by tgRodInfo or tgBasicActuatorInfo can be found in their respective classes.

Adding Single-Node Structures versus Multiple-Element Structures (Spheres vs. Rods/Boxes)

Most stuctures are built out of more than one node. For example, rods are made from two nodes: one for each endpoint of the rod. Consequently, to make a rod, you'd need to specify two nodes (with whatever tag you'd like), then make a pair with a *specific* tag which then must correlate with a builder. For example, let's emphasize using a full example of making one rod:

```
nodes:
  rodEndA: [-5, 0, 0]
  rodEndB: [5, 0, 0]

pair_groups:
  rod:
    - [rodEndA, rodEndB]

builders:
  rod:
    class: tgRodInfo
    parameters:
      density: 0.688
      radius: 0.31
```

Here, you can see that "rodEndA" and "rodEndB" are arbitrary names. You could have chosen whatever you'd like, as long as the same node names (tags) are used for the line in pair_groups. Instead, the tag "rod" is the important one! That's what correlates the nodes to a specific type of builder.

This is NOT the case when creating single-element rigid bodies. For example, consider a sphere. Spheres only have

one node associated with them: their centerpoint. Since it doesn't make sense to have spheres be a "pair," NTRTsim implements the following. Single-element rigid bodies must correlate the *node tag* to a builder, not the *pair tag* to a builder. Here's an example of creating a sphere.

```
nodes:
  examplesphere: [0, 5, 0]

builders:
  examplesphere:
    class: tgSphereInfo
    parameters:
      density: 0.5
      radius: 2
```

If you create a pair_group with the same tag as used for a builder, nothing will happen. (To-do: check and confirm nothing is accidentally created, handle these edge cases.) Point is - use the above example for making spheres.

1.2.2 Combining Structures

Shown below is an example of a spine-like structure, made by combining six Tetrahedrons.

```
substructures:
  t1/t2/t3/t4/t5/t6:
    path: ../BaseStructures/Tetrahedron.yaml
    offset: [0, 0 , -12]

bond_groups:
  string:
    t1/t2/t3/t4/t5/t6/node_node:
      - [front, front]
      - [right, right]
      - [back, back]
      - [left, left]
      - [right, front]
      - [right, left]
      - [back, front]
      - [back, left]
```

Multiple substructures can be combined in a superstructure using the "substructures" keyword. Superstructures inherit builders defined in substructures. If there is an overlap between builders in a superstructure and a substructure, the builder in the superstructure will override the builder in the substructure.

Child Structure Attributes

Each substructure is defined by a name and one or more attributes. If multiple structures share the same attribute value, an abbreviated syntax (as shown above) using the slash character can be used.

Path

Every child structure must be provided with a file path. The path can be absolute or relative to the parent structure.

Rotation

Rotation attributes are always applied first regardless of the order in which they are defined. Rotation attributes are defined as shown below using an axis, angle and an optional reference point. The axis (a vector array) refers to the axis of rotation, the angle (in degrees) refers to the angle that the structure is rotated by and the reference point (a coordinate array) refers to the point around which the structure is rotated. If no reference point is specified the center of the structure is used as a reference point.

```
example_structure:
  rotation:
    axis: [1, 0, 0]
    angle: 90
    reference: [0, 0, 0]
```

Scale

The scale attribute scales the child structure by a specified amount. This value must be a decimal number, not a fraction. Structures are scaled around their center.

```
example_structure:
  scale: 0.5
```

Translation

The translation attribute moves the structure by the specified vector array.

```
example_structure:
  translation: [0, 20, 0]
```

Offset

The offset attribute is useful for spine-like structures where multiple structures are added in a row and individual structures need to be offset from one another. Each structure is offset from its preceding structure by the specified offset vector.

```
t1/t2/t3/t4/t5/t6:
  offset: [0, 0 , -12]
```

Connections Between Structures

Connections between structures can be defined using the "bond_groups" keyword. Similar to "pair_groups", "bond_groups" are grouped by tags. Bonds are defined by the name of the structures that are being connected and the bond type used to connect those structures. In the example below the "top" node from the foot is connected to the "bottom" node of the leg using a node-to-node connection.

```
bond_groups:
  string:
    foot/leg/node_node:
      - [top, bottom]
```

If the "leg" structure is itself a superstructure with multiple substructures (such as a "knee" substructure) then the child node can be specified using the dot notation shown below. The dot character is used to denote a child structure. Multiple dot characters can be used if necessary (eg. "parent.child.grandchild.node"). Using this notation is only necessary if the node name is not unique among a structure's children.

```
bond_groups:
  string:
    foot/leg/node_node:
      - [top, knee.bottom]
```

Series of Substructures (Including Spine Structures)

A series of substructures, for example the vertebrae in a spine structure, can be easily defined using the syntax below. The syntax makes it possible to define a set of pairs that is used to connect more than two structures. When using this syntax with slashes between multiple substructures, the connections are only made between adjacent substructures in the series of slashes. In this example, the front-to-front connection is made between t1 and t2 (since they are next to each other) as well as t2 to t3 (for the same reason), but no connection is made from t1 to t3.

```
bond_groups:
  string:
    t1/t2/t3/node_node:
      - [front, front]
```

Node-to-Node Connections

Node-to-node connections simply add a string between two specified nodes (similar to how "pair_groups" work). For superstructures it is recommended to use the bond_groups syntax rather than the "pair_groups" syntax because it makes it clear which structures are being connected (without having to use the dot notation to repeat which structure the nodes for each pair belong to).

Node-to-Edge Connections

The structure below shows six 3-Prisms combined using node-to-edge connections.

```
substructures:
  3prism1/3prism2/3prism3/3prism4/3prism5/3prism6:
    path: ../BaseStructures/3Prism.yaml

bond_groups:
  horizontal_string:
    3prism1/3prism2/3prism3/3prism4/3prism5/3prism6/node_edge:
      - [top1, bottom1/bottom2]
      - [top2, bottom2/bottom3]
      - [top3, bottom3/bottom1]
```

Node-to-edge connections are defined using three or more pairs, where each pair consists of a node and an edge. Nodes can be connected to edges and vice versa. There can even be a mix of node-to-edges and edges-to-nodes within a single "node_edge" connection. An edge is defined by two nodes separated by the slash character. Node-to-edge connections work by attaching a node directly to the middle of a string. For node-to-edge connections, rotations and translation attributes for child structures are done automatically by the YAML model builder and do not need to be specified.

1.3 Learning Library Walk-through

This is a brief tutorial on how to include the machine learning libraries found in src/learning into your code. See section Using learning in Python for our current experimental multi-process Python scripts

1.3.1 Initial Steps

This example roughly follows the code in src/examples/learningSpines/BaseSpineCPGControl.h/cpp There is a lot going on in this file, so I'll try to highlight the relevant points. Note that craterEscape's escapeController provides a second example, but it doesn't utilize a few key features of the library.

First, place the following objects as member variables in your .h file (see BaseSpineCPGControl.h):

- A std::string that holds the filename of your configuration file
- A configuration object, which processes the configuration file
- An evolution object which holds all of the parameters for the population of controllers in memory
- An adapter to the evolution object which handles some of the interfacing with controllers
- A boolean that determines whether learning is active this run

Second, all of these should be initialized in your *constructor*. If you wait for onSetup, key functions within the evolution and adapter objects will never be called. You can probably just copy the constructor for BaseSpineCPGControl, +/- how complicated your config struct needs to be.

Third - here's what's important for onSetup: 1. Call adapter.initialize, this will get you a new controller queued up for this run (if learning is turned on) 2. If you're running with the same parameters for an entire trial (i.e. this part of your controller is open loop) go ahead and call adapter.step dt can be zero, and the state vector can be empty, you're just getting a nested vector of random parameters so far. 3. These parameters are scaled 0 to 1, so you're going to have to scale them to what you want (each parameter is an 'action'). This is done in the scaleNodeActions function (which is simpler than scaleEdgeActions) 4. Apply these actions to your controller as appropriate. craterEscape is actually a good example here, since its simpler. Note there are some controllers that need new parameters every step. You can find these in dev, we'll update the tutorials with that info once they're in examples (before v1.2).

Fourth - get your config.ini setup There's a preliminary listing of what all of these variables do here: http://ntrt.perryb.ca/doxygen/config_full.html The most important things are: 1. The learning variable 2. numberOfActions 3. numberOfControllers

You can choose to dump all of your parameters in one controller (numberOfActions = numberOfParameters), or divide it up by repeating parameters (i.e. sine waves have 4 parameters (actions) and you might use 8 or 24 of them in your controller). There are advantages to each of these approaches depending on which learning algorithm you're using (see next section).

Note that this is best placed in the resources folder so that you can run your code from anywhere on the path.

At this point you should be ready to go! Email me: bmirletz (at) case (dot) edu if you have trouble

1.3.2 Algorithm Options

(under construction)

1. Monte Carlo
2. Gaussian Sampling
3. Co-evolution
4. Genetic Algorithm

1.3.3 What's going on under the hood?

For the default "one process" learning algorithm, here's what's going on:

When evolution objects are constructed, they create sets of random parameters in memory according to the specifications in config.ini. Currently, one of the random sets of parameters is overwritten with the best prior set if the 'start seed' parameter in config is on.

When the adapter's initialize function is called, it asks the evolution object for the 'next set of controllers', which typically just supplies it with a nested set of vectors (actions). The controller then proceeds with the rest of the 'trial' (running a simulation with these sets of parameters) until the simulation ends (typically after a certain number of steps with graphics off). Then the simulation calls reset, onTeardown is called, and the controller provides a score to the adapter which passes that score on to the evolution object. The next step of a reset is onSetup, so the process resumes.

However, after a certain number of trials (depends on the type of learning), counters within the 'next set of controllers' function trigger the end of a generation. At this point, the controllers are sorted according to their scores, and then mutation, children, and elitism happen according to the algorithm. Finally, a new controller is passed to the adapter out of the new population.

1.3.4 Using learning in Python

An alternative is to use Python to perform the sorting and mutating. In this case, parameters are often dumped straight to a common scores.csv file, and then sorted out with Python scripts. These scripts can either generate another csv like file with just parameters (as in craterEscape), or new .nnw files (this may still just be on a branch).

Our new (as of April 2015) multi-process learning script (scripts/NTRTJobMaster.py) uses JSON to interface with NTRT. In order to use this you need to provide a JSON interface to your controller, and write a specification file. Your controller must write the score back to the JSON file. Your App must also support boost command line options. The specification file fills the role of Config.ini, and has parameters as follows:

{ "filePrefix" : A string that forms the beginning of your filename

"fileSuffix" : A string that forms the suffix of your filename

"resourcePath" : A string that points to the resources folder

"lowerPath" : A string that

"executable" : "./../build/dev/btietz/TC_goal/AppGoalTerrain",

"terrain" : A nested array of booleans (0s and 1s) that is passed to command line options for terrain or similar. The number of arrays dictates the number of runs per job

"learningParams": A container for learning parameters. (the JSON object is below, don't put things here)

{

"trialLength" : An integer - the number of timesteps for each trial

"numTrials" : An integer - the number of trials in each generation. If this is equal to the population size things will be run deterministically. Random controllers will fill in subsequent trials (cheap version of co-evolution)

"numGenerations" : Total number of generations before exiting. As of May 2015 numTrials * numGenerations must be less than 30,000. We will post here when this is fixed.

"nodeVals" : One of these for each type of parameters you want. This allows for heterogenous data types.

{

"learning" : Whether or not this process is learning

"startingControllers" : How many controllers to read in. The script will start with filePrefix_0.fileSuffix and work up one at a time until this number Additional parameters to reach populationSize will be chosen randomly

"monteCarlo" : Is learning using monteCarlo? If true, each set of parameters will be random As of May 2015 if you're running monteCarlo I would recommend numGenerations = 1, as files will be overwritten and data will be lost.

"numberOfStates" : Integer. The number of inputs to the tuned controller. If 0 these are just data, if >= 1 a neural network will be used

"numberOfOutputs" : Integer. The number of output parameters for the neural network or controller.

"numberHidden" : Integer. Only matters if numberOfStates >= 1. The number of neurons in the ANN's hidden layer

"numberOfInstances" : Integer. Must be > 0 if numberOfStates = 0. How many times are you going to interate through the outputs? Great for repeated parameters like weights. If numberOfStates > 0 this is ignored.

"populationSize" : Integer. How many controllers are we testing?

"useAverage" : Should the controllers be judged on their average value (true) or maximum value (False). Average recommended with co-evolution on,

"numberToMutate" : Integer. How many controllers have their parameters changed by mutation? This + numberOfChildren must be less than population size I recommend around half. The top N will be mutated.

"numberOfChildren" : Integer. How many times should the controllers 'mate' to cross-pollinate parameters? Mating pairs are choses by weighted probability, based on the scores.

"mutationChance" : Double between 0 and 1 (inclusive) How often a single parameter within a controller is mutated. 1 is always 0 is never.

"mutationDev" : Double between 0 and 1 (inclusive). What is the deviation of the normal distribution used to mutate parameters?

"paramMax" : Double. What is the largest this parameter should be?

"paramMin": Double. What is the smallest this parameter shoudl be

"childMutationChance" : Double. How often should a cross-pollinated child be mutated. This applies to the entire controller.

} Add a comma here if you have more than one set of parameters.

}

}

Note that replacing the explanations with numbers and added commas the end of each line, the above would lead to a valid JSON specification file, similar to: https://github.com/NASA-Tensegrity-Robotics-Toolkit/NTRTsim/blob/master/scripts/TCSpec.json

See this issue for discussion and more instructions for running the script: https://github.com/NASA-Tensegrity-Robotics-Toolkit/NTRTsim/issues/133

1.3.5 Historical Notes and Future Work

Many of the elements of the current learning library actually exist because we used to lose controller objects on reset. Therefore the evolution object would be owned by main, and passed to a controller which would use an adapter to read the parameters. Our new architecture gives us a lot more flexibility, so we should take advantage of it.

Other opprotunities for future work:

- Update config.ini to JSON: https://github.com/NASA-Tensegrity-Robotics-Toolkit/NTRTsim/issues/42
- Merge features of annealEvolution and neuroEvolution: https://github.com/NASA-Tensegrity-Robotics-Toolkit/NTRTsim/issues/131

1.4 How to Choose and Configure NTRT's Motor and Cable Models

This tutorial assumes you're familiar with the basics of building structures in NTRT. The Doxygen documentation and examples are currently the best place to get that information: http://ntrt.perryb.ca/doxygen/

The basic model we're using is that the robot has a motor connected to each cable, which is in turn connected to a spring. The motor adjusts the rest length of this system according to rules defined by the motor model.

NTRT provides two options for motors, and two options for cables. All of these are a subclass of tgSpringCable and use the same attributes in the config struct These can be subclassed into a whole range of realistic simulations for your application. The basic options in core are as follows:

1. Cables can either have contact dynamics (interact with the world) or only act on the bodies they actuate
2. Motors can be perfect actuators (easy to control) or possess inertia, friction, and a torque speed curve.

Both of these are chosen with simple switches (similar child classes) in the builder tools.

1.4.1 Cables

A cable without contact dynamics is a tgBasicActuator these are constructed by passing a tgBasicActuatorInfo to a spec. If you want contact dynamics, just use tgBasicContactCableInfo instead. There are no differences in the config structs. An example with a conditional compile can be found at line 195 of the Octahedral Complex demo: https://github.com/NASA-Tensegrity-Robotics-Toolkit/NTRTsim/blob/master/src/examples/learningSpines/OctahedralComplex/FlemonsSpineModelLearningCL.cpp

1.4.2 Motors

tgBasicActuator is a perfect actuator. As long as you don't exceed the force or speed constraints set by the config (from tgSpringCable) then it will make your desired changes to the cable's rest length. Great for getting started, but not the way motors really work.

For more complex behaviors, use tgKinematicActuator. This adds four new parameters to the specification, the radius of the motor, motor friction, motor inertia, and whether the motor is backdrivable (a boolean). More details can be found on the Doxygen page. To use it, just use tgKinematicActuatorInfo. Note that the config for a tgKinematicActuator is a subclass of tgSpringCable, so you can pass that config to either actuator class. These can also have contact dynamics by calling tgKinematicContactCableInfo. An example with a conditional compile can be found at line 161 of the TetrahedralComplex example: https://github.com/NASA-Tensegrity-Robotics-Toolkit/NTRTsim/blob/master/src/examples/learningSpines/TetrahedralComplex/FlemonsSpineModelLearning.cpp

If you need a different torque speed curve or friction behavior, you should be able to subclass the functions getAppliedTorque or integrateRestLength in tgKinematicActuator

1.4.3 Controlling Each Type of Motor

tgBasicActuator controls rest length directly, these are the commands passed by setControlInput. tgKinematicActuator controls the motor's applied torque, which is then integrated to produce a rest length change. Thus control needs to be a little more complicated. Look at the controller classes for PID controllers, and other things that are useful for more complex low level controls.

Tutorials on controllers will hopefully get written in August 2015.

If you have questions contact Brian at bmirletz (at) case (dot) edu

1.5 Data Management and Logging

Data management in NTRT can be done a variety of different ways, depending on how advanced your needs are. Some NTRT components, such as the learning library, have custom-build data management systems.

However, a new data management and logging framework, the tgDataManager, has been included in NTRTsim as of January 2017. This framework is intended to replace all past custom-written data logging code with a general solution for all NTRT applications.

This tutorial shows how to use one of the new data managers, a tgDataLogger2. For new users who want to log data, please use tgDataLogger2. It will be easiest!

PLEASE NOTE that, at the moment, you cannot create data managers from within YAML files. You can use data managers with YAML structures, but you must create a separate App file (instead of using yaml-builder/BuildTensegrityModel.cpp). See section below on using tgDataLogger2 with a YAML model.

1.5.1 Quick Start: Log data from rods and actuators using tgDataLogger2

In your application directory, change the CMakeLists.txt file to include the *sensors* library. For example, if your CMakeLists previous had the line:

```
link_libraries(tgcreator controllers core)
```

You would change this to:

```
link_libraries(tgcreator controllers core sensors)
```

The following code will direct your application to log data from tgRods and tgSpringCableActuators. You would write this code in an App file, not in a tgModel file.

First, include the following header files:

```
#include "sensors/tgDataLogger2.h"
#include "sensors/tgRodSensorInfo.h"
#include "sensors/tgSpringCableActuatorInfo.h"
```

Then, **after** you create your tgModel and add it to the simulation, perform the following steps:

1. Create a tgDataLogger2, passing in the name of the log file that you would like to create. Include a time interval for logging. If you don't include this, you will get a sample at each timestep, and your logfile will be extremely large. A suggested sampling time interval is 0.1 sec.

```
std::string log_filename = "~/path_to_my_logs/example_logfile";
double samplingTimeInterval = 0.1;
tgDataLogger2* myDataLogger = new tgDataLogger2(log_filename,␣
↪samplingTimeInterval);
```

(continues on next page)

(continued from previous page)

2. Add the model to the data logger. If your pointer to your tgModel was myModel,

```
myDataLogger->addSenseable(myModel);
```

3. Create the two sensor infos, one for tgRods and the other for tgSpringCableActuators.

```
tgRodSensorInfo* myRodSensorInfo = new tgRodSensorInfo();
tgSpringCableActuatorSensorInfo* mySCASensorInfo = new␣
↪tgSpringCableActuatorSensorInfo();
```

4. Add the sensor infos to the data logger.

```
myDataLogger->addSensorInfo(myRodSensorInfo);
myDataLogger->addSensorInfo(mySCASensorInfo);
```

5. Add the data logger to the simulation.

```
simulation.addDataManager(myDataLogger);
```

All combined, the code in your application might look like:

```
...
simulation.add(myModel);

std::string log_filename = "/home/drew/NTRT_logs/demo_application";
tgDataLogger2* myDataLogger = new tgDataLogger2(log_filename);
myDataLogger->addSenseable(myModel);

tgRodSensorInfo* myRodSensorInfo = new tgRodSensorInfo();
tgSpringCableActuatorSensorInfo* mySCASensorInfo = new␣
↪tgSpringCableActuatorSensorInfo();
myDataLogger->addSensorInfo(myRodSensorInfo);
myDataLogger->addSensorInfo(mySCASensorInfo);

simulation.addDataManager(myDataLogger);
simulation.run();
```

When the simulation exits, you will have a log file written to /home/drew/NTRT_logs/demo_application_(some_timestamp).txt.

Be sure to change the log file path to something that exists on your computer. A folder under your home directory is a good choice. Note, however, that if you write the log files somewhere outside the NTRT github directory, your logs will not be synced with the repository. (It is a *good idea* to not sync your logs, since it will take up space on other people's computers.)

1.5.2 The structure of a tgDataLogger2 data file

The tgDataLogger2 log file names are ended with a timestamp. This timestamp is when you ran your application. For example, 01082017_150631 is January 8th 2017 at 3:06pm (and 31 seconds.)

The log file from a tgDataLogger2 is a comma-separated-value file (CSV). It can be read by most spreadsheet applications (e.g. MS Excel, LibreOffice Calc) as well as MATLAB (see for example MATLAB's csvread command: https://www.mathworks.com/help/matlab/ref/csvread.html).

The log file consist of the following:

1. A line of debugging information, stating what sensors have been created on the model, and the timestamp of the log file.
2. Headings for each of the sensor readings.

   These headings have the following structure: First, the sensor number, which is assigned arbitrarily by tgDataLogger2. Then, the type of sensor, then an open parenthesis "(" and the tags of the specific object that's being sensed, then a ")." and a label for the specific field that will be output in that row.

   For example, if sensor 3 will be sensing a rod with tags "t4 t5", its label for the X position would be "3_rod(t4 t5).X"
3. Rows of output of the sensor data

   Note that sensor data are taken at every timestep of the simulation, and these timesteps are saved as the first column of the log file.

An example first few lines of a log file with one rod sensor only, on a single model with two rods, with each rod having the tgTags "rod", is:

```
tgDataLogger2 started logging at time 01082017_150631, with 2 sensors on 1 senseable␣
↪objects.
time,0_rod(rod).X,0_rod(rod).Y,0_rod(rod).Z,0_rod(rod).Euler1,0_rod(rod).Euler2,0_
↪rod(rod).Euler3,0_rod(rod).mass,1_rod(rod).X,1_rod(rod).Y,1_rod(rod).Z,1_rod(rod).
↪Euler1,1_rod(rod).Euler2,1_rod(rod).Euler3,1_rod(rod).mass,
0.001,0,6,0,0,-0,0,38.9055,0,10,0,0,-0,0,38.9055,1.67374,2,200,
0.002,0,5.9999,0,0,-0,0,38.9055,0,9.9999,0,0,-0,0,38.9055,1.67374,1.99998,199.987,
0.003,0,5.99972,0,0,-0,0,38.9055,0,9.9997,0,0,-0,0,38.9055,1.67374,1.99995,199.969,
...
```

Note that, at the time of the writing of this tutorial, the "1 senseable objects" refers to the number of base tgSenseable objects attached to the data logger, NOT the total number of models and children. E.g., this is the number of models/senseables that were explicitly attached using the addSenseable method in the App file. The above example had 1 tgModel with 3 children (2 rods and 1 spring cable actuator), and sensors were only created for the rods.

Sensor data from a tgRod using tgRodSensor

The tgRodSensor class outputs the following sensor data:

1. The X, Y, and Z positions of the center of mass of the rod
2. The rotation of the rod: its three Euler angles, via tgRod's getOrientation method. TO-DO: check and see which angles these are, exactly.
3. The mass of the rod. This does not change with timestep, and is provided for backwards compatibility with the original tgDataLogger.

Sensor data from a tgSpringCableActuator using tgSpringCableActuatorSensor

The tgSpringCableActuatorSensor class outputs the following sensor data:

1. The rest length. This is like x0 in F = -k*(x - x0) for the spring in the spring-cable.
2. The current total length of the cable. This is like x in F = -k*(x - x0) for the spring-cable.
3. The tension in the cable. This is like F in F = -k*(x - x0).

1.5.3 Using tgDataLogger2 with YAML models

Copy yamlbuilder/BuildTensegrityModel.cpp to a new folder, and add the code from the section above. For an example of how this is done, refer to the AppSpineKinematicsTest application, under src/dev/ultra-spine/SpineKinematicsTest/AppSpineKinematicsTest.cpp. This file is a copy of BuildTensegrityModel that contains a controller and a tgDataLogger2.

Note that since tgBasicActuator is a tgSpringCableActuator, the tgSpringCableActuatorSensor and its Info class will work fine with the tgBasicActuators created by the YAML builder.

1.5.4 Using tgCompoundRigidSensor

Also new in the tgDataLogger2 infrastructure is a sensor that logs information about compound rigid bodies. Called tgCompoundRigidSensor, it detects tgModels that have been compounded together, using a specific tag that's appended to each model in a compound (see src/tgcreator/tgRigidAutoCompound for more information about this tag hash). This sensor outputs the position and orientation of a compound rigid body.

The position of a compound is defined as the average of the centers of mass of each of its constituent models. Note that this is NOT necessarily the center of mass of the compound itself: for example, if the compound structure contains models of different sizes, the average of the centers-of-mass will not take the different masses into account. See issue #202 for more information. https://github.com/NASA-Tensegrity-Robotics-Toolkit/NTRTsim/issues/202

As of 2017-02-03, the orientation of a compound rigid body is not implemented yet. Currently, an empty string is placed in each of the 'orientation' columns. See issue #203 for more information. https://github.com/NASA-Tensegrity-Robotics-Toolkit/NTRTsim/issues/203

The output of a tgCompoundRigidSensor looks like:

```
0_compound(compound_4cBWDx).X,0_compound(compound_4cBWDx).Y,0_compound(compound_
↪4cBWDx).Z,0_compound(compound_4cBWDx).Euler1,0_compound(compound_4cBWDx).Euler2,0_
↪compound(compound_4cBWDx).Euler3,0_compound(compound_4cBWDx).mass,
-35.9804,15,2.13853e-16,,,,0.195487,
```

The 'mass' parameter is a sum of all the masses of the models in the compound rigid body.

Like the rods and cables, the word "compound" is pre-pended to each column. Currently, the only tag that's written between the parentheses in the heading is the tag that identifies all the models in the rigid compound. This is always the word "compound" with an underscore, then a 6-digit alphanumeric hash that's randomly created for each compound. This hash will (should!) change with each run of the simulator, so your log files will have different headings each time you run it. This is necessary for consistency between simulations of the same type of compound (e.g. a spine vertebra with a specific size) in possibly multiple positions in the same App, or in similar uses between different Apps.

Note also that these compounds are not ordered in any manner. It will be up to you to figure out which compound corresponds with which of your physical objects in the simulation. For example, the AppSpineKinematicsTest application logs vertebrae in some weird order, like 2-1-3-4-6-5. We suggest you look at the compound's position at t=0 and compare that to what you program in your YAML file or model .cpp file.

A suggested fix, if someone wants to implement it, would be to have the sensor output the union of all tags of its constituent models. See issue #204. https://github.com/NASA-Tensegrity-Robotics-Toolkit/NTRTsim/issues/204

Advanced Uses of tgDataManager

This framework allows for other tgDataManagers to be created, not just loggers. For folks doing message passing using JSON, for example, you could create a class like tgMessagePasser that inherits from tgDataManager, and all the sensors and sensor infos will still work.

To create new sensors, you will need to make both a new sensor and a new sensor info class. The sensor info class is what allows a tgDataManager to create the appropriate sensors for tgSenseable objects.

At the moment, only tgModels are sensed (they are the only classes that inherit from tgSenseable.) However, it would be very possible to sense a controller, or something else, by having that inherit from tgSenseable and then by adding it to the data manager using the addSenseable method.

Note that the data manager does NOT create nor destroy its senseable objects. It only stores pointers to those objects, and on setup/teardown and in the destructor, only deletes those pointers not the objects themselves. Remember, tgModel.teardown is handled by tgSimulation.

Other Notes

- You can use the ~ character ("tilde") to represent your HOME directory in the log file name that's passed in to tgDataLogger2.

1.6 Third Party Libraries

This tutorial provides information on the various third party libraries used by NTRT.

1.6.1 JSONcpp

JSON is an acronym of JavaScript Object Notation, and is a standard interface for serialization/de-serialization (getting data in and out of files). Information on JSON can be found at http://json.org/

NTRT depends on JSONcpp. While it is installed by default by setup.sh, users who are interested in JSONcpp's API can find it here: http://open-source-parsers.github.io/jsoncpp-docs/doxygen/index.html

Examples of JSON's use within NTRT can be found in examples/3_prism_serialize More fully featured examples (using JSON's arrays, etc) can be found in dev/btietz/JSONTests

1.6.2 Neural Net

The blog post that is the source of our Neural Network library can be found here: https://takinginitiative.wordpress.com/2008/06/23/c-back-propagation-neural-network-code-v2/

1.7 Contribute to Tutorials

This section contains details on contributing to NTRT's tutorials.

1.7.1 Install Sphinx

NTRT's tutorials are written using Sphinx. So first you'll need to ensure you have Sphinx installed. You can learn more here:

http://sphinx-doc.org/latest/install.html

1.7.2 Getting Started with Sphinx

You can find details on Sphinx's markup on Sphinx's website:

http://sphinx-doc.org/tutorial.html

1.7.3 Modifying NTRT's Tutorials

NTRTSim's tutorials can be found in the docs/source directory.

Once you've made your desired changes, double check that they throw no errors on generation. You can generate HTML output by running **make html** in the **doc** folder. Once you've verified that no errors occur, push your changes to the repository.