# Gear Geometry

Therer are small, medium, and large gears.



Likewise, there are small, medium and large gear "vessels" that can hold the gears. Below are the dimensions for the small gear holder.



Figure Small Gear Vessel Dimensions

Figure Medium Gear Vessel Dimensions

Figure 3 Large Gear Vessel Dimensions



Figure 4 Kit 1



Figure 5 Large Gear Vessel Dimensions

There is a little mixing of units here. The linear dimensions of the vessel are in inches. The linear units of ROS are always in meters. Thus, the offsets are described as values in meter units. While the vessel part is described in inch units. The part as described by 4 3/8 inches is equivalent to .11125 meters.

With much effort the STL mesh that describes this small gear holder was centered at its centroid (or middle of the part for both x and y). Then offsets were computed for each of the four "holders" of the gears. The vision system supplies the centroid of the gear vessel and from this the relative position of each gear holder can be computed. Included in the vision system is the rotation of the vessel, which is part of the computation to determine the gear holder positions. The code to achieve this

slotpose = gearvesselpose \* slotpose

where the pose is a transform matrix describing the position and orientation of the gear vessel and the slot position and orientation. Initially, the slot pose orientation is the identity matrix.

The vision system provides a position (x,y) that we augment with a zero z position and a rotationn around z. Then, the pose of the gear vessel is computed in the tf or transform namespace by the instance rotation provided as a Quaternion and the instance translational position from the origin in the world coordinate system.

tf::Pose Shapes::GetInstancePose(Instance & instance) {

// rotation angle axis + position

return tf::Pose(GetInstanceRotation(instance),

tf::Vector3(instance.x, instance.y, instance.z));

}

The instance rotation is compute by providing the Quaternion constructor a vector around which to rotate (the z axis) and the amount of the rotation around z (i.e., instance.rotation). The vision system provides the instance.rotation value.

tf::Quaternion Shapes::GetInstanceRotation(Instance & instance) {

return tf::Quaternion(tf::Vector3(0.0, 0.0, 1.0), instance.rotation);

}

# JSON Description of the World Model

In the described robot controller, JSON was used to describe the "world model" of the robot(s). As the name, implies the world model contains the model of the objects in the world. The world model of the robots consists of the dynamic objects that are manipulated by the robots (e.g., the gears) and static objects that are part of the scene (e.g., walls).

JSON was used to describe the objects. JSON is an open-standard format that uses human-readable text to exchange hierarchical data represented in a tree consisting of attribute–value pairs. The JSON format is syntactically identical to the code for creating JavaScript objects as well as a subset of YAML. Further, there a "Non-SQL" data bases that use JSON as the schema representation and for defining data.

The JSON structure is a hierarchical tree of nodes, branches, children, without cycles. Every JSON tree has exactly one root element. All other nodes are contained under this single root element. Nodes can be nested within another node to establish parent/child relationships. A branch is then described as a node and all the children under the node, and all the childrens children under that, ad infinitum.

implicit root with children: "child1" and "child2"

{ "child1": { "attr11": "value11" },

"child2": { "attr21": "value21" }

}

"Parts" and "Instances" are the two primary JSON branches under the root. At this point in time, the boost Property Tree parses the JSON (as well as traditional ini format) for syntax validation and tree manipulcation API that was used to translate and interpret the JSON as a Shapes data repository with Part model containing Shape Geometry and other attributes, and instances which are instantiations of the part models. For example below is the JSON to describe the small and medium size gears and the vessels to hold the gears:

{

**"parts"**:{

**"sku\_small\_gear"**:{  },

**"sku\_small\_gear\_vessel"**:{  },

**"sku\_medium\_gear"**:{  },

**"sku\_medium\_gear\_vessel"**:{  }

   },

**"instances"**:{

**"sku\_small\_gear1"**:{  },

**"sku\_small\_gear2"**:{  },

**"sku\_small\_gear3"**:{  },

**"sku\_small\_gear4"**:{  },

**"sku\_medium\_gear1"**:{  },

**"outline\_sku\_small\_gear\_vessel1"**:{  },

**"sku\_small\_gear\_vessel1"**:{  },

**"sku\_medium\_gear\_vessel1"**:{  }

   }

}

In the controller, the instances are drawn in ROS as RVIZ markers. The instance name given by the child branches under the instance branch (i.e., sku\_small\_gear1, sku\_sm sku\_small\_gear\_vessel1all\_gear2, sku\_small\_gear3, sku\_small\_gear4, sku\_medium\_gear1, outline\_sku\_small\_gear\_vessel1, and sku\_medium\_gear\_vessel1) is used as the name in the controller. Gear "type" can be of any type of gear: small, medium or large. Gear trays can contain a combination of gear type holders. The distinction used herein, is that a gear vessel can hold all of the same gear types (e.g., sku\_small\_gear\_vessel1 only can hold small gears), while a kit can contain a combination of gear types (e.g., sku\_kit2 can hold small or medium or large gears). Each part describes the geometry and part of the containing slots for gears.

## Gear Tray

Below is a figure showing the STL visualization for a medium gear vessel/tray that will hold the medium gears. The gear vessel design was specified in inches and had a few issues. First, the centroid of the vessel part was not in the middle of the XY plane. Second, the part's bottom in the Z axis was negarive, so if it was placed on a flat surface, it would "in" the surface. Using FreeCAD, the part was centered in the middle, and the Z axis was raised so that all z values were greater than or equal to zero.

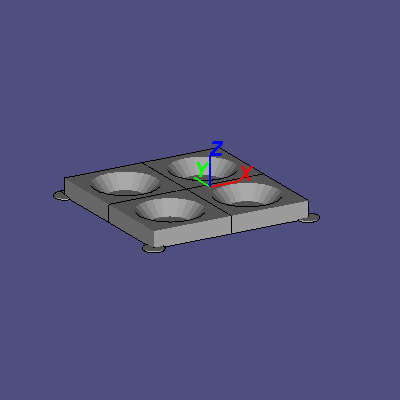


Figure STL Visualization of the Medium Gear Vessel

The medium gear vessel has 2 components: the part description and instances of the part. The part description describes attributes that all instances of the part exhibit – part definition (including type, metatype, subtype, color) with specific types having further detail about its definition (e.g., a mesh has the location of the STL file describing the part for RVIZ.) A part "instance" uses it metatype to reference the part description, but adds items for position and rotation, which can be supplied by a vision system. There can be multiple "instances" of any "part". For example, 4 small gear instances can reference the small gear part design.

Walking through the JSON, the medium gear vessel JSON "part" description is shown below:

**"sku\_medium\_gear\_vessel"**:{    
         **"type"**:"primitive",  
         **"metatype"**:"mesh",  
         **"subtype"**:"solid",  
         **"file"**:"file:///.../nistfanuc\_ws/src/nist\_robotsnc/worldmodel/medium\_gear\_holder\_centeredZposge0.stl",  
         **"scale"**:0.0254,  
         **"color"**:"CYAN",  
         **"centroid"**:"(4.5in, 4.5in), slot (2.25in, 2.25in)",  
         **"contains"**:{ . . . }  
      }

Each part branch contains children: centroid, type, metatype, subtype, and color provide the information that describes the part to the controller and the simulation visualization. Specific types of parts (in this case mesh) have different additional child branches to further describe the part profile. In the case of mesh, these include the "file" which describes the location of the STL file and scale which describes the scaling factor to apply to the

* "centroid" child is used for documentation purposes, and is actually not used by the controller.
* "type" describes the part type, in this case a vessel, i.e., a holder of gears,
* "metatype" describes the type of scene object, in this case, an STL mesh,
* "subtype" describes a refinement of the type, either solid or outline, of the equivalent graphical object. Some types cannot support all subtype, for example, a mesh cannot be an outline.
* "color" is a child branch of all the parts, and in this case provides the mesh color (i.e., "CYAN"), which will be translated from a string into an ROS RVIZ equivalent representation. Color is used to describe all instances, but can be overridden if an instance color branch is defined.
* "contains" is a branch that describes the slots where gears can be inserted. Each instance of this holder provides a state description (empty or full) of subbranches in its instance.

A mesh part "type" contains these additional child attributes:

* "file" is a child branch specific to an metatype mesh, and provides a location of the STL mesh file,
* "scale" is a child branch specific to an metatype mesh that provides the scale factor to use for displaying the mesh, in this case the scale factor is the conversion from inch to meter, i.e., 0.0254.

Below the JSON for the part description child contains branch is expanded. The type of the part has a "contains" branch, which is a holder – it contains slots to hold gears. The "contains" branch has a child "metatype", which is a "name" in the "part" sub branch of the root that can fit into the contained slot. Included the "contains" description is the location on the part (an offset from the centroid of the part). The child JSON branch "position" describes a list which contains the x,y,z offset from the centroid. The child JSON branch "rotation" describes a rotation of the slot, if any. The child JSON branch "state" is a branch that must be in the instance definition. For an instance, the "state" is either "filled" or "empty". Thus, if it is "empty" it is able to hold a "free" gear.

"**contains**":{  
            **"slot1"**:{    
               **"type"**:"holder",  
               **"metatype"**:"sku\_medium\_gear",  
               **"position"**:[  0.0396, 0.0396, 0.0 ],  
               **"rotation"**:0,  
               **"state"**:"tobedefined"  
            },  
**. . .**

**"slot4"**:{    
               **"type"**:"holder",  
               **"metatype"**:"sku\_medium\_gear",  
               **"position"**:[  -0.0396, -0.0396, 0.0 ],  
               **"rotation"**:0,  
               **"state"**:"tobedefined"  
            }

}

Instances is a subbranch of the root JSON which contains the objects in the world model.

{

"instances":{

"sku\_small\_gear1":{

"type":"gear",

"metatype":"sku\_small\_gear",

"color":"RED",

"position":[ 0.22989,-0.25126, 0.0 ],

"rotation":-0.82,

"state":"free"

}

. . .

}

}

A simple instance will be described which references a part model for its representation.

* "type" is the category of the instance, in this case a gear. It could be a vessel or outline.
* "metatype" gives the name in the part branch, or the part geometry model name.
* "color" is provides the object color (i.e., "RED"), which will be translated from a string into an ROS RVIZ equivalent representation. Color is used to describe all parts, but can be overridden if an instance color branch is defined.
* "position" describes the x,y,z translation of the part in the world coordinate system. The list (i.e., numbers between the brackets "[]") is parsed into a tf::Vector3 representation. When combined with the "rotation" it describes a 4D representation that is then converted into a 6D posed, i.e., tf::Pose. The position is typically provided by a vision system that can identify the position and rotation of the parts.
* "rotation" describes rotation around the Z axis of the instance. Only the Z axis is required as the parts are all flat on the bottom and assumed to be resting flat on a surface. However, the part can be turned and the "rotation" attribute describes this amount. When combined with the "position" attribute it can be represented as a 6D pose (assuming zero rotation in X and Y axes.) The rotation is typically provided by a vision system that can identify the position and rotation of parts.
* "state" describes the part or holder slot stateness. We assume initially that all parts are "free" when in fact the part could already be placed in a holder slot. This would only require simple mathematics to see if the part pose matches a holder container slot pose, however, the part may not be discernible when contained in a tray slot. Likewise, gear holder slots may be "filled" but will always be assumed to be "empty". It is TBD how this will be handled.

Of note, the position translation is described in the world coordinate system, but a transform is applied to bring the parts into a robot coordinate system, since there are two robots and the part instances are described from a Fanuc robot centered at (0,0,0) in the world coordinate system.

As another example, a small gear vessel JSON will be reviewed. Below is the JSON describing an instance (i.e., sku\_small\_gear\_vessel1) of a small gear vessel in the world model. The type is a "holder" indicating that it is holder of gears. The metatype (i.e., " sku\_small\_gear\_vessel" ) references a part model to indicate how the instance will be drawn. The subtype is solid indicating that it is not an outline. The position is derived from a vision system and in combination with the "rotation" provides the pose of the gear vessel (position and orientation). The "contains" child describes the state for each holder slot 1..4. As stated, initially the states of all holder slots is empty as it is not clear how an initial placement of a gear in a holder could be detected.

"sku\_small\_gear\_vessel1" :

{

"type" : "holder",

"metatype": "sku\_small\_gear\_vessel",

"subtype" : "solid",

"position":[ 0.3155,0.13662, 0.0],

"rotation": -1.4,

"contains":

{ "slot1": { "state" : "empty"} ,

"slot2": { "state" : "empty"} ,

"slot3": { "state" : "empty"} ,

"slot4": { "state" : "empty"}

}

},

Once a gear holder slot has been filled, the "state" changes to "filled" to prevent another gear from being placed into the slot.

# Gear Placement Algorithm

Setup all instance  
 of gear and tray

Find instance of gear  
Sm, med, lg

Find matching  
Sm, med, lg gear slot in tray

IN

Display

None

None