JSC-61777-models/interactions/contact

# JSC Engineering Orbital Dynamics Contact Model

**Simulation and Graphics Branch (ER7)**
**Software, Robotics, and Simulation Division**
**Engineering Directorate**

## Package Release JEOD v5.1

## Document Revision 1.2
## July 2023

**National Aeronautics and Space Administration**
**Lyndon B. Johnson Space Center**
**Houston, Texas**

JSC-61777-models/interactions/contact

# JSC Engineering Orbital Dynamics
# Contact Model

**Document Revision 1.2**
**July 2023**

**Christopher Thebeau**

**Simulation and Graphics Branch (ER7)**
**Software, Robotics, and Simulation Division**
**Engineering Directorate**

**National Aeronautics and Space Administration**
**Lyndon B. Johnson Space Center**
**Houston, Texas**

## Abstract

The Contact Model is designed to provide a way for dynamic bodies created with the Dynamic Body Model [2] to interact beyond mere mass attachment. Without the Contact Model, dynamic bodies are completely separate entities that can not influence each other's state if they are not attached via a mass tree [6]. Using the Contact Model surfaces can be defined which generate forces when they are inter-penetrated (contacted) by surfaces on other dynamic bodies. The basic infrastructure provided by the Contact Model is designed to be extended allowing for the simulation of detailed docking, grappling, and birthing operations.

# Contents

# Chapter 1

# Introduction

## 1.1 Purpose and Objectives of the Contact Model

The Contact Model is designed to provide a way for dynamic bodies created with the Dynamic Body Model [2] to interact beyond mere mass attachment. Without the Contact Model, dynamic bodies are completely separate entities that can not influence each other's state if they are not attached via a mass tree [6]. Using the Contact Model surfaces can be defined which generate forces when they are inter-penetrated (contacted) by surfaces on other dynamic bodies. The basic infrastructure provided by the Contact Model is designed to be extended allowing for the simulation of detailed docking, grappling, and birthing operations.

## 1.2 Context within JEOD

The following document is parent to this document:

- *JSC Engineering Orbital Dynamics* [3]

The Contact Model forms a component of the interactions suite of models within JEOD v5.1. It is located at models/interactions/contact.

## 1.3 Document History

| Author | Date | Revision | Description |
|---|---|---|---|
| Christopher Thebeau | 04JAN2012 | 1.2 | model redesign |
| Christopher Thebeau | 24SEP2010 | 1.1 | initial version |
| Christopher Thebeau | 23JUL2010 | 1.0 | draft version |

## 1.4 Document Organization

This document is formatted in accordance with the NASA Software Engineering Requirements Standard [4].

The document comprises chapters organized as follows:

**Chapter 1: Introduction** -This introduction describes the objective and purpose of the Contact Model.

**Chapter 2: Product Requirements** -The requirements chapter describes the requirements on the Contact Model.

**Chapter 3: Product Specification** -The specification chapter describes the architecture and design of the Contact Model.

**Chapter 4: User Guide** -The user guide chapter describes how to use the Contact Model.

**Chapter 5: Inspections, Tests, and Metrics** -The inspections, tests, and metrics describes the procedures and results that demonstrate the satisfaction of the requirements for the Contact Model.

# Chapter 2

# Product Requirements

*Requirement Contact_1: JEOD Requirements*

**Requirement:**
This model shall meet the JEOD project requirements specified in the JEOD v5.1 top-level document [3] .

**Rationale:**
This model shall, at a minimum, meet all external and internal requirements applied to the JEOD v5.1 release.

**Verification:**
Inspection

*Requirement Contact_2: Contact Between Surfaces in 3D Space*

**Requirement:**
The Contact Model shall determine if two or more 3D surfaces are in contact and resolve the resultant forces utilizing user defined dynamics.

**Rationale:**
Contact between surfaces is the core functionality of the Contact Model.

**Verification:**
Inspection, Test

*Requirement Contact_3: Provide a Default Contact Force Model Based on Springs*

**Requirement:**
The Contact Model shall provide default contact dynamics functionality based on simple springs.

**Rationale:**

A basic contact force generation model needed to be provided with the Contact Model and has been in the form of a simple spring.

**Verification:**

Inspection, Test

*Requirement Contact_4: Limit Circular Contact*

**Requirement:**

The Contact Model shall limit contact determination to surfaces that do not reside on the same vehicle or attached vehicles.

**Rationale:**

In order to avoid potential circular contact or attachments, which are not allowed in JEOD, surfaces residing on the same body or tree are excluded from contacting each other.

**Verification:**

Inspection, Test

*Requirement Contact_5: Contact Model Extensibility*

**Requirement:**

The Contact Model shall be extensible.

**Rationale:**

Other objects and classes should be able to derive from components of the Contact Model.

**Verification:**

Inspection

# Chapter 3

# Product Specification

## 3.1 Conceptual Design

The Contact Model was designed to provide a way for dynamic bodies to interact. This interaction is called “contact” since forces are applied when surfaces defined on those dynamic bodies interpenetrate. The uses for the Contact Model are many-fold, encompassing simple vehicle collisions to complex docking dynamics. In order to accommodate that range of use the Contact Model was designed with extension as an expectation.

Two basic considerations drove the design of the Contact Model. The first is that the Contact Model should be an extension of the Surface Model [5]. The second is that a user should only have to define the surfaces and interaction types ( allowable pairs of surfaces ) that will generate forces. The user is not required to define specific pairs of interacting surfaces but that can done if desired.

## 3.2 Mathematical Formulations

### 3.2.1 Springs

The default force generation function included in the Contact Model is based on simple springs. This takes the form of

$$F = -kx - cv \tag{3.1}$$

Where k is the spring constant, x is displacement or penetration distance, c is the damping coefficient, and v is relative velocity.

### 3.2.2 Vector Calculations

The geometry calculations in the Contact Model rely on the Mathematical Functions Model [7] and the equations will not be repeated here. Most of the the vector equations are utilized in the Contact Model; including vector transforms, scaling, addition, and subtraction.

## 3.3 Interactions

### 3.3.1 Surface Model

As mentioned previously the Contact Model is an extension of the Surface Model [5]. Below is a listing of the Contact Model classes that are derived directly from Surface Model classes.

| **Base Surface Model Class** | **Derived Contact Model Class** |
|---|---|
| InteractionFacet | ContactFacet |
| FacetParams | ContactParams |
| InteractionSurface | ContactSurface |
| InteractionFacetFactory | PointContactFacetFactory |
| InteractionFacetFactory | LineContactFacetFactory |

In addition new classes were created in the Surface Model to support the special geometries required by the Contact Model. The FlatPlateCircular class is used to generate spherical contact surface about a point. The Cylinder class is used to generate a rod like structure with half spheres encasing the ends about a line.

## 3.4 Detailed Design

### 3.4.1 API

Follow this link for the *Contact Model API* [1].

### 3.4.2 Contact Class

The Contact class acts as a manager for the contact interaction and is a concept not seen in other extensions of the surface model such as aerodynamics and radiation pressure. The purpose of the Contact class is to construct a list, called contact_pairs, of ContactPair objects which link two ContactFacet objects together and calculates / stores the relative state between them. During run time the Contact class loops though this list so each pair of facets can be checked to see if contact has occurred.

The Contact.contact_pairs list is populated in two stages during initialization. Each ContactFacet object that registers with an instance of the Contact class creates a ContactPair object with itself as the subject but absent a target. After all contact facets are registered the Contact class method Contact.initialize_contact does a double loop through the incomplete contact pairs to construct new instances of the ContactPair class with both a subject and a target. Several checks are performed to determine if a pair of the two contact facets can be constructed. Since the ContactPair class is a pure virtual base class, as is the ContactFacet class, the first check is to determine if a derived class exists that can contain the specific types of contact facets. An additional check is performed by comparing each facets ContactParam type to the list of pair interactions that is also contained in the Contact class. If a PairInteraction object defining the interaction between the contact facets' parameter types doesn't exist then no pair is created.

At runtime the Contact class loops through the pairs contained in the contact_pairs list. After inquiring to ensure that the pair is complete (has a subject and target), active, and optionally that the facets are in range to interact the Contact class asks the contact pair to determine if contact has occurred.

### 3.4.3 ContactPair Class

The ContactPair class is a virtual base class, and its derived classes perform most of the work in the Contact Model. It is a ContactPair class that determines if and when two ContactFacets interact. The base class contains references to two contact facets, a subject and a target. It also contains a RelativeDerivedState object which is used to calculate and store the relative state between the subject and target ContactFacets. In addition each ContactPair contains a reference to a PairInteraction object that defines force calculation method in the event of contact. During initialization the ContactPair class constructs the relative state between its subject and target facets. During runtime the virtual method in_contact is used by all derived classes of the ContactPair class to determine if their subject and target are in contact with each other. If the in_contact method determines that contact has occurred then appropriate forces are generated using the PairInteraction associated with the ContactPair. All unique pairs of ContactFacet subclass types require a specific implementation of ContactPair. For example the Contact Model contains two derived classes extending the base ContactFacet class, but contains three ContactPair derived classes to deal with the possible pairings between these contact facets. Each derived ContactFacet class can interact with others of the same type which requires two distinct ContactPair implementations. For them to interact with each other we need a third ContactPair implementation.

### 3.4.4 ContactSurface Class

A contact surface is a collection of contact facets that are contained on the same dynanmic body (DynBody [2]). It is where the forces generated on individual facets will be summed so they can all be applied to the dynamic body. As previously mentioned the ContactSurface class is a subclass of an InteractionSurface from the Surface Model [5], however the stipulation that it apply only to one DynBody instance is a limitation on the standard behavior associated with a surface as defined in the Surface Model.

### 3.4.5 ContactFacet Class

A contact facet must contain a reference to a dynamic body (DynBody [2]). At its most basic a contact facet is a vehicle point (BodyRefFrame [2]) on a specific dynamic body. From the position and normal of the vehicle point, the geometry of the specific type of facet is constructed and used by the ContactPair class to determine when a contact facet intersects another contact facet. During initialization specific derived classes of ContactFacet determine whether specific ContactPairs can be created. Extensibility of the types of surfaces available to the Contact Model is achieved by creating new types of contact facets. It is also necessary to create new ContactPair classes to define the interaction between the new contact facet and others of its type or any of the preexisting types of contact facets.

### 3.4.6 ContactParams Class

The ContactParams class add no new functionality to the base FacetParams class from the Surface Model [5]. The name associated with the parameters is used to match pairs of contact facets with the correct PairInteraction object.

### 3.4.7 PairInteraction Class

The PairInteraction class is a pure virtual base class and is intended to be extended to define multiple force calculation methods. Extensions of this class are expected to contain all the parameters they need to calculate contact forces. In addition, an inheriting class must define the pure virtual method calculate_forces which takes all relevant geometry information that it might need to calculate contact forces. The calculate_forces function is called from a contact pair once its geometry calculations have determined that contact between facets has occurred. Each simulation defined PairInteraction subclass must be registered with an instance of the Contact class. When ContactPair instances are created the ContactFacet ContactParams types must match with a registered PairInteraction or no pair is created. A reference to the a specific PairInteraction is stored in each successfully created ContactPair.

### 3.4.8 Springs

The default force generation parameters included in the Contact Model are based on simple springs, equation 3.1.

By default this spring force is designed to oppose the penetration of contact, but a sign reversal could invert this behavior making the spring forces applicable to simulations requiring latching.

## 3.5 Inventory

# Chapter 4

# User Guide

## 4.1 Instructions for Simulation Users

### 4.1.1 General Description

The verification simulation SIM_contact_T10 and its associated input files are used in the examples below. They demonstrate the use of the Contact Model with a Trick 10 series simulation executive. The simulation SIM_contact contains an example of using the Contact Model with a Trick 7 series simulation executive.

### 4.1.2 Pair Interactions

Central to having surfaces actually contact each other is the creation of instances of the Pair_Interaction class. Each such object links two instances of Contact_Params by name.

```
execfile( "Contact_Modified_data/contact/pair_interaction.py")
set_contact_interaction(contact)
```

As seen below a pair interaction contains the names of the parameters that will produce interaction as well as the values need to calculated forces from the contact should it occur. In this example we are using the Contact Model provided spring model, so we define the spring constant (spring_k), damping coefficient (damping_b), and the coefficient of friction (mu).

```
def set_contact_interaction(contact_reference) :

  pair_interaction_local = trick.SpringPairInteraction()
  pair_interaction_local.thisown = 0

  pair_interaction_local.params_1 = "steel"
  pair_interaction_local.params_2 = "steel"

  pair_interaction_local.spring_k = trick.attach_units( "lbf/in",20.0)
```

```
    pair_interaction_local.damping_b = trick.attach_units( "lbf*s/in",0.4)
    pair_interaction_local.mu = 0.05

    contact_reference.contact.register_interaction(pair_interaction_local)

    return
```

### 4.1.3 Contact Parameters

All vehicles in the SET_test runs contained in SIM_contact_T10 use the same contact parameters. The inclusion of contact parameters takes the following form.

```
  execfile( "Contact_Modified_data/contact/contact_params.py")
  set_contact_params(veh1_dyn)
```

Contact parameters are purposely very simple and are used to control the types of possible interactions. In this case all contact facets will have parameters of the name "steel" and they have the potential for contact if there exists an instance of the Pair_Interaction class that defines a "steel" on "steel" interaction like the example above. Creating and naming an instance of the ContactParams class is show below.

```
  def set_contact_params(sv_dyn_reference) :

    contact_params_local = trick.ContactParams()
    contact_params_local.thisown = 0

    contact_params_local.name = "steel"

    sv_dyn_reference.facet_params = contact_params_local
    sv_dyn_reference.contact_surface_factory.add_facet_params(contact_params_local)


    return
```

### 4.1.4 Contact Surface and Facets

Now that there are contact parameters available to be associated with the contact facets and defined pair interactions to generate valid pairings between them, the contact facets are constructed.

```
  execfile( "Contact_Modified_data/contact/point_facet.py")
  set_contact_point_facet(veh1_dyn, "veh1")
```

This is an example of adding one facet. However, implementing multiple facets would only involve modifying the python function to accept more parameters for the position, normal, parameter name, etc. The type of facet being created below is of the FlatPlateCircular class, so in addition to the required fields of “name,” “position” and “param_name” you must also specify the “normal” and “radius.” Since this facet is intended for use in the Contact Model a “mass_body_name” must also be specified, and in this case the associated mass_body shares the same name as the surface_model.struct_body_name.

```
def set_contact_point_facet(sv_dyn_reference, SV_NAME) :

  exec('sv_dyn_reference.surface_model.struct_body_name = "' + SV_NAME + '"')

  fp = trick.FlatPlateCircular()
  fp.thisown = 0

  exec( 'fp.name = "' + SV_NAME + '_facet"')
  fp.position  = trick.attach_units( "m",[ 0.0 , 0.0, 0.0 ])
  fp.normal  = [ 1.0 , 0.0 , 0.0 ]
  fp.radius  = trick.attach_units( "m",1.0)
  fp.param_name = "steel"
  exec('fp.mass_body_name = "' + SV_NAME + '"')

  sv_dyn_reference.facet_ptr = fp
  sv_dyn_reference.surface_model.add_facet(fp)

  return
```

### 4.1.5 Logging

Logging the internal details of the Contact Model, such as the forces a specific facets, is difficult due to the number of base class pointer references to derived classes. However, it is straightforward to log the forces and torques collected by contact on the entire contact surface.

```
def log_contact_data ( log_cycle ) :
   # Import the JEOD logger
   import sys
   import os
   sys.path.append ('/'.join([os.getenv("JEOD_HOME"), "lib/jeod/python"]))
   import jeod_log

   # Create the logger.
   logger = jeod_log.Logger (data_record.drd)

   logger.open_group (1, "contact_data")

   logger.log_scalar(
      ("veh1_dyn.body.composite_properties.mass",
       "veh2_dyn.body.composite_properties.mass"))
```

```
logger.log_vector(
   ("veh1_dyn.body.composite_body.state.trans.position",
    "veh1_dyn.body.composite_body.state.trans.velocity",
    "veh1_dyn.contact_surface.contact_force",
    "veh1_dyn.contact_surface.contact_torque",
    "veh2_dyn.body.composite_body.state.trans.position",
    "veh2_dyn.body.composite_body.state.trans.velocity",
    "veh2_dyn.contact_surface.contact_force",
    "veh2_dyn.contact_surface.contact_torque"))

logger.close_group()
```

## 4.2 Instruction for Simulation Developers

The examples given in this sections come from the S_define for SIM_contact_T10 and describe the use of the Contact Model with Trick 10 simulation executive.

### 4.2.1 Vehicle Simulation Object Additions

The majority of the definitions and jobs needed for the Contact Model are best placed in the simulation objects for the vehicles. In general each vehicle will contain its own unique instance of the SurfaceModel class which will contain all the facets and parameters associated with the vehicle. The following lines set up an instance of the SurfaceModel class and a pointer to add facet parameters to it.

```
SurfaceModel              surface_model;
FacetParams             * facet_params;
```

Then it is time to instantiate a ContactSurface object and a factory to create it.

```
ContactSurface            contact_surface;
ContactSurfaceFactory     contact_surface_factory;
```

This is followed by the definition of several variables needed by the initialization routines and the input file. There is a facet pointer which is used to add facets to SurfaceModel. There are also pre-declarations of the types of contact facets that will be created in the simulation input files, contact parameters and a specific type of pair interaction.

```
FlatPlateCircular     * flat_plate_circular;
Cylinder              * cylinder;
ContactParams * contact_params;
SpringPairInteraction * spring_pair_interaction;
Facet * facet_ptr;
unsigned int integer;
```

Next we get to the initialization jobs that should occur in the following order in the S_define:

- SurfaceModel.initialize_mass_connections - creates the linkage between a Facet and a Mass or in the case of the Contact Model, a facet and a dynamics body,
- ContactSurfaceFactory.create_surface - creates a ContactSurface from the Surface base class, which means that it creates a ContactFacet from the information contained in each Facet base class,
- ContactManager.register_contact - registers a set of contact facets with the contact manager, with the easiest way being to use the array of contact facets in the contact surface,

```
  P_BODY ("initialization") surface_model.initialize_mass_connections(
     internal_dynamics->dyn_manager );
  P_BODY ("initialization") contact_surface_factory.create_surface(
     & surface_model,
     & contact_surface);
   P_BODY  ("initialization") internal_contact->contact.register_contact(
      contact_surface.contact_facets,
      contact_surface.facets_size);
```

The only recurring job in the vehicle simulation object from the contact model is a call to collect_forces_torques in each contact surface. This collects forces and torques from every contact facet associated with the contact surface and sums them to produce the total forces from contact.

```
  P_DYN ("derivative") contact_surface.collect_forces_torques();
```

### 4.2.2 Contact Simulation Object

Unlike some of the other Surface Model derived interaction models, the Contact Model requires a separate manager to track the contact facets so a determination of contact forces can be accomplished. The basic form of this simulation object will contain an instance of the Contact class. This manager object should be initialized after all the vehicles have completed the registration of their contact surfaces. It is the function Contact.initialize_contact that creates pairs based on the contact facets, parameters, and pair interactions that have been registered with the manager. The call to the method check_contact is the main driver for the entire Contact Model. It is this function that begins the process of determining if contact has occurred during runtime.

```
##include "interactions/contact/include/contact.hh"

class ContactSimObject: public Trick::SimObject {

   public:
    Contact contact;

// Instances for matching to other sim objects:
    DynamicsSimObject * internal_dynamics;

//Constructor
    ContactSimObject(
      DynamicsSimObject  & ext_dynamics) {

      internal_dynamics = & ext_dynamics;

      P_DYN  ("initialization") contact.initialize_contact(
          & internal_dynamics->dyn_manager);
      //
      // Derivative class jobs
      //
      P_DYN   ("derivative") contact.check_contact();
    }

  private:
    ContactSimObject (const ContactSimObject&);
    ContactSimObject & operator = (const ContactSimObject&);

};
ContactSimObject contact(dynamics);
```

## 4.3 Instruction for Model Developers

The Contact Model was designed with the expectation of user extension. Some extensions such as adding new types of contact facets or new pair interactions are relatively easy. Some such as extending the contact base class to include a completely new concept such as latching or ground contact may be challenging depending on the complexity desired.

### 4.3.1 Pair Interactions

The virtual base class for interaction and force generation in the Contact Model is called PairInteraction. This class contains a pure virtual function, called force_torque, used to calculate the forces and torques a specific facet produces when in contact. This method takes in information determined from the geometry of the interacting facets, but not knowledge of the specific facet geometries. In the released version of the Contact Model, one derived class of PairInteraction is included which is the SpringPairInteraction class. This basic functionality applies forces and torques assuming that the facets resist inter-penetration using the dynamics of simple springs. The PairInteraction class can be extended to simulate whatever force generation dynamics the user desires.

### 4.3.2 Contact Facets

It is possible to add new types of contact facets to the Contact Model. All types of contact facets are derived from the base class ContactFacet. To add a new type it is necessary to create a new class derived from ContactFacet. Then one must create a matching ContactFacetFactory for the new type of ContactFacet. If the new type of ContactFacet can be defined by the geometry of an existing facet in the Surface Model, then there is no need to create a matching facet there. If not, a new derived class of the Surface Model Facet class will need to be created in the Surface Model [5]. For example the PointContactFacet makes use of the Surface Model class CircularFlatPlate as its base. A CircularFlatPlate contains a point and normal, but unlike a FlatPlate it also contains a radius which can be used as the interaction distance needed to properly define a PointContactFacet.

Now that a new type of ContactFacet has been created it is necessary to define how it will interact with types that already exist if you want contact to occur between those contact facets. Unless this is done a ContactPair between the new type and old will not be created by the Contact class, which can be a good way to limit list size and control the resource use of the Contact Model.

### 4.3.3 Contact

The manger class, called Contact, is also designed with extension in mind. This JEOD release contains an example of such an extension in SIM_ground_contact located in the verif directory of Contact Model. This simulation includes two new models. One is a simple radius of the planet base terrain model called "terrain". The other called "contact_ground" is an extension of the Contact Model to include contact between vehicle facets and the surface of a planet. The treatment of the ground contact problem in this example simulation and contact ground model is merely demostrative, for it assumes a non rotating earth, contact at the radius of the planet, and other simplifications. However, it does give a powerful demonstration of the extensibility of the Contact Model.

# Chapter 5

# Inspections, Tests, and Metrics

## 5.1 Inspections

*Inspection Contact_1: Top-level inspection*

This document structure, the code, and associated files have been inspected, and together satisfy requirement Contact_1.

*Inspection Contact_2: Extensibility inspection*

Examples of the extensibility of the Contact Model are numerous in the released version of JEOD and Contact_5 is easily verified.

- The Contact class is extended for ground contact in the example simulation SIM_ground_contact_T10.
- The SpringPairInteraction class is an extension of the PairInteraction class.
- The PointContactFacet class and the LineContactFacet class extend the ContactFacet class.

*Inspection Contact_3: Circular contact inspection*

The code was inspected and used in non-JEOD released simulations and the non-circular nature of the Contact Model was verified. Contact_4.

## 5.2 Tests

*Test Contact_1: Two-Sphere Contact: Aligned*

**Purpose:**
The purpose of this test is to see the results of two point contact facets contacting and

rebounding. For this test the contact facets were aligned (no translation or rotation offsets) along the line of approach.
SIM directory: models/interactions/contact/verif/SIM_contact_T10
Run directory: RUN_point

**Requirements:**
By passing this test, the Contact Model partially satisfies requirements Contact_2 and Contact_3.

**Procedure:**
In this test, the simulation was run and the position, velocity, and contact forces of the two bodies was examined. The principle test being that the facets make contact at the appropriate range and produce forces in the correct directions.

**Results:**
The contact occurred at the correct distance. The two bodies then separated and the forces generated were equal and opposite.

*Test Contact_2: Two-Sphere Contact: Inversed*

**Purpose:**
The purpose of this test is to see the results of two point contact facets contacting and rebounding. For this test the contact facets were rotated 180 degrees about line of approach, but had not translational offset about that line.
SIM directory: models/interactions/contact/verif/SIM_contact_T10
Run directory: RUN_point_inverse

**Requirements:**
By passing this test, the Contact Model partially satisfies requirements Contact_2 and Contact_3.

**Procedure:**
In this test, the simulation was run and the position, velocity, and contact forces of the two bodies was examined. The principle test being that the facets make contact at the appropriate range and produce forces in the correct directions.

**Results:**
The contact occurred at the correct distance. The two bodies then separated and the forces generated were equal and opposite.

*Test Contact_3: Two-Sphere Contact: Off Center*

**Purpose:**
The purpose of this test is to see the results of two point contact facets contacting and rebounding. For this test the contact facets were rotated 180 degrees about and translationally offset from the line of approach.
SIM directory: models/interactions/contact/verif/SIM_contact_T10
Run directory: RUN_point_off_center

**Requirements:**

By passing this test, the Contact Model partially satisfies requirements Contact_2 and Contact_3.

**Procedure:**

In this test, the simulation was run and the position, velocity, and contact forces of the two bodies was examined. The principle test being that the facets make contact at the appropriate range and produce forces in the correct directions.

**Results:**

The contact occurred at the correct distance. The two bodies then separated and the forces generated were equal and opposite.

*Test Contact_4: Two-Cylinder Contact: Aligned*

**Purpose:**

The purpose of this test is to see the results of two line contact facets contacting and rebounding. For this test the contact facets were aligned (no translation or rotation offsets) along the line of approach.
SIM directory: models/interactions/contact/verif/SIM_contact_T10
Run directory: RUN_line

**Requirements:**

By passing this test, the Contact Model partially satisfies requirements Contact_2 and Contact_3.

**Procedure:**

In this test, the simulation was run and the position, velocity, and contact forces of the two bodies was examined. The principle test being that the facets make contact at the appropriate range and produce forces in the correct directions.

**Results:**

The contact occurred at the correct distance. The two bodies then separated and the forces generated were equal and opposite.

*Test Contact_5: Two-Cylinder Contact: Inverse*

**Purpose:**

The purpose of this test is to see the results of two line contact facets contacting and rebounding. For this test the contact facets were rotated 180 degrees about the line of approach.
SIM directory: models/interactions/contact/verif/SIM_contact_T10
Run directory: RUN_line_inverse

**Requirements:**

By passing this test, the Contact Model partially satisfies requirements Contact_2 and Contact_3.

**Procedure:**

In this test, the simulation was run and the position, velocity, and contact forces of the two bodies was examined. The principle test being that the facets make contact at the appropriate range and produce forces in the correct directions.

**Results:**

The contact occurred at the correct distance. The two bodies then separated and the forces generated were equal and opposite.

*Test Contact_6: Two-Cylinder Contact: Arbitrary*

**Purpose:**

The purpose of this test is to see the results of two line contact facets contacting and rebounding. For this test the contact facets were rotated at arbitrary angles about and translationally offset from the line of approach.
SIM directory: models/interactions/contact/verif/SIM_contact_T10
Run directory: RUN_line_arbitary

**Requirements:**

By passing this test, the Contact Model partially satisfies requirements Contact_2 and Contact_3.

**Procedure:**

In this test, the simulation was run and the position, velocity, and contact forces of the two bodies was examined. The principle test being that the facets make contact at the appropriate range and produce forces in the correct directions.

**Results:**

The contact occurred at the correct distance. The two bodies then separated and the forces generated were equal and opposite.

*Test Contact_7: Sphere Cylinder Contact*

**Purpose:**

The purpose of this test is to see the results of a point contact facet and a line contact facet contacting and rebounding.
SIM directory: models/interactions/contact/verif/SIM_contact_T10
Run directory: RUN_line_point

**Requirements:**

By passing this test, the Contact Model partially satisfies requirements Contact_2 and Contact_3.

**Procedure:**

In this test, the simulation was run and the position, velocity, and contact forces of the two bodies was examined. The principle test being that the facets make contact at the appropriate range and produce forces in the correct directions.

**Results:**

The contact occurred at the correct distance. The two bodies then separated and the forces generated were equal and opposite.

## 5.3 Requirements Traceability

| **Requirement** | **Inspection or test** |
|---|---|
| Contact_1 - Top-level requirements | Insp. Contact_1 - meets Top-level |
| Contact_2 - Contact Between Surfaces in 3D Space | All tests |
| Contact_3 - Default Spring Contact Force | All tests |
| Contact_4 - Limit Circular Contact | Insp. Contact_3 |
| Contact_5 - Contact Model Extensibility | Insp. Contact_2 |

Table 5.1: Requirements Traceability

## 5.4 Metrics

Table 5.2: Coarse Metrics

| | **Number of Lines** | | | |
|---|---|---|---|---|
| **File Name** | **Blank** | **Comment** | **Code** | **Total** |
| **Total** | 0 | 0 | 0 | 0 |

Table 5.3: Cyclomatic Complexity

| Method | File | Line | ECC |
|---|---|---|---|
| jeod::ContactFacet::get_name (void) | include/contact_facet.hh | 182 | 1 |
| jeod::ContactUtils::create_relstate_name (char * name1, char * name2, char ** out_str) | include/contact_utils_inline.hh | 85 | 1 |
| jeod::ContactUtils::copy_const_char_to_char (const char * in_str, char ** out_str) | include/contact_utils_inline.hh | 114 | 1 |
| jeod::ContactUtils::dist_line_segments (double p1[3], double p2[3], double p3[3], double p4[3], double *pa, double *pb) | include/contact_utils_inline.hh | 145 | 21 |
| jeod::Contact::Contact (void) | src/contact.cc | 49 | 1 |
| jeod::Contact::~Contact (void) | src/contact.cc | 66 | 4 |
| jeod::Contact::check_contact (void) | src/contact.cc | 89 | 6 |
| jeod::Contact::initialize_contact (DynManager * manager) | src/contact.cc | 111 | 8 |
| jeod::Contact::unique_pair (const ContactFacet * facet_1, const ContactFacet * facet_2) | src/contact.cc | 145 | 6 |
| jeod::Contact::register_contact (ContactFacet * facet) | src/contact.cc | 169 | 2 |
| jeod::Contact::register_contact (ContactFacet ** facets, unsigned int n Facets) | src/contact.cc | 189 | 2 |
| jeod::Contact::register_contact (ContactFacet * facet1, ContactFacet * facet2) | src/contact.cc | 208 | 3 |

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

| Method | File | Line | ECC |
|---|---|---|---|
| jeod::Contact::register_contact (ContactFacet ** facets1, unsigned int n Facets1, ContactFacet ** facets2, unsigned int n Facets2) | src/contact.cc | 232 | 3 |
| jeod::Contact::register_interaction (PairInteraction * interaction) | src/contact.cc | 258 | 1 |
| jeod::Contact::find_interaction (ContactParams * params_1, ContactParams * params_2) | src/contact.cc | 271 | 3 |
| jeod::ContactFacet::Contact Facet (void) | src/contact_facet.cc | 46 | 1 |
| jeod::ContactFacet::~Contact Facet (void) | src/contact_facet.cc | 62 | 1 |
| jeod::ContactFacet::create_vehicle_point () | src/contact_facet.cc | 73 | 2 |
| jeod::ContactPair::Contact Pair (void) | src/contact_pair.cc | 44 | 1 |
| jeod::ContactPair::~Contact Pair (void) | src/contact_pair.cc | 59 | 1 |
| jeod::ContactPair::in_range (void) | src/contact_pair.cc | 68 | 3 |
| jeod::ContactPair::is_active (void) | src/contact_pair.cc | 83 | 4 |
| jeod::ContactPair::is_complete (void) | src/contact_pair.cc | 97 | 2 |
| jeod::ContactPair::get_subject (void) | src/contact_pair.cc | 111 | 1 |
| jeod::ContactPair::get_target (void) | src/contact_pair.cc | 122 | 1 |
| jeod::ContactPair::check_tree (void) | src/contact_pair.cc | 133 | 2 |
| jeod::ContactPair::initialize_relstate (DynManager * dyn_manager) | src/contact_pair.cc | 154 | 1 |

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

| Method | File | Line | ECC |
|---|---|---|---|
| jeod::ContactParams::ContactParams (void) | src/contact_params.cc | 42 | 1 |
| jeod::ContactParams::~ContactParams (void) | src/contact_params.cc | 53 | 1 |
| jeod::ContactSurface::JEOD_DECLARE_ATTRIBUTES (ContactFacet) | src/contact_surface.cc | 50 | 1 |
| jeod::ContactSurface::~ContactSurface (void) | src/contact_surface.cc | 71 | 4 |
| jeod::ContactSurface::allocate_array (unsigned int size) | src/contact_surface.cc | 96 | 3 |
| jeod::ContactSurface::allocate_interaction_facet ( Facet* facet, InteractionFacetFactory* factory, FacetParams* params, unsigned int index) | src/contact_surface.cc | 134 | 4 |
| jeod::ContactSurface::collect_forces_torques (void) | src/contact_surface.cc | 211 | 2 |
| jeod::ContactSurfaceFactory::ContactSurfaceFactory (void) | src/contact_surface_factory.cc | 50 | 1 |
| jeod::ContactSurfaceFactory::~ContactSurfaceFactory (void) | src/contact_surface_factory.cc | 64 | 1 |
| jeod::ContactSurfaceFactory::create_surface (SurfaceModel* surface, InteractionSurface* inter_surface) | src/contact_surface_factory.cc | 82 | 13 |
| jeod::ContactSurfaceFactory::add_facet_params (FacetParams* to_add) | src/contact_surface_factory.cc | 192 | 3 |
| jeod::LineContactFacet::LineContactFacet (void) | src/line_contact_facet.cc | 49 | 1 |
| jeod::LineContactFacet::~LineContactFacet (void) | src/line_contact_facet.cc | 63 | 1 |
| jeod::LineContactFacet::create_pair (void) | src/line_contact_facet.cc | 75 | 1 |

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

| Method | File | Line | ECC |
|---|---|---|---|
| jeod::LineContactFacet:: create_pair (ContactFacet * target, Contact * contact) | src/line_contact_facet.cc | 94 | 6 |
| jeod::LineContactFacet:: calculate_contact_point (double nvec[3]) | src/line_contact_facet.cc | 167 | 4 |
| jeod::LineContactFacet::set_ max_dimension (void) | src/line_contact_facet.cc | 214 | 1 |
| jeod::LineContactFacet:: calculate_torque (double *tmp_force) | src/line_contact_facet.cc | 225 | 1 |
| jeod::LineContactFacet Factory::JEOD_DECLAR E_ATTRIBUTES (Line ContactFacet) | src/line_contact_facet_ factory.cc | 47 | 1 |
| jeod::LineContactFacet Factory::~LineContactFacet Factory (void) | src/line_contact_facet_ factory.cc | 64 | 1 |
| jeod::LineContactFacet Factory::create_facet ( Facet* facet, FacetParams* params) | src/line_contact_facet_ factory.cc | 81 | 4 |
| jeod::LineContactFacet Factory::is_correct_factory ( Facet* facet) | src/line_contact_facet_ factory.cc | 160 | 2 |
| jeod::LineContactPair::Line ContactPair (void) | src/line_contact_pair.cc | 44 | 1 |
| jeod::LineContactPair::~Line ContactPair (void) | src/line_contact_pair.cc | 57 | 1 |
| jeod::LineContactPair::in_ contact (void) | src/line_contact_pair.cc | 66 | 2 |
| jeod::LineContactPair:: initialize_pair (Contact Facet *subject_facet, ContactFacet *target_facet) | src/line_contact_pair.cc | 166 | 2 |
| jeod::LinePointContactPair:: LinePointContactPair (void) | src/line_point_contact_pair.cc | 45 | 1 |

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

| **Method** | **File** | **Line** | **ECC** |
|---|---|---|---|
| jeod::LinePointContactPair::~ LinePointContactPair (void) | src/line_point_contact_pair.cc | 58 | 1 |
| jeod::LinePointContactPair:: in_contact (void) | src/line_point_contact_pair.cc | 67 | 2 |
| jeod::LinePointContactPair:: initialize_pair (Contact Facet *subject_facet, ContactFacet *target_facet) | src/line_point_contact_pair.cc | 151 | 2 |
| jeod::PairInteraction::Pair Interaction (void) | src/pair_interaction.cc | 43 | 1 |
| jeod::PairInteraction::~Pair Interaction (void) | src/pair_interaction.cc | 57 | 1 |
| jeod::PairInteraction::is_ correct_interaction (Contact Params *subject_params, ContactParams *target_ params) | src/pair_interaction.cc | 66 | 5 |
| jeod::PointContactFacet:: PointContactFacet (void) | src/point_contact_facet.cc | 47 | 1 |
| jeod::PointContactFacet::~ PointContactFacet (void) | src/point_contact_facet.cc | 60 | 1 |
| jeod::PointContactFacet:: create_pair (void) | src/point_contact_facet.cc | 72 | 1 |
| jeod::PointContactFacet:: create_pair (ContactFacet * target, Contact * contact) | src/point_contact_facet.cc | 91 | 3 |
| jeod::PointContactFacet:: calculate_contact_point (double nvec[3]) | src/point_contact_facet.cc | 133 | 1 |
| jeod::PointContactFacet::set_ max_dimension (void) | src/point_contact_facet.cc | 150 | 1 |
| jeod::PointContactFacet:: calculate_torque (double *tmp_force) | src/point_contact_facet.cc | 161 | 1 |
| jeod::PointContactFacet Factory::JEOD_DECLAR E_ATTRIBUTES (Point ContactFacet) | src/point_contact_facet_ factory.cc | 47 | 1 |

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

| **Method** | **File** | **Line** | **ECC** |
|---|---|---|---|
| jeod::PointContactFacet Factory::~PointContact FacetFactory (void) | src/point_contact_facet_ factory.cc | 64 | 1 |
| jeod::PointContactFacet Factory::create_facet ( Facet* facet, FacetParams* params) | src/point_contact_facet_ factory.cc | 81 | 4 |
| jeod::PointContactFacet Factory::is_correct_factory ( Facet* facet) | src/point_contact_facet_ factory.cc | 158 | 2 |
| jeod::PointContactPair::Point ContactPair (void) | src/point_contact_pair.cc | 43 | 1 |
| jeod::PointContactPair::~ PointContactPair (void) | src/point_contact_pair.cc | 56 | 1 |
| jeod::PointContactPair::in_ contact (void) | src/point_contact_pair.cc | 65 | 2 |
| jeod::PointContactPair:: initialize_pair (Contact Facet *subject_facet, ContactFacet *target_facet) | src/point_contact_pair.cc | 117 | 2 |
| jeod::SpringPairInteraction:: SpringPairInteraction (void) | src/spring_pair_interaction.cc | 50 | 1 |
| jeod::SpringPairInteraction::~ SpringPairInteraction (void) | src/spring_pair_interaction.cc | 65 | 1 |
| jeod::SpringPairInteraction:: calculate_forces (Contact Facet * subject, Contact Facet * target, Relative DerivedState * rel_state, double* penetration_vector, double* rel_velocity) | src/spring_pair_interaction.cc | 74 | 2 |

# Bibliography

[1] Generated by doxygen. *JEOD Contact Model API*. NASA, Johnson Space Center, Software, Robotics & Simulation Division, Simulation and Graphics Branch, 2101 NASA Parkway, Houston, Texas, 77058, July 2023.

[2] Hammen, D. Dynamic Body Model. Technical Report JSC-61777-dynamics/dyn_body, NASA, Johnson Space Center, Houston, Texas, July 2023.

[3] Jackson, A., Thebeau, C. JSC Engineering Orbital Dynamics. Technical Report JSC-61777-docs, NASA, Johnson Space Center, Houston, Texas, July 2023.

[4] NASA. NASA Software Engineering Requirements. Technical Report NPR-7150.2, NASA, NASA Headquarters, Washington, D.C., September 2004.

[5] Spencer, A. Surface Model. Technical Report JSC-61777-utils/surface_model, NASA, Johnson Space Center, Houston, Texas, July 2023.

[6] Thebeau, C. Mass Body Model. Technical Report JSC-61777-dynamics/mass, NASA, Johnson Space Center, Houston, Texas, July 2023.

[7] Thompson, B. Mathematical Functions Model. Technical Report JSC-61777-utils/math, NASA, Johnson Space Center, Houston, Texas, July 2023.