NORTHEASTERN UNIVERSITY

ME5250 Robot Mechanics and Control

Date: 04/26/2024

Project: Inverse dynamics of RR Robot with circulatory Trajectory

ABSTRACT

This research investigates the inverse dynamics problem of a rotating-revolute (RR) robotic manipulator, aiming to determine the joint torques necessary for executing a circular trajectory at a constant speed within the manipulator's workspace. Employing an iterative numerical method guided by principles outlined in Section 8.3 of the textbook, the study focuses on the RR manipulator, featuring two revolute joints, to maintain tractability while exploring the intricacies of the inverse dynamics problem. Through meticulous implementation of input modeling, inverse kinematics computation, mechanical modeling, and forward kinematics analysis, the project achieves a comprehensive understanding of the RR robot's dynamics. Verification of the computed joint torques is conducted through either analytical calculation or forward dynamics simulation, affirming the accuracy of the computed joint torque trajectories and assessing the efficacy of the iterative numerical method. The results, supported by plots illustrating the recorded joint torques during the simulation, provide valuable insights into solving the inverse dynamics problem for the RR manipulator.

INTRODUCTION

Robotic manipulators are essential to many industrial applications because they provide unmatched accuracy and adaptability in completing manufacturing and medical jobs. In the field of robotic control, it is critical to comprehend these systems' dynamics in order to create effective control algorithms that produce the appropriate end-effector trajectories. One of the core ideas in robotics is the inverse dynamics issue, which is figuring out the joint torques needed to accomplish certain end-effector motions. In this research, we investigate the inverse dynamics issue for a rotating-revolute (RR) robotic manipulator to calculate the joint torques required to move along a circular track at a constant speed inside the manipulator's workspace.

The RR manipulator, which has two revolute joints, is a basic yet simpler form that is frequently used in robotic applications. We can investigate the nuances of the inverse dynamics issue while preserving tractability if we concentrate on this setup. Our goal is to create an iterative numerical method to solve the inverse dynamics issue for the RR manipulator, following the guidelines in Section 8.3 of the textbook. We will be able to calculate the joint torques needed to reach the specified joint locations, velocities, and accelerations needed to follow a circular trajectory thanks to this technique.

We will use an analytical calculation or forward dynamics simulation method of verification to confirm the correctness of our estimated joint torque trajectories. We may simulate the forward dynamics by applying starting circumstances to the RR manipulator, such as joint locations and velocities, and then comparing the ensuing end-effector movements with

the required circular trajectory. This stage of verification confirms that our computed joint torques are accurate and highlights how well our iterative numerical technique works to solve the inverse dynamics issue for the RR manipulator.

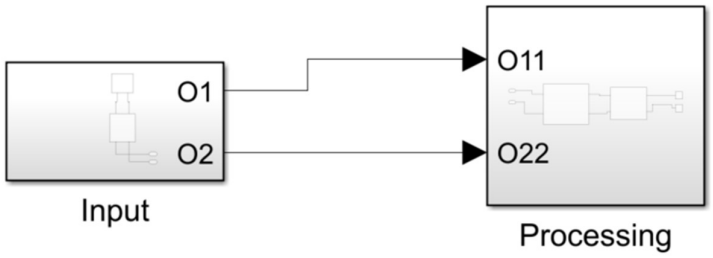


Fig.1 Overall block diagram of RR robot circular trajectory simulation

METHODOLOGY

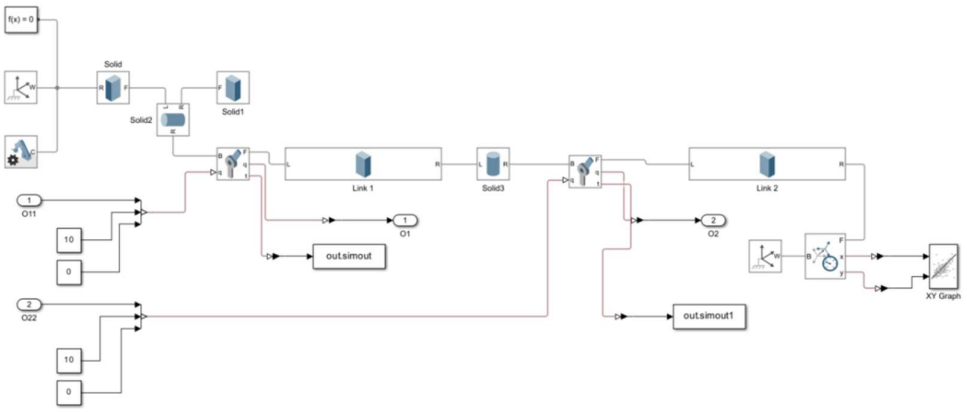
1. **Input Model**:
   - The path taken by the robotic arm is incorporated into the input model. It defines the x and y path for the robotic simulation using the sin and cos functions after receiving information from the clock. The following are the expressions that are utilized in this project for input:
   - 1 + 0.25*sin((2*pi/5)*u + pi/2 and 1 + 0.25*cos((2*pi/5)*u + pi/2)
2. **Inverse Kinematics**:
   - It comprises the matrices and the MATLAB function that feeds the inverse function, which computes the angles (O1, O2) and other necessary metrics, the X1 and Y1 from the trajectory model, and the mechanical model of our robotic arm.

   - The following are the formulae that are utilized in this project for finding the angle:

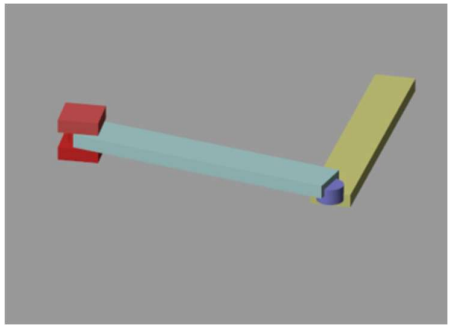
$$O1 = atan2(Y1, X1) - atan2l2 * sinO2, l1 + l2 * cosO2$$

$$O2 = acos\left((X1^2 + Y1^2 - l1^2 - l2^2)/(2 * l1 * l2)\right)$$

3. **Mechanical Model**:
   - It includes a mechanical model of our robotic arm that includes every joint, support, arm, and leg of the robot. To handle a wide range of motions, Simulink offers incredibly useful features including joints, sensors, linkages, functions, and continuous assistance.
   - As shown in fig.2 below, Number of blocks, sensors and joints are used to develop this mechanical model replicate the RR robot.



a.) Simulink Blocks



b.) RR Robot simulation

Fig.2 Mechanical Model of the RR robot

4. **Forward Kinematics**:
    - It has all the matrices and functions that, like inverse kinematics model, use angles from the mechanical model as the input and compute X and Y to move the arm as the output.

RESULT AND DISCUSSION

Finally, Using the angle outputs from the forward kinematics model, the circular trajectory of RR robot is plotted with the help of matplot codes as shown in fig.3.

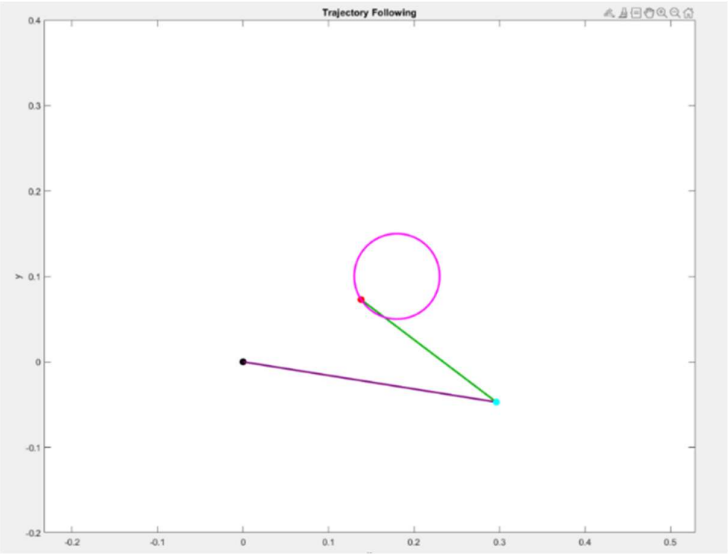


Fig.3 Circular trajectory of the RR robot and RR robot visualization

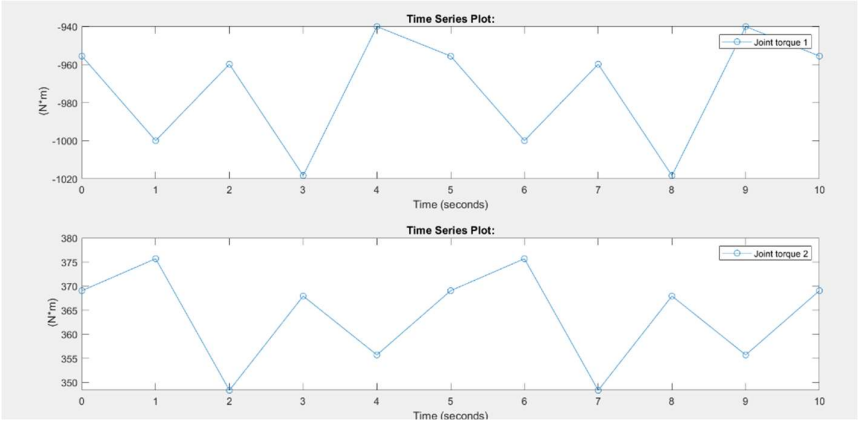


Fig.4 Plots of Joint Torques

The joint torques are recorded in graph using matplot as shown in fig.4 and this is acquired from the desired trajectory of the end-effector of the robot. Therefore, we verified that joint torque trajectories are correct when comparing to the simulated output of the RR Robot. For reference, the following is the link for this project's output: https://youtu.be/8FWqnjeGYjw

CONCLUSION

In conclusion, this study successfully addresses the inverse dynamics problem for a rotating-revolute (RR) robotic manipulator, elucidating the joint torques necessary to traverse a circular trajectory with consistent velocity within the robot's workspace. By employing an iterative numerical method guided by established principles, the project not only provides a robust framework for solving the inverse dynamics problem but also enhances comprehension of the manipulator's dynamics. The findings underscore the importance of understanding and solving inverse dynamics problems in robotic control, paving the way for enhanced efficiency and precision in various industrial and medical applications.

Code used to find the joint torque:

```
open_system('RR');
set_param('RR/To Workspace',...
    'VariableName','simoutToWorkspace')

set_param('RR/To File',...
    'FileName','simoutToFile.mat',...
    'MatrixName','simoutToFileVariable')

out= sim('RR');
load('simoutToFile.mat')
subplot(2,1,1)
plot(out.simoutToWorkspace,'-o')
legend('Joint torque 1')

load('simoutToFile3.mat')
subplot(2,1,2)
plot(out.simoutToWorkspace1,'-o')
legend('Joint torque 2')
```

```
/*
 * RR.c
 *
 * Academic License - for use in teaching, academic research, and meeting
 * course requirements at degree granting institutions only.  Not for
 * government, commercial, or other organizational use.
 *
 * Code generation for model "RR".
 *
 * Model version              : 11.13
 * Simulink Coder version : 24.1 (R2024a) 19-Nov-2023
 * C source code generated on : Fri Apr 26 15:24:11 2024
 *
 * Target selection: grt.tlc
 * Note: GRT includes extra infrastructure and instrumentation for prototyping
 * Embedded hardware selection: Intel->x86-64 (Windows64)
 * Code generation objective: Debugging
 * Validation result: Not run
 */

#include "RR.h"
#include "rtwtypes.h"
#include <math.h>
#include "RR_private.h"
#include <stddef.h>
#include <string.h>
#include "rt_nonfinite.h"
#include "rt_defines.h"

/* Block signals (default storage) */
B_RR_T RR_B;

/* Block states (default storage) */
DW_RR_T RR_DW;

/* Real-time model */
static RT_MODEL_RR_T RR_M_;
RT_MODEL_RR_T *const RR_M = &RR_M_;
real_T rt_atan2d_snf(real_T u0, real_T u1)
{
  real_T y;
  if (rtIsNaN(u0) || rtIsNaN(u1)) {
    y = (rtNaN);
  } else if (rtIsInf(u0) && rtIsInf(u1)) {
    int32_T tmp;
    int32_T tmp_0;
    if (u1 > 0.0) {
      tmp = 1;
    } else {
      tmp = -1;
```

```
    }

    if (u0 > 0.0) {
      tmp_0 = 1;
    } else {
      tmp_0 = -1;
    }

    y = atan2(tmp_0, tmp);
  } else if (u1 == 0.0) {
    if (u0 > 0.0) {
      y = RT_PI / 2.0;
    } else if (u0 < 0.0) {
      y = -(RT_PI / 2.0);
    } else {
      y = 0.0;
    }
  } else {
    y = atan2(u0, u1);
  }

  return y;
}

/* Model step function */
void RR_step(void)
{
  {
    NeslSimulationData *simulationData;
    NeslSimulator *simulator;
    NeuDiagnosticManager *diagnosticManager;
    NeuDiagnosticTree *diagnosticTree;
    char *msg;
    real_T T20[16];
    real_T tmp_e[16];
    real_T tmp_f[16];
    real_T tmp_0[8];
    real_T tmp_2[8];
    real_T O2;
    real_T T20_0;
    real_T time;
    real_T time_0;
    real_T time_1;
    real_T time_2;
    real_T tmp_4;
    real_T tmp_5;
    real_T tmp_6;
    real_T tmp_7;
    real_T tmp_8;
    real_T tmp_9;
```

```
  real_T tmp_a;
  real_T tmp_b;
  real_T tmp_c;
  real_T tmp_d;
  int32_T i;
  int32_T i_0;
  int32_T tmp_g;
  int_T tmp_1[3];
  int_T tmp_3[3];
  boolean_T tmp;

  /* Clock: '<S4>/Clock' */
  RR_B.Clock = RR_M->Timing.t[0];

  /* Fcn: '<S4>/Fcn' */
  RR_B.Fcn = sin(1.2566370614359172 * RR_B.Clock + 1.5707963267948966) * 0.25
    + 1.0;

  /* Fcn: '<S4>/Fcn1' */
  RR_B.Fcn1 = cos(1.2566370614359172 * RR_B.Clock + 1.5707963267948966) * 0.25
    + 1.0;

  /* MATLAB Function: '<S1>/Algorithm for robotic arm input' */
  /* :  l1 = 1; */
  /* :  l2 = 1; */
  /* :  O2 = acos((X1^2+Y1^2 - l1^2 - l2^2)/(2*l1*l2)); */
  O2 = acos((((RR_B.Fcn * RR_B.Fcn + RR_B.Fcn1 * RR_B.Fcn1) - 1.0) - 1.0) /
            2.0);

  /* :  s_O2 = sin(O2); */
  /* :  c_O2 = cos(O2); */
  /* :  O1 = atan2(Y1,X1) - atan2(l2*s_O2, (l1+l2*c_O2)); */
  RR_B.O1 = rt_atan2d_snf(RR_B.Fcn1, RR_B.Fcn) - rt_atan2d_snf(sin(O2), cos(O2)
    + 1.0);
  RR_B.O2 = O2;

  /* SimscapeInputBlock: '<S24>/INPUT_2_1_1' incorporates:
   *  Constant: '<S6>/Constant'
   *  Constant: '<S6>/Constant1'
   */
  RR_B.INPUT_2_1_1[0] = RR_B.O1;
  RR_B.INPUT_2_1_1[1] = RR_P.Constant_Value;
  RR_B.INPUT_2_1_1[2] = RR_P.Constant1_Value;
  if (rtmIsMajorTimeStep(RR_M)) {
    RR_DW.INPUT_2_1_1_Discrete[0] = !(RR_DW.INPUT_2_1_1_Discrete[1] ==
      RR_B.INPUT_2_1_1[2]);
    RR_DW.INPUT_2_1_1_Discrete[1] = RR_B.INPUT_2_1_1[2];
  }

  RR_B.INPUT_2_1_1[2] = RR_DW.INPUT_2_1_1_Discrete[1];
```

```
  RR_B.INPUT_2_1_1[3] = RR_DW.INPUT_2_1_1_Discrete[0];

  /* End of SimscapeInputBlock: '<S24>/INPUT_2_1_1' */

  /* SimscapeInputBlock: '<S24>/INPUT_1_1_1' incorporates:
   *  Constant: '<S6>/Constant2'
   *  Constant: '<S6>/Constant3'
   */
  RR_B.INPUT_1_1_1[0] = RR_B.O2;
  RR_B.INPUT_1_1_1[1] = RR_P.Constant2_Value;
  RR_B.INPUT_1_1_1[2] = RR_P.Constant3_Value;
  if (rtmIsMajorTimeStep(RR_M)) {
    RR_DW.INPUT_1_1_1_Discrete[0] = !(RR_DW.INPUT_1_1_1_Discrete[1] ==
      RR_B.INPUT_1_1_1[2]);
    RR_DW.INPUT_1_1_1_Discrete[1] = RR_B.INPUT_1_1_1[2];
  }

  RR_B.INPUT_1_1_1[2] = RR_DW.INPUT_1_1_1_Discrete[1];
  RR_B.INPUT_1_1_1[3] = RR_DW.INPUT_1_1_1_Discrete[0];

  /* End of SimscapeInputBlock: '<S24>/INPUT_1_1_1' */

  /* SimscapeExecutionBlock: '<S24>/OUTPUT_1_0' */
  simulationData = (NeslSimulationData *)RR_DW.OUTPUT_1_0_SimData;
  time = RR_M->Timing.t[0];
  simulationData->mData->mTime.mN = 1;
  simulationData->mData->mTime.mX = &time;
  simulationData->mData->mContStates.mN = 0;
  simulationData->mData->mContStates.mX = NULL;
  simulationData->mData->mDiscStates.mN = 0;
  simulationData->mData->mDiscStates.mX = &RR_DW.OUTPUT_1_0_Discrete;
  simulationData->mData->mModeVector.mN = 0;
  simulationData->mData->mModeVector.mX = &RR_DW.OUTPUT_1_0_Modes;
  tmp = false;
  simulationData->mData->mFoundZcEvents = tmp;
  simulationData->mData->mHadEvents = false;
  simulationData->mData->mIsMajorTimeStep = rtmIsMajorTimeStep(RR_M);
  tmp = false;
  simulationData->mData->mIsSolverAssertCheck = tmp;
  simulationData->mData->mIsSolverCheckingCIC = false;
  simulationData->mData->mIsComputingJacobian = false;
  simulationData->mData->mIsEvaluatingF0 = false;
  simulationData->mData->mIsSolverRequestingReset = false;
  simulationData->mData->mIsModeUpdateTimeStep = rtsiIsModeUpdateTimeStep
    (&RR_M->solverInfo);
  tmp_1[0] = 0;
  tmp_0[0] = RR_B.INPUT_2_1_1[0];
  tmp_0[1] = RR_B.INPUT_2_1_1[1];
  tmp_0[2] = RR_B.INPUT_2_1_1[2];
  tmp_0[3] = RR_B.INPUT_2_1_1[3];
```

```
  tmp_1[1] = 4;
  tmp_0[4] = RR_B.INPUT_1_1_1[0];
  tmp_0[5] = RR_B.INPUT_1_1_1[1];
  tmp_0[6] = RR_B.INPUT_1_1_1[2];
  tmp_0[7] = RR_B.INPUT_1_1_1[3];
  tmp_1[2] = 8;
  simulationData->mData->mInputValues.mN = 8;
  simulationData->mData->mInputValues.mX = &tmp_0[0];
  simulationData->mData->mInputOffsets.mN = 3;
  simulationData->mData->mInputOffsets.mX = &tmp_1[0];
  simulationData->mData->mOutputs.mN = 6;
  simulationData->mData->mOutputs.mX = &RR_B.OUTPUT_1_0[0];
  simulationData->mData->mTolerances.mN = 0;
  simulationData->mData->mTolerances.mX = NULL;
  simulationData->mData->mCstateHasChanged = false;
  simulationData->mData->mDstateHasChanged = false;
  time_0 = RR_M->Timing.t[0];
  simulationData->mData->mTime.mN = 1;
  simulationData->mData->mTime.mX = &time_0;
  simulationData->mData->mSampleHits.mN = 0;
  simulationData->mData->mSampleHits.mX = NULL;
  simulationData->mData->mIsFundamentalSampleHit = false;
  simulationData->mData->mHadEvents = false;
  simulator = (NeslSimulator *)RR_DW.OUTPUT_1_0_Simulator;
  diagnosticManager = (NeuDiagnosticManager *)RR_DW.OUTPUT_1_0_DiagMgr;
  diagnosticTree = neu_diagnostic_manager_get_initial_tree(diagnosticManager);
  i = ne_simulator_method(simulator, NESL_SIM_OUTPUTS, simulationData,
    diagnosticManager);
  if (i != 0) {
    tmp = error_buffer_is_empty(rtmGetErrorStatus(RR_M));
    if (tmp) {
      msg = rtw_diagnostics_msg(diagnosticTree);
      rtmSetErrorStatus(RR_M, msg);
    }
  }

  /* End of SimscapeExecutionBlock: '<S24>/OUTPUT_1_0' */

  /* MATLAB Function: '<S2>/Algorithm for Circulatory trajectory' */
  /* :  l1 = 1; */
  /* :  l2 = 1; */
  /* :  T10 = [cos(O1), -sin(O1), 0, l1*cos(O1); */
  /* :      sin(O1), cos(O1), 0, l1*sin(O1); */
  /* :      0, 0, 1, 0; */
  /* :      0, 0, 0, 1]; */
  /* :  T21 = [cos(O2), -sin(O2), 0, l1*cos(O2); */
  /* :      sin(O2), cos(O2), 0, l1*sin(O2); */
  /* :      0, 0, 1, 0; */
  /* :      0, 0, 0, 1]; */
  /* :  T20 = T10*T21; */
```

```
  O2 = cos(RR_B.OUTPUT_1_0[0]);
  tmp_4 = sin(RR_B.OUTPUT_1_0[0]);
  tmp_5 = cos(RR_B.OUTPUT_1_0[0]);
  tmp_6 = sin(RR_B.OUTPUT_1_0[0]);
  T20_0 = cos(RR_B.OUTPUT_1_0[0]);
  tmp_7 = sin(RR_B.OUTPUT_1_0[0]);
  tmp_8 = cos(RR_B.OUTPUT_1_0[2]);
  tmp_9 = sin(RR_B.OUTPUT_1_0[2]);
  tmp_a = cos(RR_B.OUTPUT_1_0[2]);
  tmp_b = sin(RR_B.OUTPUT_1_0[2]);
  tmp_c = cos(RR_B.OUTPUT_1_0[2]);
  tmp_d = sin(RR_B.OUTPUT_1_0[2]);
  tmp_e[0] = O2;
  tmp_e[4] = -tmp_4;
  tmp_e[8] = 0.0;
  tmp_e[12] = tmp_5;
  tmp_e[1] = tmp_6;
  tmp_e[5] = T20_0;
  tmp_e[9] = 0.0;
  tmp_e[13] = tmp_7;
  tmp_f[0] = tmp_8;
  tmp_f[4] = -tmp_9;
  tmp_f[8] = 0.0;
  tmp_f[12] = tmp_a;
  tmp_f[1] = tmp_b;
  tmp_f[5] = tmp_c;
  tmp_f[9] = 0.0;
  tmp_f[13] = tmp_d;
  tmp_e[2] = 0.0;
  tmp_e[3] = 0.0;
  tmp_f[2] = 0.0;
  tmp_f[3] = 0.0;
  tmp_e[6] = 0.0;
  tmp_e[7] = 0.0;
  tmp_f[6] = 0.0;
  tmp_f[7] = 0.0;
  tmp_e[10] = 1.0;
  tmp_e[11] = 0.0;
  tmp_f[10] = 1.0;
  tmp_f[11] = 0.0;
  tmp_e[14] = 0.0;
  tmp_e[15] = 1.0;
  tmp_f[14] = 0.0;
  tmp_f[15] = 1.0;
  for (i = 0; i < 4; i++) {
    tmp_g = i << 2;
    O2 = tmp_f[tmp_g];
    tmp_4 = tmp_f[tmp_g + 1];
    tmp_5 = tmp_f[tmp_g + 2];
    tmp_6 = tmp_f[tmp_g + 3];
```

```
    for (i_0 = 0; i_0 < 4; i_0++) {
      T20_0 = O2 * tmp_e[i_0];
      T20_0 += tmp_e[i_0 + 4] * tmp_4;
      T20_0 += tmp_e[i_0 + 8] * tmp_5;
      T20_0 += tmp_e[i_0 + 12] * tmp_6;
      T20[i_0 + tmp_g] = T20_0;
    }
  }

  /* :  X = T20(1,4); */
  RR_B.X = T20[12];

  /* :  Y = T20(2,4); */
  RR_B.Y = T20[13];

  /* End of MATLAB Function: '<S2>/Algorithm for Circulatory trajectory' */
  /* SimscapeExecutionBlock: '<S24>/STATE_1' */
  simulationData = (NeslSimulationData *)RR_DW.STATE_1_SimData;
  time_1 = RR_M->Timing.t[0];
  simulationData->mData->mTime.mN = 1;
  simulationData->mData->mTime.mX = &time_1;
  simulationData->mData->mContStates.mN = 0;
  simulationData->mData->mContStates.mX = NULL;
  simulationData->mData->mDiscStates.mN = 0;
  simulationData->mData->mDiscStates.mX = &RR_DW.STATE_1_Discrete;
  simulationData->mData->mModeVector.mN = 0;
  simulationData->mData->mModeVector.mX = &RR_DW.STATE_1_Modes;
  tmp = false;
  simulationData->mData->mFoundZcEvents = tmp;
  simulationData->mData->mHadEvents = false;
  simulationData->mData->mIsMajorTimeStep = rtmIsMajorTimeStep(RR_M);
  tmp = false;
  simulationData->mData->mIsSolverAssertCheck = tmp;
  simulationData->mData->mIsSolverCheckingCIC = false;
  simulationData->mData->mIsComputingJacobian = false;
  simulationData->mData->mIsEvaluatingF0 = false;
  simulationData->mData->mIsSolverRequestingReset = false;
  simulationData->mData->mIsModeUpdateTimeStep = rtsiIsModeUpdateTimeStep
    (&RR_M->solverInfo);
  tmp_3[0] = 0;
  tmp_2[0] = RR_B.INPUT_2_1_1[0];
  tmp_2[1] = RR_B.INPUT_2_1_1[1];
  tmp_2[2] = RR_B.INPUT_2_1_1[2];
  tmp_2[3] = RR_B.INPUT_2_1_1[3];
  tmp_3[1] = 4;
  tmp_2[4] = RR_B.INPUT_1_1_1[0];
  tmp_2[5] = RR_B.INPUT_1_1_1[1];
  tmp_2[6] = RR_B.INPUT_1_1_1[2];
  tmp_2[7] = RR_B.INPUT_1_1_1[3];
  tmp_3[2] = 8;
```

```
    simulationData->mData->mInputValues.mN = 8;
    simulationData->mData->mInputValues.mX = &tmp_2[0];
    simulationData->mData->mInputOffsets.mN = 3;
    simulationData->mData->mInputOffsets.mX = &tmp_3[0];
    simulationData->mData->mOutputs.mN = 0;
    simulationData->mData->mOutputs.mX = NULL;
    simulationData->mData->mTolerances.mN = 0;
    simulationData->mData->mTolerances.mX = NULL;
    simulationData->mData->mCstateHasChanged = false;
    simulationData->mData->mDstateHasChanged = false;
    time_2 = RR_M->Timing.t[0];
    simulationData->mData->mTime.mN = 1;
    simulationData->mData->mTime.mX = &time_2;
    simulationData->mData->mSampleHits.mN = 0;
    simulationData->mData->mSampleHits.mX = NULL;
    simulationData->mData->mIsFundamentalSampleHit = false;
    simulationData->mData->mHadEvents = false;
    simulator = (NeslSimulator *)RR_DW.STATE_1_Simulator;
    diagnosticManager = (NeuDiagnosticManager *)RR_DW.STATE_1_DiagMgr;
    diagnosticTree = neu_diagnostic_manager_get_initial_tree(diagnosticManager);
    i = ne_simulator_method(simulator, NESL_SIM_OUTPUTS, simulationData,
      diagnosticManager);
    if (i != 0) {
      tmp = error_buffer_is_empty(rtmGetErrorStatus(RR_M));
      if (tmp) {
        msg = rtw_diagnostics_msg(diagnosticTree);
        rtmSetErrorStatus(RR_M, msg);
      }
    }

    /* End of SimscapeExecutionBlock: '<S24>/STATE_1' */
  }

  /* Matfile logging */
  rt_UpdateTXYLogVars(RR_M->rtwLogInfo, (RR_M->Timing.t));

  {
    NeslSimulationData *simulationData;
    NeslSimulator *simulator;
    NeuDiagnosticManager *diagnosticManager;
    NeuDiagnosticTree *diagnosticTree;
    char *msg;
    real_T tmp_0[8];
    real_T time;
    int32_T tmp_2;
    int_T tmp_1[3];
    boolean_T tmp;

    /* Update for SimscapeExecutionBlock: '<S24>/STATE_1' */
    simulationData = (NeslSimulationData *)RR_DW.STATE_1_SimData;
```

```
  time = RR_M->Timing.t[0];
  simulationData->mData->mTime.mN = 1;
  simulationData->mData->mTime.mX = &time;
  simulationData->mData->mContStates.mN = 0;
  simulationData->mData->mContStates.mX = NULL;
  simulationData->mData->mDiscStates.mN = 0;
  simulationData->mData->mDiscStates.mX = &RR_DW.STATE_1_Discrete;
  simulationData->mData->mModeVector.mN = 0;
  simulationData->mData->mModeVector.mX = &RR_DW.STATE_1_Modes;
  tmp = false;
  simulationData->mData->mFoundZcEvents = tmp;
  simulationData->mData->mHadEvents = false;
  simulationData->mData->mIsMajorTimeStep = rtmIsMajorTimeStep(RR_M);
  tmp = false;
  simulationData->mData->mIsSolverAssertCheck = tmp;
  simulationData->mData->mIsSolverCheckingCIC = false;
  simulationData->mData->mIsComputingJacobian = false;
  simulationData->mData->mIsEvaluatingF0 = false;
  simulationData->mData->mIsSolverRequestingReset = false;
  simulationData->mData->mIsModeUpdateTimeStep = rtsiIsModeUpdateTimeStep
    (&RR_M->solverInfo);
  tmp_1[0] = 0;
  tmp_0[0] = RR_B.INPUT_2_1_1[0];
  tmp_0[1] = RR_B.INPUT_2_1_1[1];
  tmp_0[2] = RR_B.INPUT_2_1_1[2];
  tmp_0[3] = RR_B.INPUT_2_1_1[3];
  tmp_1[1] = 4;
  tmp_0[4] = RR_B.INPUT_1_1_1[0];
  tmp_0[5] = RR_B.INPUT_1_1_1[1];
  tmp_0[6] = RR_B.INPUT_1_1_1[2];
  tmp_0[7] = RR_B.INPUT_1_1_1[3];
  tmp_1[2] = 8;
  simulationData->mData->mInputValues.mN = 8;
  simulationData->mData->mInputValues.mX = &tmp_0[0];
  simulationData->mData->mInputOffsets.mN = 3;
  simulationData->mData->mInputOffsets.mX = &tmp_1[0];
  simulator = (NeslSimulator *)RR_DW.STATE_1_Simulator;
  diagnosticManager = (NeuDiagnosticManager *)RR_DW.STATE_1_DiagMgr;
  diagnosticTree = neu_diagnostic_manager_get_initial_tree(diagnosticManager);
  tmp_2 = ne_simulator_method(simulator, NESL_SIM_UPDATE, simulationData,
    diagnosticManager);
  if (tmp_2 != 0) {
    tmp = error_buffer_is_empty(rtmGetErrorStatus(RR_M));
    if (tmp) {
      msg = rtw_diagnostics_msg(diagnosticTree);
      rtmSetErrorStatus(RR_M, msg);
    }
  }

  /* End of Update for SimscapeExecutionBlock: '<S24>/STATE_1' */
```

```
  }

  /* signal main to stop simulation */
  {                                    /* Sample time: [0.0s, 0.0s] */
    if ((rtmGetTFinal(RR_M)!=-1) &&
        !((rtmGetTFinal(RR_M)-RR_M->Timing.t[0]) > RR_M->Timing.t[0] *
          (DBL_EPSILON))) {
      rtmSetErrorStatus(RR_M, "Simulation finished");
    }
  }

  /* Update absolute time for base rate */
  /* The "clockTick0" counts the number of times the code of this task has
   * been executed. The absolute time is the multiplication of "clockTick0"
   * and "Timing.stepSize0". Size of "clockTick0" ensures timer will not
   * overflow during the application lifespan selected.
   * Timer of this task consists of two 32 bit unsigned integers.
   * The two integers represent the low bits Timing.clockTick0 and the high bits
   * Timing.clockTickH0. When the low bit overflows to 0, the high bits increment.
   */
  if (!(++RR_M->Timing.clockTick0)) {
    ++RR_M->Timing.clockTickH0;
  }

  RR_M->Timing.t[0] = RR_M->Timing.clockTick0 * RR_M->Timing.stepSize0 +
    RR_M->Timing.clockTickH0 * RR_M->Timing.stepSize0 * 4294967296.0;

  {
    /* Update absolute timer for sample time: [0.2s, 0.0s] */
    /* The "clockTick1" counts the number of times the code of this task has
     * been executed. The resolution of this integer timer is 0.2, which is the step
size
     * of the task. Size of "clockTick1" ensures timer will not overflow during the
     * application lifespan selected.
     * Timer of this task consists of two 32 bit unsigned integers.
     * The two integers represent the low bits Timing.clockTick1 and the high bits
     * Timing.clockTickH1. When the low bit overflows to 0, the high bits increment.
     */
    RR_M->Timing.clockTick1++;
    if (!RR_M->Timing.clockTick1) {
      RR_M->Timing.clockTickH1++;
    }
  }
}

/* Model initialize function */
void RR_initialize(void)
{
  /* Registration code */
```

```
  /* initialize non-finites */
  rt_InitInfAndNaN(sizeof(real_T));

  /* initialize real-time model */
  (void) memset((void *)RR_M, 0,
                sizeof(RT_MODEL_RR_T));

  {
    /* Setup solver object */
    rtsiSetSimTimeStepPtr(&RR_M->solverInfo, &RR_M->Timing.simTimeStep);
    rtsiSetTPtr(&RR_M->solverInfo, &rtmGetTPtr(RR_M));
    rtsiSetStepSizePtr(&RR_M->solverInfo, &RR_M->Timing.stepSize0);
    rtsiSetErrorStatusPtr(&RR_M->solverInfo, (&rtmGetErrorStatus(RR_M)));
    rtsiSetRTModelPtr(&RR_M->solverInfo, RR_M);
  }

  rtsiSetSimTimeStep(&RR_M->solverInfo, MAJOR_TIME_STEP);
  rtsiSetIsMinorTimeStepWithModeChange(&RR_M->solverInfo, false);
  rtsiSetIsContModeFrozen(&RR_M->solverInfo, false);
  rtsiSetSolverName(&RR_M->solverInfo,"FixedStepDiscrete");
  rtmSetTPtr(RR_M, &RR_M->Timing.tArray[0]);
  rtmSetTFinal(RR_M, 10.0);
  RR_M->Timing.stepSize0 = 0.2;

  /* Setup for data logging */
  {
    static RTWLogInfo rt_DataLoggingInfo;
    rt_DataLoggingInfo.loggingInterval = (NULL);
    RR_M->rtwLogInfo = &rt_DataLoggingInfo;
  }

  /* Setup for data logging */
  {
    rtliSetLogXSignalInfo(RR_M->rtwLogInfo, (NULL));
    rtliSetLogXSignalPtrs(RR_M->rtwLogInfo, (NULL));
    rtliSetLogT(RR_M->rtwLogInfo, "tout");
    rtliSetLogX(RR_M->rtwLogInfo, "");
    rtliSetLogXFinal(RR_M->rtwLogInfo, "");
    rtliSetLogVarNameModifier(RR_M->rtwLogInfo, "rt_");
    rtliSetLogFormat(RR_M->rtwLogInfo, 4);
    rtliSetLogMaxRows(RR_M->rtwLogInfo, 1000);
    rtliSetLogDecimation(RR_M->rtwLogInfo, 1);
    rtliSetLogY(RR_M->rtwLogInfo, "");
    rtliSetLogYSignalInfo(RR_M->rtwLogInfo, (NULL));
    rtliSetLogYSignalPtrs(RR_M->rtwLogInfo, (NULL));
  }

  /* block I/O */
  (void) memset(((void *) &RR_B), 0,
                sizeof(B_RR_T));
```

```
  /* states (dwork) */
  (void) memset((void *)&RR_DW, 0,
                sizeof(DW_RR_T));

  /* Matfile logging */
  rt_StartDataLoggingWithStartTime(RR_M->rtwLogInfo, 0.0, rtmGetTFinal(RR_M),
    RR_M->Timing.stepSize0, (&rtmGetErrorStatus(RR_M)));

  {
    NeModelParameters modelParameters;
    NeModelParameters modelParameters_0;
    NeslSimulationData *tmp;
    NeslSimulator *simulator;
    NeuDiagnosticManager *diagnosticManager;
    NeuDiagnosticTree *diagnosticTree;
    char *msg;
    real_T tmp_0;
    int32_T tmp_1;
    boolean_T zcDisabled;

    /* Start for SimscapeExecutionBlock: '<S24>/OUTPUT_1_0' */
    simulator = nesl_lease_simulator(
      "RR/Processing/Mechnical Model /Solver Configuration_1", 1, 0);
    RR_DW.OUTPUT_1_0_Simulator = (void *)simulator;
    zcDisabled = pointer_is_null(RR_DW.OUTPUT_1_0_Simulator);
    if (zcDisabled) {
      RR_836bb176_1_gateway();
      simulator = nesl_lease_simulator(
        "RR/Processing/Mechnical Model /Solver Configuration_1", 1, 0);
      RR_DW.OUTPUT_1_0_Simulator = (void *)simulator;
    }

    tmp = nesl_create_simulation_data();
    RR_DW.OUTPUT_1_0_SimData = (void *)tmp;
    diagnosticManager = rtw_create_diagnostics();
    RR_DW.OUTPUT_1_0_DiagMgr = (void *)diagnosticManager;
    modelParameters.mSolverType = NE_SOLVER_TYPE_DAE;
    modelParameters.mSolverAbsTol = 0.001;
    modelParameters.mSolverRelTol = 0.001;
    modelParameters.mSolverModifyAbsTol = NE_MODIFY_ABS_TOL_NO;
    modelParameters.mStartTime = 0.0;
    modelParameters.mLoadInitialState = false;
    modelParameters.mUseSimState = false;
    modelParameters.mLinTrimCompile = false;
    modelParameters.mLoggingMode = SSC_LOGGING_ON;
    modelParameters.mRTWModifiedTimeStamp = 6.36060226E+8;
    modelParameters.mUseModelRefSolver = false;
    modelParameters.mTargetFPGAHIL = false;
    tmp_0 = 0.001;
```

```
  modelParameters.mSolverTolerance = tmp_0;
  tmp_0 = 0.2;
  modelParameters.mFixedStepSize = tmp_0;
  zcDisabled = false;
  modelParameters.mVariableStepSolver = zcDisabled;
  zcDisabled = false;
  modelParameters.mIsUsingODEN = zcDisabled;
  modelParameters.mZcDisabled = true;
  simulator = (NeslSimulator *)RR_DW.OUTPUT_1_0_Simulator;
  diagnosticManager = (NeuDiagnosticManager *)RR_DW.OUTPUT_1_0_DiagMgr;
  diagnosticTree = neu_diagnostic_manager_get_initial_tree(diagnosticManager);
  tmp_1 = nesl_initialize_simulator(simulator, &modelParameters,
    diagnosticManager);
  if (tmp_1 != 0) {
    zcDisabled = error_buffer_is_empty(rtmGetErrorStatus(RR_M));
    if (zcDisabled) {
      msg = rtw_diagnostics_msg(diagnosticTree);
      rtmSetErrorStatus(RR_M, msg);
    }
  }

  /* End of Start for SimscapeExecutionBlock: '<S24>/OUTPUT_1_0' */

  /* Start for SimscapeExecutionBlock: '<S24>/STATE_1' */
  simulator = nesl_lease_simulator(
    "RR/Processing/Mechnical Model /Solver Configuration_1", 0, 0);
  RR_DW.STATE_1_Simulator = (void *)simulator;
  zcDisabled = pointer_is_null(RR_DW.STATE_1_Simulator);
  if (zcDisabled) {
    RR_836bb176_1_gateway();
    simulator = nesl_lease_simulator(
      "RR/Processing/Mechnical Model /Solver Configuration_1", 0, 0);
    RR_DW.STATE_1_Simulator = (void *)simulator;
  }

  tmp = nesl_create_simulation_data();
  RR_DW.STATE_1_SimData = (void *)tmp;
  diagnosticManager = rtw_create_diagnostics();
  RR_DW.STATE_1_DiagMgr = (void *)diagnosticManager;
  modelParameters_0.mSolverType = NE_SOLVER_TYPE_DAE;
  modelParameters_0.mSolverAbsTol = 0.001;
  modelParameters_0.mSolverRelTol = 0.001;
  modelParameters_0.mSolverModifyAbsTol = NE_MODIFY_ABS_TOL_NO;
  modelParameters_0.mStartTime = 0.0;
  modelParameters_0.mLoadInitialState = false;
  modelParameters_0.mUseSimState = false;
  modelParameters_0.mLinTrimCompile = false;
  modelParameters_0.mLoggingMode = SSC_LOGGING_ON;
  modelParameters_0.mRTWModifiedTimeStamp = 6.36060226E+8;
  modelParameters_0.mUseModelRefSolver = false;
```

```
    modelParameters_0.mTargetFPGAHIL = false;
    tmp_0 = 0.001;
    modelParameters_0.mSolverTolerance = tmp_0;
    tmp_0 = 0.2;
    modelParameters_0.mFixedStepSize = tmp_0;
    zcDisabled = false;
    modelParameters_0.mVariableStepSolver = zcDisabled;
    zcDisabled = false;
    modelParameters_0.mIsUsingODEN = zcDisabled;
    modelParameters_0.mZcDisabled = true;
    simulator = (NeslSimulator *)RR_DW.STATE_1_Simulator;
    diagnosticManager = (NeuDiagnosticManager *)RR_DW.STATE_1_DiagMgr;
    diagnosticTree = neu_diagnostic_manager_get_initial_tree(diagnosticManager);
    tmp_1 = nesl_initialize_simulator(simulator, &modelParameters_0,
      diagnosticManager);
    if (tmp_1 != 0) {
      zcDisabled = error_buffer_is_empty(rtmGetErrorStatus(RR_M));
      if (zcDisabled) {
        msg = rtw_diagnostics_msg(diagnosticTree);
        rtmSetErrorStatus(RR_M, msg);
      }
    }

    /* End of Start for SimscapeExecutionBlock: '<S24>/STATE_1' */
  }
}

/* Model terminate function */
void RR_terminate(void)
{
  NeslSimulationData *simulationData;
  NeuDiagnosticManager *diagnosticManager;

  /* Terminate for SimscapeExecutionBlock: '<S24>/OUTPUT_1_0' */
  diagnosticManager = (NeuDiagnosticManager *)RR_DW.OUTPUT_1_0_DiagMgr;
  neu_destroy_diagnostic_manager(diagnosticManager);
  simulationData = (NeslSimulationData *)RR_DW.OUTPUT_1_0_SimData;
  nesl_destroy_simulation_data(simulationData);
  nesl_erase_simulator("RR/Processing/Mechnical Model /Solver Configuration_1");
  nesl_destroy_registry();

  /* Terminate for SimscapeExecutionBlock: '<S24>/STATE_1' */
  diagnosticManager = (NeuDiagnosticManager *)RR_DW.STATE_1_DiagMgr;
  neu_destroy_diagnostic_manager(diagnosticManager);
  simulationData = (NeslSimulationData *)RR_DW.STATE_1_SimData;
  nesl_destroy_simulation_data(simulationData);
  nesl_erase_simulator("RR/Processing/Mechnical Model /Solver Configuration_1");
  nesl_destroy_registry();
}
```