多阶段动力学建模框架

A Multi-Stage Dynamic Modeling Framework

姓名 朱明仁

学号 3120200445

学院 机械与车辆学院

日期 May 17, 2021

北京理工大學
BEIJING INSTITUTE OF TECHNOLOGY

Contents

摘 要

在有关系统动力学的学习和研究过程中，我们经常遇到多阶段的动力学建模与仿真问题，例如过程中系统发生碰撞导致系统状态发生突变或者系统在不同的状态下经历不同的力场作用等等。然而，我们却缺乏灵活方便的计算机辅助工具来对这一问题进行方便的表达和有效的解决。本文提出了一种对多阶段动力学进行建模的概念框架，并基于 Python 对其进行了实现，以方便学生或研究人员对这一问题进行学习和研究。本文的主要工作包括：（1）对多阶段动力学过程进行概念上的梳理；（2）基于 Python 对多阶段动力学过程的建模框架进行实现，并阐述其接口与技术原理；（3）为该建模框架提供了丰富且详尽的实际案例说明。

关键字: 系统动力学，多阶段过程，建模仿真框架，Python

Abstract

When studying the system dynamics, we often meet problems about multi-stage dynamic modeling and simulation, such as a crush lead to the saltation of system states and different force fields due to the different systems states. However, we lack of flexible and convenient computer-aid tools to express and solve such problems. Here we proposed a conception framework to help with modeling multi-stage dynamics and built it in code with Python for students and researchers. Following is the main work about the paper: (1) we carried out a conception comb of multi-stage dynamic processes; (2) we implemented the modeling framework of multi-stage dynamic processes based on Python and detailed the APIs and technical principles; (3) we provided rich case studies for the modeling framework.

Keywords: System Dynamics, Multi-Stage Processes, Modeling and Simulation Framework, Python

1. 多阶段动力学过程

多阶段动力学过程是指系统的状态变化不能用单一的动力学方程描述的过程。引起这种多阶段现象的原因可以分为两类：系统的状态突变与系统的动力学突变。状态突变是指系统的状态受到来自系统动力学之外的因素影响而带来的变化，这种现象在各种动力学过程中十分常见：例如物理系统在连续运动过程中发生的碰撞可以带来系统状态的突变，又例如量子系统受到外界测量行为的影响从而引发的波函数坍缩，再例如生物种群遭受到某种灾变而导致的物种数目锐减，等等。动力学突变则是指系统的动力学发生的变化，这种现象往往发生在系统所处的环境发生变化的情况下，例如物理粒子从一个力场进入另一个力场或者人口政策的变化导致人口增长规律的变化等等。状态突变和动力学突变两者之间并不是互斥的，即系统状态发生突变的同时其动力学亦可发生突变。总之，系统的动力学过程以多阶段呈现的现象是普遍的，从而多阶段动力学过程的问题也是普遍的，这也是我们提出多阶段动力学过程建模框架的实际需求背景。

1.1 动力学阶段及其属性

在多阶段动力学建模框架中，我们把系统未发生状态突变和动力学突变的一段连续过程称为一个动力学阶段。系统从一个动力学阶段进入另一个动力学阶段的过程我们称之为系统动力学阶段的跃迁。在一个动力学阶段中系统遵循且仅遵循唯一一个系统动力学方程的支配，在连续系统中可以表达为：

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, t) \tag{1.1}$$

其中 $\mathbf{x}$ 为系统的状态向量；$\dot{\mathbf{x}}$ 为系统状态向量对时间的一阶导数；$\mathbf{f}$ 为系统该阶段的动力学函数，是一个与 $\mathbf{x}$ 同维度的向量函数；t 为时间变量。当 $\mathbf{f}$ 为 $\mathbf{x}$ 的线性函数时，该系统阶段称为线性阶段，否则称为非线性阶段；当 $\mathbf{f}$ 中不显含 t 时，该阶段称为定常阶段或时不变阶段，否则称为非定常阶段或时变阶段。如果我们对阶段进行编号，那么阶段 i 的动力学方程可以表达为 $\dot{\mathbf{x}} = \mathbf{f}_i(\mathbf{x}, t)$。离散系统的动力学方程表达为差分形式即可。

当系统的状态或者动力学发生突变时，我们称系统到达了阶段的边界，阶段边界可以使用边界方程进行表达：

$$g(\mathbf{x}, t) = 0 \tag{1.2}$$

其中 g 为系统的边界函数，是一个标量函数。而当系统处于阶段内部时则可以将状态范围用一个不等式表达，我们在所提框架中使用小于：

$$g(\mathbf{x},t) < 0 \tag{1.3}$$

事实上，系统的边界可能不止一个，不同的边界可能触发不同的状态突变或动力学突变，系统从而跃迁到不同的阶段。阶段 i 到阶段 j 的边界可以表达为 $g_{ij}(\mathbf{x},t) = 0$，多边界阶段的内部则可以使用一组不等式 $\{g_{ij}(\mathbf{x},t) < 0 | j \in \mathscr{C}_i\}$ 来表达，其中 $\mathscr{C}_i$ 代表阶段 i 的后继阶段集合。

当系统到达阶段的边界时，可能触发系统状态突变或动力学突变。动力学突变可以通过更改动力学方程来表达，状态突变则需要使用状态突变方程来表达：

$$\mathbf{x}^+ = \mathbf{h}(\mathbf{x}^-,t) \tag{1.4}$$

其中 $\mathbf{x}^-$ 为突变前的系统状态向量；$\mathbf{x}^+$ 为突变后的系统状态变量；$\mathbf{h}$ 为状态突变函数，是一个与 $\mathbf{x}$ 同维度的向量函数。系统到达阶段的不同边界可能触发不同的突变，阶段 i 到阶段 j 的状态突变可以表达为 $\mathbf{x}^+ = \mathbf{h}_{ij}(\mathbf{x}^-,t)$。

我们把阶段 i 的动力学函数 $\mathbf{f}_i$、后继阶段集 $\mathscr{C}_i$、边界函数集 $\mathscr{G}_i = \{g_{ij} | j \in \mathscr{C}_i\}$ 以及状态突变函数集 $\mathscr{H}_i = \{\mathbf{h}_{ij} | j \in \mathscr{C}_i\}$ 称为阶段 i 的属性。当且仅当两个阶段的四个属性完全相同时，我们称这两个阶段同态。类似于量子力学中的同态粒子，同态的阶段完全相同，不可也不需区分。有了同态的概念之后，我们就不需要按照时间进度来区分动力学阶段。当系统跳出某个动力学阶段之后完全可以再次回到该阶段，而不需要仅仅因为时间的不同而另外新建一个完全相同的阶段模型。

1.2 多阶段动力学及其拓扑结构

从上一节的讨论中可知，系统动力学的某个阶段 i 具备四个属性，即动力学函数 $\mathbf{f}_i$、后继阶段集 $\mathscr{C}_i$、边界函数集 $\mathscr{G}_i$ 以及状态突变函数集 $\mathscr{H}_i$。对于具有 N 个阶段的动力学过程而言，我们可以将其表示如下：

(1) 动力学向量：

$$F = \begin{pmatrix} \mathbf{f}_1 \\ \mathbf{f}_2 \\ \vdots \\ \mathbf{f}_N \end{pmatrix} \tag{1.5}$$

(2) 边界矩阵：

$$G = \begin{pmatrix} g_{11} & g_{12} & \cdots & g_{1N} \\ g_{21} & g_{22} & \cdots & g_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ g_{N1} & g_{N2} & \cdots & g_{NN} \end{pmatrix} \tag{1.6}$$

其中对于任意 $i \in \mathscr{N}$，$\mathscr{N}$ 为阶段的编号集，如果 $j \notin \mathscr{C}_i$，那么 g_{ij} 将不存在，我们使用 X 表示。

(3) 突变矩阵：

$$H = \begin{pmatrix} \mathbf{h}_{11} & \mathbf{h}_{12} & \cdots & \mathbf{h}_{1N} \\ \mathbf{h}_{21} & \mathbf{h}_{22} & \cdots & \mathbf{h}_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{h}_{N1} & \mathbf{h}_{N2} & \cdots & \mathbf{h}_{NN} \end{pmatrix} \tag{1.7}$$

同边界函数矩阵 G，如果 $\mathbf{h}_{ij}$ 不存在，我们使用 X 来表示。

系统的多阶段动力学可表述如下：系统运行在阶段 i 时，以动力学 $\mathbf{f}_i$ 持续运行，直到系统到达阶段边界 g_{ij}，触发状态突变 h_{ij}，从而系统进入阶段 j，以此往复。应当指出的是，在某些情况下系统到达阶段边界并不会触发显示的状态突变，这意味着 $\mathbf{x}^+ = \mathbf{x}^-$，以这种形式存在的状态突变称为哑突变，相应的状态突变函数为恒等变换，称为哑突变函数。同理，某些边界情况也并不会触发动力学突变而只会触发状态突变，这时系统到达的边界为 g_{ii}，触发的状态突变为 $\mathbf{h}_{ii}$，这就要求突变函数矩阵 H 的对角元素 $\mathbf{h}_{ii}$ 均不能是哑突变函数，因为动力学阶段的跃迁必须经历状态突变或动力学突变其中的至少一个。

进一步地，我们可以使用一个包含 N 个节点的有向图来表达一个 N 阶段动力学过程。我们将每个动力学函数 $\mathbf{f}_i$ 视为节点，将每个边界函数与突变函数二元组 $(g_{ij}, \mathbf{h}_{ij})$ 视为节点 i 与节点 j 的一条有向弧。值得指出的是，借由这种有向图表达的多阶段动力学过程可以自然避开不存在的边界函数和突变函数，而不需要接触额外的符号。由此我们可以得到多阶段动力学的拓扑结构：

$$D = (V, E) \tag{1.8a}$$

$$V = \{\mathbf{f}_i | i \in \mathscr{N}\} \tag{1.8b}$$

$$E = \{(g_{ij}, \mathbf{h}_{ij}) | \forall j \in \mathscr{C}_i, \forall i \in \mathscr{N}\} \tag{1.8c}$$

拓扑结构的建立使得我们利用图论的工具来理解和解决多阶段动力学问题成为可能。应当指出的是，虽然动力学过程拥有一定数目的阶段，但是系统在实际运行或仿真的过程中可能受到初始值、时间区间或随机效应等因素的影响并不一定会将所有的阶段

都执行到。我们将系统一次运行过程中所遍历的动力学阶段序列 $\{(\mathbf{f}_k, g_{k,k+1}, \mathbf{h}_{k,k+1})_{k=0}^{K-1}\}$ 称为一条动力学路径。

1.3 多阶段动力学过程示例

接下来我们使用一个简单但是足够充分的例子对上述多阶段动力学过程框架进行说明。考虑如图 1.1 所示的例 1，一个小球在天花板和底部弹簧之间来回运动：小球质量为 m，坐标为 x，速度为 v，状态向量为 $\mathbf{s} = (x, v)^T$；弹簧保持原长时高度为 0，劲度系数为 k；天花板高度为 h，小球与其发生碰撞的恢复系数为 η；运动阻尼系数为 c；重力加速度为 g。

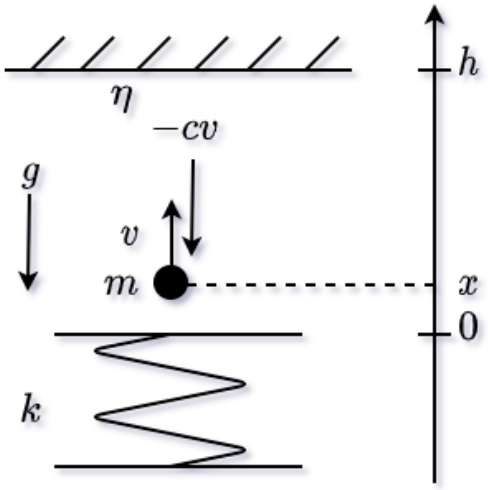


图 1.1: 在弹簧和天花板之间往复运动的小球（例 1）

根据动力学方程和边界进行划分，该系统具有两个动力学阶段，阶段 1 是小球在空中受重力的动力学阶段，阶段 2 是小球落地与弹簧作用的动力学阶段。

阶段 1 的动力学方程为：

$$\dot{x} = v \tag{1.9a}$$

$$\dot{v} = -\frac{c}{m}v - g \tag{1.9b}$$

阶段 1 有两个边界，首先是触发与天花板碰撞的边界：

$$x - h = 0 \tag{1.10}$$

此处为保证满足阶段内部用小于不等式的表达 $x-h<0$ 而将边界写成了 $x-h=0$ 而非 $h-x=0$，下同。当系统到达该边界时将会触发碰撞的状态突变：

$$x^+ = x^- \tag{1.11a}$$
$$v^+ = -\eta v^- \tag{1.11b}$$

完成状态突变后系统将重新回到阶段 1。

阶段 1 的第二个边界则是与弹簧开始作用：

$$-x = 0 \tag{1.12}$$

当系统到达该边界时不会触发显示的状态突变，因此此处的状态为哑突变：

$$x^+ = x^- \tag{1.13a}$$
$$v^+ = v^- \tag{1.13b}$$

之后系统将进入阶段 2。

阶段 2 的动力学方程为：

$$\dot{x} = v \tag{1.14a}$$
$$\dot{v} = -\frac{k}{m}x - \frac{c}{m}v - g \tag{1.14b}$$

阶段 2 的边界为：

$$x = 0 \tag{1.15}$$

注意此处与阶段 1 到阶段 2 的边界是不同的，因为阶段 2 的内部用小于不等式表达为 $x<0$。

该边界同样触发哑变换：

$$x^+ = x^- \tag{1.16a}$$
$$v^+ = v^- \tag{1.16b}$$

之后系统将进入阶段 1。至此，该系统动力学的两个阶段分别分析完成。

该系统多阶段动力学的矩阵表达如下：

(1) 动力学向量：

$$F = \begin{pmatrix} Eq.(1.9) \\ Eq.(1.14) \end{pmatrix} \tag{1.17}$$

(2) 边界矩阵：

$$G = \begin{pmatrix} Eq.(1.10) & Eq.(1.12) \\ Eq.(1.15) & X \end{pmatrix} \tag{1.18}$$

(3) 突变矩阵：

$$H = \begin{pmatrix} Eq.(1.11) & Eq.(1.13) \\ Eq.(1.16) & X \end{pmatrix} \tag{1.19}$$

拓扑结构如图 1.2 所示：

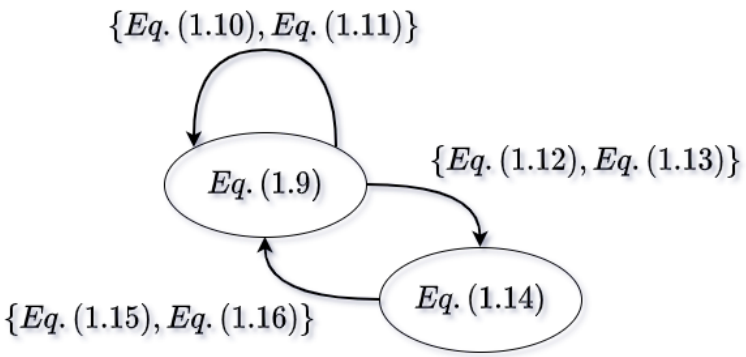


图 1.2: 拓扑结构（例 1）

至此，我们已经完成了多阶段动力学建模的概念框架并进行了示例说明，接下来我们将对其进行编程实现，介绍 Python 接口及其技术原理。

2. 代码框架与技术说明

我们首先利用已完成的代码框架对图 1.1 所示的例 1 进行建模并运行仿真，熟悉操作流程，然后再进入代码框架的详细说明。例 1 中的系统参数设置如下：$m=1$，$c=0.5$，$g=9.8$，$k=10$，$h=5$，$\eta=0.8$。

2.1 框架使用流程示范

首先引入数组库 Numpy[1]、绘图库 Matplotlib[2] 以及多阶段动力学建模库 MultiConditionalDynamic，并设置系统参数：

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  from MultiConditionalDynamic import *
4
5  # 设置系统参数
6  m = 1.
7  c = 0.5
8  g = 9.8
9  k = 10.
10 h = 5.
11 eta = 0.8
```

然后分别建立阶段 1 和阶段 2 的各个属性。阶段 1 建模：

```
1  # 动力学方程 Eq.(1.9)
2  def dynamic1(t, x):
3      return np.array([x[1], -c*x[1]/m-g])
4
5  # 边界函数 Eq.(1.10)
6  def event11(t, x):
```

```
    return x[0] - h

# 突变函数 Eq.(1.11)
def trans11(t, x):
    return np.array([x[0], -eta*x[1]])

# 边界函数 Eq.(1.12)
def event12(t, x):
    return -x[0]

# 突变函数 Eq.(1.13)
def trans12(t, x):
    return x

# 阶段 1
MCDynamic1 = MultiConditionalDynamic(
        dynamic1, [event11, event12], [trans11, trans12]
)
```

阶段 2 建模:

```
# 动力学方程 Eq.(1.14)
def dynamic2(t, x):
    return np.array([x[1], -k*x[0]/m-c*x[1]/m-g])

# 边界函数 Eq.(1.15)
def event21(t, x):
    return x[0]

# 突变函数 Eq.(1.16)
def trans21(t, x):
    return x

# 阶段 2
MCDynamic2 = MultiConditionalDynamic(
        dynamic2, [event21], [trans21]
)
```

最后根据各个阶段之间的拓扑关系进行链接：

```
MCDynamic1.nxts = [MCDynamic1, MCDynamic2]
MCDynamic2.nxts = [MCDynamic1]
```

此处需要注意的是，各个阶段的边界函数列表、突变函数列表以及后继阶段列表必须保持顺序的一致。至此，我们就完成了例 1 所示的多阶段动力学过程建模。接下来我们对其进行一次仿真运行。

```
# 初值设置
x0 = np.array([0, 100])

# 时间区间设定
t_span = [0, 20]

# 采样频率设定
t_density = 1000

# 运行
res = MCDynamic1.run(x0, t_span, t_density)

# 绘图
fig = plt.figure(figsize=(10, 4))
ax = fig.add_subplot(111)
l1 = ax.plot(res[0], res[1][0, :], "r-", label="x")
ax2 = ax.twinx()
l2 = ax2.plot(res[0], res[1][1, :], "b-", label="v")
ls = l1 + l2
lbs = [l.get_label() for l in ls]
ax.legend(ls, lbs, fontsize=20)
ax.set_xlabel("Time", fontsize=20)
ax.set_ylabel("Position-x", fontsize=20)
ax2.set_ylabel("Speed-v", fontsize=20)
ax.tick_params(labelsize=20)
ax2.tick_params(labelsize=20)
ax.grid()
plt.show()
```

仿真结果如图 2.1 所示，其中红色线代表位置 x 的变化（左轴），蓝色线代表速度 v 的变化（右轴），横轴代表时间：

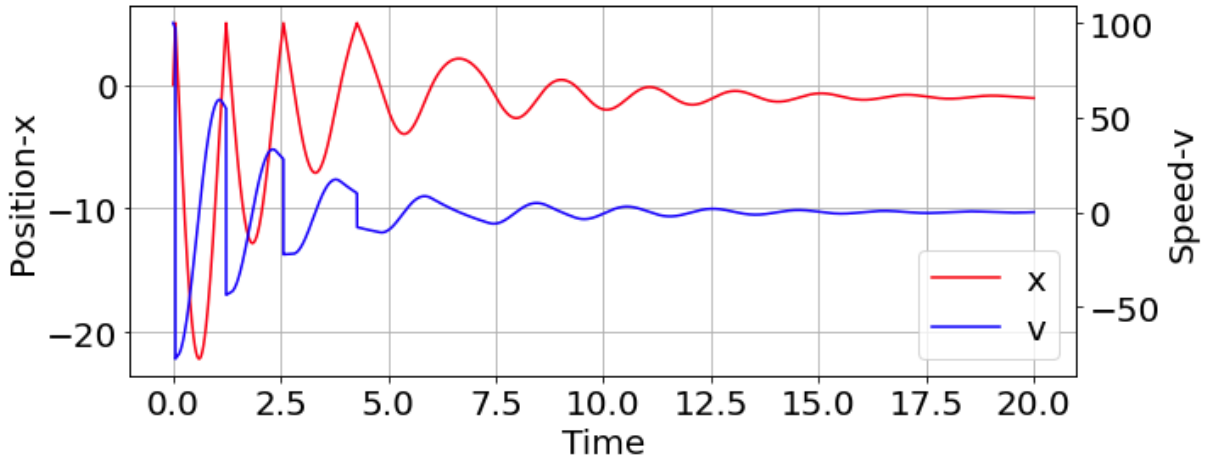


图 2.1: 例 1 仿真结果

从图 2.1 中我们可以看到小球在与天花板碰撞 4 次之后就已经无法达到天花板的高度了，在约 10 个时间单位之后也不再能回到弹簧之上，只能随着弹簧做阻尼震荡，并最终趋于平衡。此次仿真过程中系统的动力学路径为 $\{1,1,2,1,1,2,1,1,2,1,1,2,1,2,1,2\}$。

2.2 框架接口与技术说明

从上一节的框架使用示范中我们知道，使用框架对多阶段动力学过程进行建模与仿真仅需三步：首先建立各个阶段的动力学方程、边界函数以及突变函数，随后根据拓扑关系将各个阶段进行链接，最后设置初始状态和仿真参数运行仿真。其中多阶段建模和仿真的过程仅需要寥寥几行代码就能实现，并且逻辑清晰直观，理解起来毫无困难，足见该框架的简洁易用。

接下来我们对该框架的核心对象 MultiConditionalDynamic 及其属性与方法进行讲解。MultiConditionalDynamic 类用于描述动力学阶段，其构建函数如下：

```
1  class MultiConditionalDynamic:
2
3      def __init__(self, dynamic = None, events = None, transes = None, nxts
           = None):
4          self.dynamic = dynamic
5          self.events = events
6          self.transes = transes
7          self.nxts = nxts
```

可以看到其具备 4 个属性：dynamic、events、transes、nxts，分别对应着动力学阶段的动力学方程 $\mathbf{f}$、边界函数集 $\mathscr{G}$、突变函数集 $\mathscr{H}$ 以及后继阶段集 $\mathscr{C}$。需要注意的是，除了 dynamic 属性之外，其余三个属性类型均为 Python 的列表形式，dynamic 和 events、transes 内部的元素类型均为 Python 的函数。这些函数接收的参数均为 (t,x) 的二元组，其中 t 代表时间参数，为一个浮点类型的标量；而 x 代表状态向量参数，为 Numpy 的一维浮点数组。nxts 为后继阶段列表，其中的元素为 MultiConditionalDynamic 的类实例。

MultiConditionalDynamic 类的一个核心方法为 run 方法，其接受参数如下：

```
1  class MultiConditionalDynamic:
2
3      def run(self, x0, t_span, t_density = 1000):
4                      ......
```

其中 x0 代表开始运行的状态初值，为一个 Numpy 的一维浮点数组；t_span 代表运行的时间区间，为一个包含两个元素的 Python 列表，分别代表运行的开始时刻与结束时刻；t_density 代表采样频率，即一个时间单位内采样仿真数据的次数，如果用户设定的时间以秒为单位的话，那么 t_density 的单位就是赫兹，该参数默认为 1000。

run 方法的使用也很简单，在搭建完成多阶段动力学模型之后，确定状态初值所在的动力学阶段，对该阶段相应的 MultiConditionalDynamic 实例运行 run 方法即可，系统内部发生的阶段跃迁将由该框架自动完成。run 方法的返回结果是一个包含两个元素的 Python 列表，其中第一个元素为仿真采样的一维时间数组，由 t_span 和 t_density 决定；第二个元素为采样到的二维状态数组，第一个维度为状态向量的维度，第二个维度为采样时间的维度。

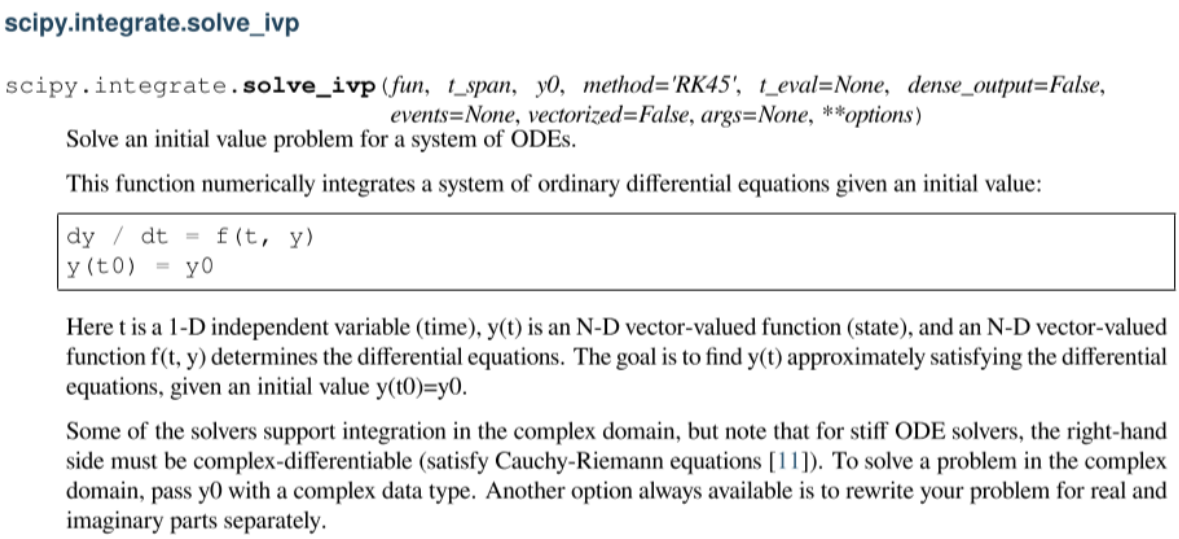
为了实现对阶段动力学方程的积分以及边界事件的监听，我们在 run 方法中引用了 Python 科学计算库 Scipy[3] 中的一个核心算子 solve_ivp：

```
1  class MultiConditionalDynamic:
2
3      def run(self, x0, t_span, t_density = 1000):
4                      ......
5                      sol = solve_ivp(self.dynamic, t_span, x0, t_eval = t_eval,
                           events = events)
6                      ......
```

solve_ivp 是 Scipy 中专门用于解决初值问题的一个积分算子，可选多种积分算法，具备事件监听机制，因此可以用于监测动力学阶段的边界。事实上，solve_ivp 算子在运行结束后会返回一个事件监听结果，如果积分的中断是由用户事件引起的，那么可以通过特

定方式查询到引发中断的具体事件，从而确定动力学阶段的触发边界，完成多阶段动力学内部的阶段跃迁。

Scipy 官方文档对 solve_ivp 算子的描述如图 2.2 所示：



scipy.integrate.solve_ivp

scipy.integrate.**solve_ivp**(*fun*, *t_span*, *y0*, *method='RK45'*, *t_eval=None*, *dense_output=False*, *events=None*, *vectorized=False*, *args=None*, ***options*)

Solve an initial value problem for a system of ODEs.

This function numerically integrates a system of ordinary differential equations given an initial value:

```
dy / dt = f(t, y)
y(t0) = y0
```

Here t is a 1-D independent variable (time), y(t) is an N-D vector-valued function (state), and an N-D vector-valued function f(t, y) determines the differential equations. The goal is to find y(t) approximately satisfying the differential equations, given an initial value y(t0)=y0.

Some of the solvers support integration in the complex domain, but note that for stiff ODE solvers, the right-hand side must be complex-differentiable (satisfy Cauchy-Riemann equations [11]). To solve a problem in the complex domain, pass y0 with a complex data type. Another option always available is to rewrite your problem for real and imaginary parts separately.

图 2.2: solve_ivp 算子的官方描述

有关 solve_ivp 的详情可点击图 2.2 的图题进入 solve_ivp 的官方页面。

3. 复杂系统的多阶段动力学建模

在对例 1 所示的多阶段动力学过程进行建模之后，我们接下来再对更加复杂一点的系统进行其多阶段动力学的建模，并强调另一建模工具与多阶段动力学框架的集成运用，证明多阶段动力学框架良好的可拓展性。需要指出的是，以下给出的两个示例中所采用的多阶段动力学建模框架主体为 ConditionalDynamic，而不再是 MultiConditionalDynamic。两者之间的区别在于前者针对的是所有阶段仅具有单一后继的多阶段动力学过程，因此其适用于具有链状拓扑或环状拓扑的多阶段动力学过程，而接下来所展示的示例的拓扑结构均为自环（后继阶段仅仅为自身的拓扑结构）。

3.1 定点旋转的二连杆碰撞过程

如图 3.1 所示，一个二连杆机构的一端被固定在墙上，另一端自由，两个连杆的质量分别为 m_1、m_2，长度分别为 l_1、l_2。如图建立广义坐标 q_1、q_2 和广义速度 u_1、u_2 分别代表两杆的旋转角度与角速度。连杆固定点距离地面高度为 h，与地面碰撞恢复系数为 η，重力加速度为 g。

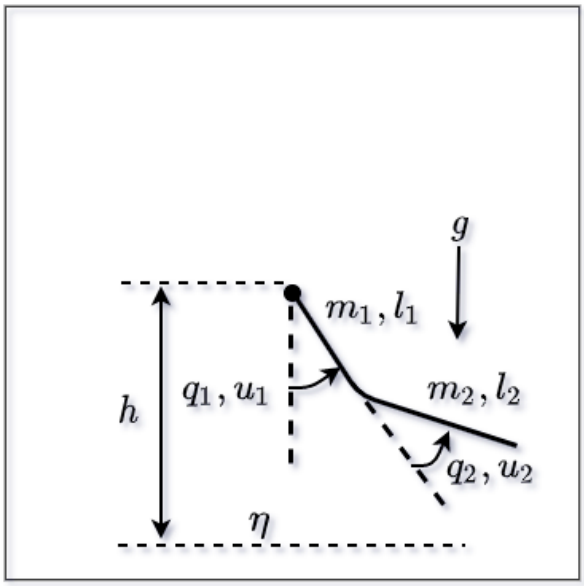


图 3.1: 定点旋转的二连杆（例 2）

该系统具有的动力学阶段数目为一个，但是由于存在碰撞过程导致的状态突变，因此仍属于多阶段动力学问题。为了建立系统的多阶段动力学模型，首先必须得到阶段的动力学方程。虽然我们可以对该系统进行一定程度的计算来得到其动力学方程，但是对于更为复杂的系统而言人力计算就会失效，因此我们在此引入另一个 Python 科学计算的建模工具 Sympy[4]，更准确地说，是 Sympy 中的 sympy.physics.mechanics 模块。Sympy 是一个 Python 语言的符号计算库，其内含的 Mechanics 模块则是对经典物理中的多体动力学 [5] 进行建模的有效工具。应当在此指出的是，多体动力学所指的是由多个刚体组成的系统动力学，与多阶段动力学的概念并不重叠。相反，两种动力学分类相辅相成，使得物理系统具有更加细致精细的结构。

首先引入需要的库文件：

```
1 import numpy as np
2 import scipy.linalg as linalg
3 import matplotlib.pyplot as plt
4 import matplotlib.animation as anim
5 from sympy import *
6 from sympy.physics.mechanics import *
7 from ConditionalDynamic import *
```

接着对系统进行多体动力学建模。以符号形式设定系统参数：

```
1 m1, m2, l1, l2, g = symbols("m1, m2, l1, l2, g")
2 q1, q2, u1, u2 = dynamicsymbols("q1, q2, u1, u2")
3 q1d, q2d, u1d, u2d = dynamicsymbols("q1, q2, u1, u2", 1)
```

建立多体运动学模型：

```
1 N = ReferenceFrame("N")
2 O = Point("O")
3 O.set_vel(N, 0)
4 A = N.orientnew("A", "axis", [q1, N.x])
5 C1 = O.locatenew("C1", Rational(1, 2)*l1*A.y)
6 P = O.locatenew("P", l1*A.y)
7 B = A.orientnew("B", "axis", [q2, A.x])
8 C2 = P.locatenew("P", Rational(1, 2)*l2*B.y)
9 Q = P.locatenew("Q", l2*B.y)
```

以 Kane 方法 [6] 建立多体动力学模型：

```
I1 = Rational(1, 3)*m1*l1**2*(outer(A.x, A.x)+outer(A.z, A.z))
OP = RigidBody("OP", C1, A, m1, (I1, C1))
I2 = Rational(1, 3)*m2*l2**2*(outer(B.x, B.x)+outer(B.z, B.z))
PQ = RigidBody("PQ", C2, B, m2, (I2, C2))
BodyList = [OP, PQ]
ForceList = [(C1, m1*g*N.y), (C2, m2*g*N.y)]
kane = KanesMethod(N, q_ind = [q1, q2], u_ind = [u1, u2], kd_eqs = [q1d -
    u1, q2d - u2])
f, f_star = kane.kanes_equations( ForceList , BodyList)
```

以数值形式设置系统参数并求解二阶加速度：

```
constants = {
    m1: 1.,
    m2: 1.,
    I1: 1.,
    I2: 1.,
    g: 9.8
}
h, eta = 1.5, 0.8
qdds = solve((f+f_star).subs(constants), [u1.diff(), u2.diff()])
```

至此，我们求解了系统的多体动力学，可以对其进行多阶段动力学建模。动力学方程：

```
def dynamic(t, x):
    dx1 = x[2]
    dx2 = x[3]
    p_dict = {q1: x[0], q2: x[1], u1: x[2], u2: x[3]}
    dx3 = qdds[u1.diff()].subs(p_dict)
    dx4 = qdds[u2.diff()].subs(p_dict)
    return np.array([dx1, dx2, dx3, dx4])
```

边界函数：

```
def event(t, x):
    return np.cos(x[0])+np.cos(x[0]+x[1])-h
```

突变函数：

```
def trans(t, x):
    A = np.array([
        [-np.sin(x[0]) -np.sin(x[0]+x[1]), -np.sin(x[0]+x[1])],
        [np.cos(x[0]) +np.cos(x[0]+x[1]), np.cos(x[0]+x[1])]
    ])
    Ainv = np.linalg.inv(A)
    T = kinetic_energy(N, OP, PQ).subs(constants)
    T1 = T.diff(q1d)
    T2 = T.diff(q2d)
    q1d_new, q2d_new = symbols("q1d_new, q2d_new")
    sol = solve(Matrix([
            (Ainv[0, 1]*(T1.subs({q1d: q1d_new, q2d: q2d_new})-T1.subs({
                q1d: x[2], q2d: x[3]}))+Ainv[1, 1]*(T2.subs({q1d: q1d_new,
                q2d: q2d_new})-T2.subs({q1d: x[2], q2d: x[3]}))).subs({q1: x
                [0], q2: x[1]}),
            A[0, 0]*(q1d_new+eta*x[2])+A[0, 1]*(q2d_new+eta*x[3])
        ]), [q1d_new, q2d_new])
    return np.array([x[0], x[1], sol[q1d_new].evalf(), sol[q2d_new].evalf()
        ])
```

可以看到该二连杆系统碰撞突变函数的实现同样是比较困难的，事实上，这一步仍需要人工进行计算，在此我们使用的方法是 Lagrange 的碰撞方程。多阶段建模：

```
dyn = ConditionalDynamic(dynamic, event, trans)
```

ConditionalDynamic 类的实例在自环结构下可以省略链接不写。至此，我们完成了对定点旋转的二连杆与地面碰撞过程的多阶段动力学建模。最后的仿真也很简单：

```
x0 = np.array([np.pi/2, 0, 0, 0])
t_span = [0, 10]
res = dyn.run(x0, t_span)
```

将仿真的结果绘图：

```
plt.figure(figsize=(10, 7))
```

```
2  plt.subplot(2, 1, 1)
3  plt.plot(res[0], res[1][0, :], "r-", label = "q1")
4  plt.plot(res[0], res[1][1, :], "b-", label = "q2")
5  plt.legend()
6  plt.grid()
7  plt.subplot(2, 1, 2)
8  plt.plot(res[0], res[1][2, :], "r--", label = "u1")
9  plt.plot(res[0], res[1][3, :], "b--", label = "u2")
10 plt.legend()
11 plt.grid()
12 plt.show()
```

仿真结果如图 3.2 所示：

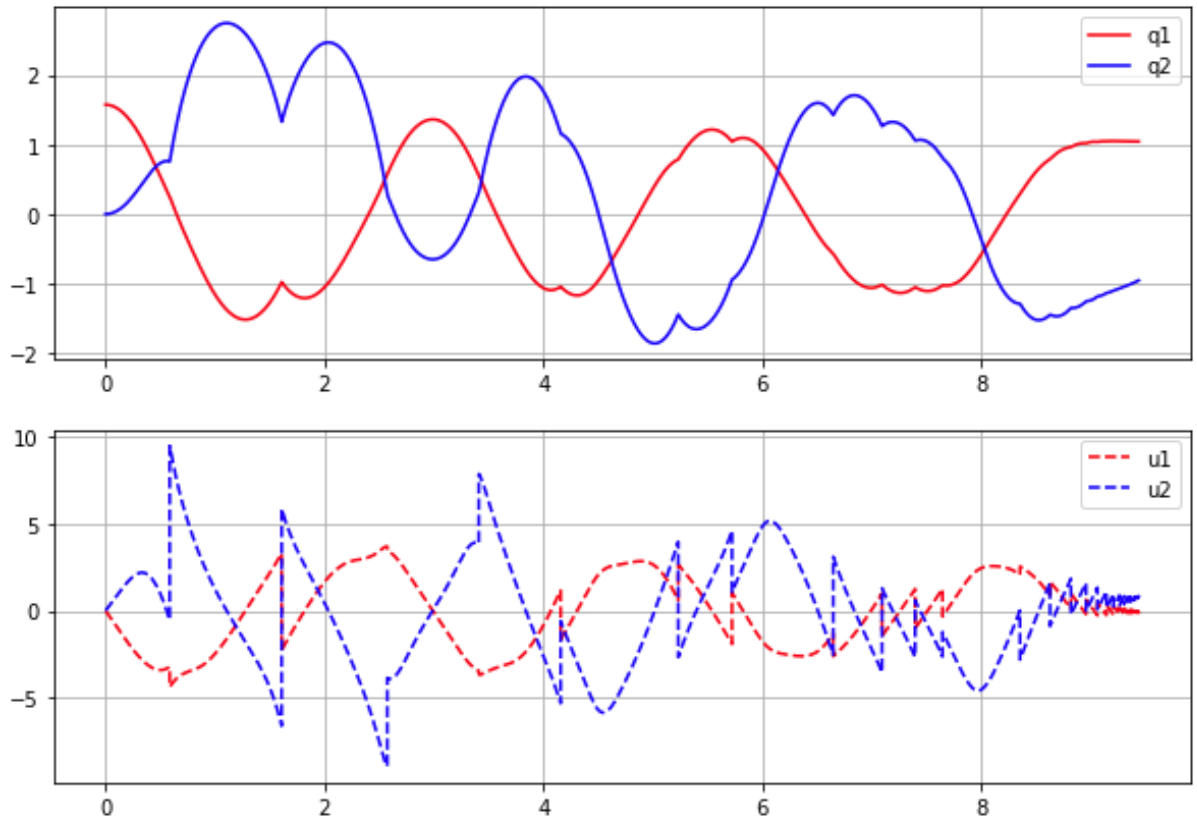


图 3.2: 例 2 仿真结果

我们还可以利用仿真数据生成动画，具体可参见附件。

3.2 旋转立方体下落-碰撞-弹起过程

如图 3.3 所示，一个立方体从高处旋转下落，与地面发生非弹性碰撞后又弹起，以此往复直到失去机械能。这同样属于一个多阶段动力学过程，具有自环的拓扑结构。系统的状态变量包括代表立方体位置的质心的 x、y、z 坐标和代表立方体姿态的欧拉角 ψ、θ、ϕ 及其各自的一阶导数速度项。

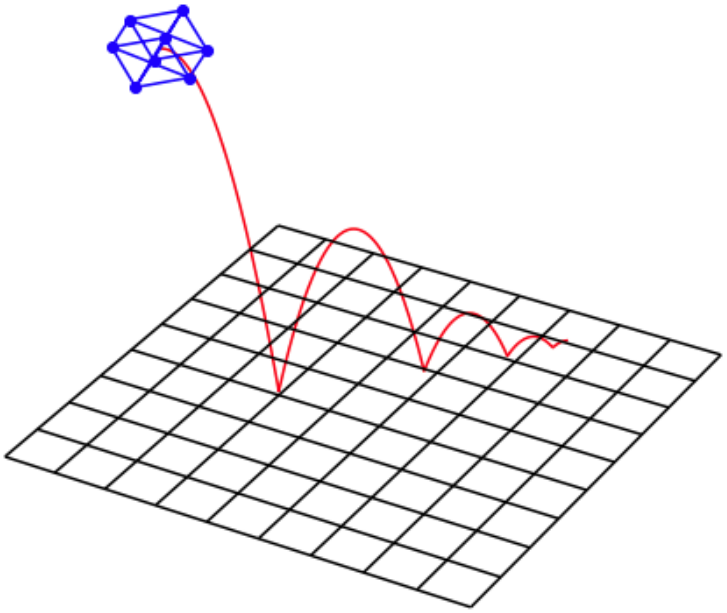


图 3.3: 旋转的立方体与地面碰撞（例 3）

我们同样利用 Sympy 对其进行多体动力学求解（详见附件），由此得到其动力学方程：

```
def dynamic(t, s):
    params = {theta: s[4], u1: s[9], u2: s[10], u3: s[11]}
    return np.array([
        s[6], s[7], s[8], s[9], s[10], s[11],
        sys[v1_d].subs(params),
        sys[v2_d].subs(params),
        sys[v3_d].subs(params),
        sys[u1_d].subs(params),
        sys[u2_d].subs(params),
        sys[u3_d].subs(params)
    ])
```

边界函数：

```
def event(t, s):
    params = {
        x: s[0], y: s[1], z: s[2],
        psi: s[3], theta: s[4], phi: s[5],
        x_d: s[6], y_d: s[7], z_d: s[8],
        psi_d: s[9], theta_d: s[10], phi_d: s[11]
    }
    buttom = min(
        [A, B, C, D, E, F, G, H],
        key = lambda p: me.dot(p.pos_from(O), N.z).subs(cons).subs(params)
    )
    return -me.dot(buttom.pos_from(O), N.z).subs(cons).subs(params).evalf()
```

可以看到该系统阶段的边界函数并不是一个常规意义上的“连续”函数，因为它有一个 *min* 算子取所有角点 z 坐标的最小值。这表明该多阶段动力学框架对边界函数的要求并不高，普适性良好。突变函数：

```
def trans(t, s):
        v1new, v2new, v3new, u1new, u2new, u3new = me.dynamicsymbols(
        "v1new, v2new, v3new, u1new, u2new, u3new"
    )

        d_old = {
        x: s[0], y: s[1], z: s[2],
        psi: s[3], theta: s[4], phi: s[5],
        x_d: s[6], y_d: s[7], z_d: s[8],
        psi_d: s[9], theta_d: s[10], phi_d: s[11]
    }

        d_new = {
        x: s[0], y: s[1], z: s[2],
        psi: s[3], theta: s[4], phi: s[5],
        x_d: v1new, y_d: v2new, z_d: v3new,
        psi_d: u1new, theta_d: u2new, phi_d: u3new
```

```
    }

    buttom = min(
        [A, B, C, D, E, F, G, H],
        key = lambda p: me.dot(p.pos_from(O), N.z).subs(cons).subs(d_old)
    )

    Tx = T.diff(x_d).subs(cons)
    Ty = T.diff(y_d).subs(cons)
    Tpsi = T.diff(psi_d).subs(cons)
    Ttheta = T.diff(theta_d).subs(cons)
    Tphi = T.diff(phi_d).subs(cons)
    dz = me.dot(buttom.pos_from(O), N.z).subs(cons). diff ()

        Eq = [
        Tx.subs(d_new) - Tx.subs(d_old),
        Ty.subs(d_new) - Ty.subs(d_old),
        Tpsi.subs(d_new) - Tpsi.subs(d_old),
        Ttheta.subs(d_new) - Ttheta.subs(d_old),
        Tphi.subs(d_new) - Tphi.subs(d_old),
        dz.subs(d_new) + eta*dz.subs(d_old)
    ]

    sol = sm.solve(Eq, [v1new, v2new, v3new, u1new, u2new, u3new])

    return np.array([
        s[0], s[1], s[2], s[3], s[4], s[5],
        sol[v1new].evalf(), sol[v2new].evalf(), sol[v3new].evalf(),
        sol[u1new].evalf(), sol[u2new].evalf(), sol[u3new].evalf()
    ])
```

多阶段建模与仿真:

```
dyn = ConditionalDynamic(dynamic, event, trans)
x0 = np.array([0., 0., 10., 1., 2., 3., 1., 1., 0., 3., 2., 1.])
t_span = [0, 5]
res = dyn.run(x0, t_span)
```

其结果如图 3.4 所示。

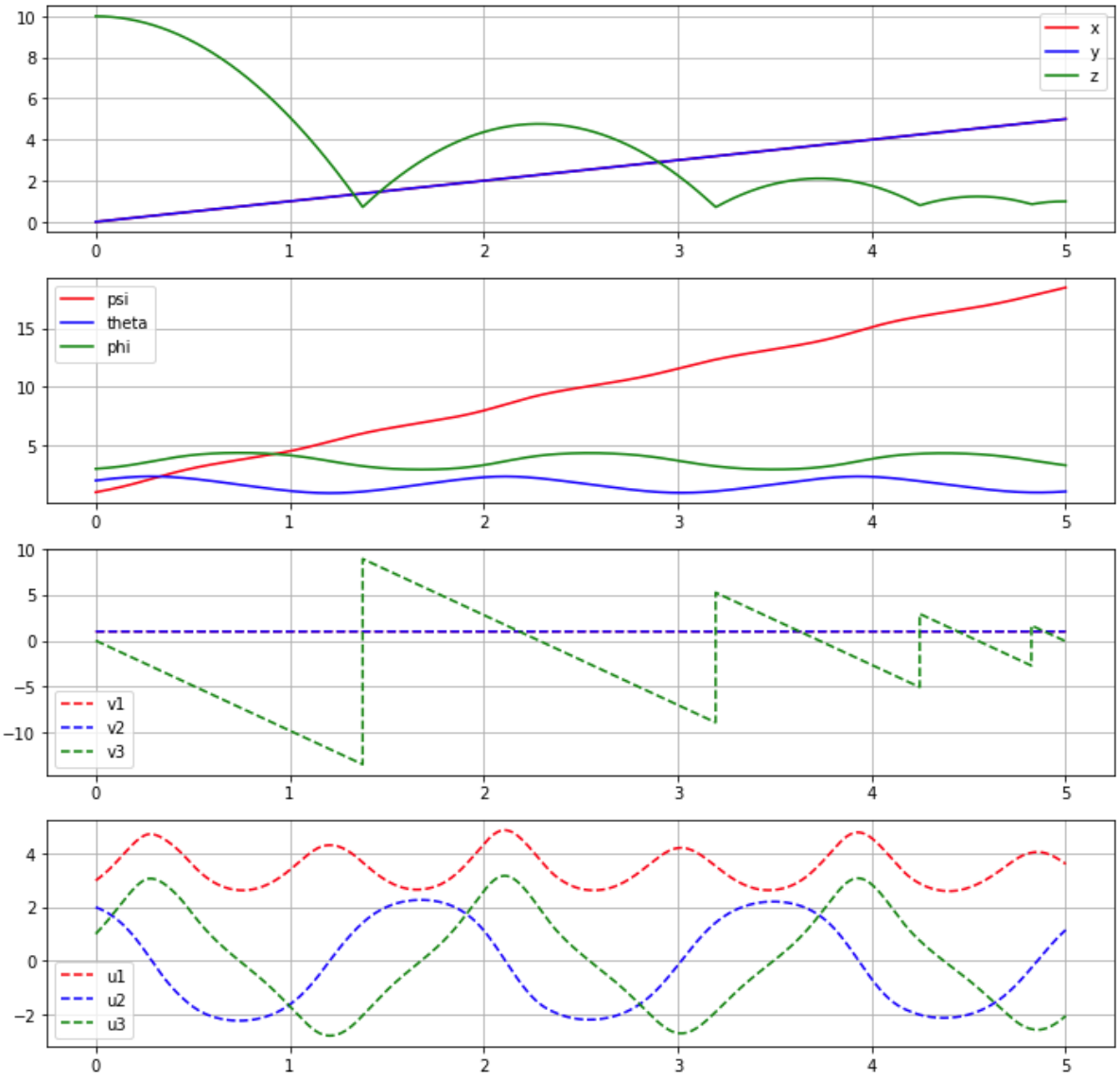


图 3.4: 例 3 仿真结果

4. 总结与展望

本文针对多阶段动力学问题提出的建模与仿真框架具备以下特点：

(1) 针对的问题具有明显的需求，受众广泛；

(2) 概念清晰明确，逻辑严谨缜密，有理论工具支撑；

(3) 基于 Python 的代码实现，使用方便，拓展性高。

但是，目前该项目仍处于初步建设阶段，也存在着不少的问题。项目未来的发展方向规划如下：

(1) 增加离散动力学处理机制，拓展应用范围；

(2) 建立标准模型库，自动处理常见物理对象及其约束；

(3) 建立自动控制算法与处理机制，使自动化建模、仿真、控制成为有机一体；

(4) 开发具备交互界面的软件系统，为不善于编程操作的学生和研究人员提供完善的服务。

路漫漫其修远兮，吾将上下而求索！愿这一套工具能为从事相关学习与研究的学生和科技工作者提供一些实质性的帮助，若得如此，夫复何求！

参考文献

[1] Charles Harris, K. Millman, Stéfan Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten Kerkwijk, Matthew Brett, Allan Haldane, Jaime Río, Mark Wiebe, Pearu Peterson, and Travis Oliphant. Array programming with numpy. 06 2020.

[2] Giovanni Moruzzi. *Plotting with Matplotlib*, pages 53–69. 06 2020.

[3] Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. Scipy 1.0–fundamental algorithms for scientific computing in python. *arXiv: Mathematical Software*, 2019.

[4] Aaron Meurer, Christopher Smith, Mateusz Paprocki, Ondřej Čertík, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian Granger, Richard Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew Curry, and Anthony Scopatz. Sympy: Symbolic computing in python. 05 2016.

[5] 王耀兵. 空间机器人. 北京理工大学出版社, 2018.

[6] 陈滨. 分析动力学（第二版）. 北京大学出版社, 2012.

附 录

附件说明

(1) MultiConditionalDynamic.py 为多阶段动力学建模核心代码

(2) ConditionalDynamic.py 为单后继多阶段动力学建模核心代码

(3) exampleX.ipynb 为各示例的可执行 jupyter notebook 文件

(4) exampleX.html 为 exampleX.ipynb 文件的复刻，可直接在浏览器打开，供审阅

(5) exampleX.gif 为各示例生成的仿真动画

(6) tmp.X 为另一些多阶段动力学建模框架的测试示例

MultiConditionalDynamic 类

```
1  import numpy as np
2  from scipy.integrate import solve_ivp
3
4
5  class MultiConditionalDynamic:
6
7      def __init__(self, dynamic = None, events = None, transes = None, nxts
           = None):
8          self.dynamic = dynamic
9          self.events = events
10         self.transes = transes
11         self.nxts = nxts
12
13     def run(self, x0, t_span, t_density = 1000):
```

```
14          if not self.dynamic:
15              raise Exception("Dynamic has not been set!")
16
17          t_eval = np.linspace(t_span[0], t_span[1], np.int((t_span[1]-t_span
                [0])*t_density))
18          if self.events:
19              for event in self.events:
20                  event.terminal = True
21                  event.direction = True
22              events = self.events
23          else:
24              events = None
25          sol = solve_ivp(self.dynamic, t_span, x0, t_eval = t_eval, events =
                events)
26
27          if not sol.success:
28              raise Exception(sol.message)
29
30          if sol.status == 0:
31              return [sol.t, sol.y]
32
33          idx = 0
34          for event in self.events:
35              if sol.t_events[idx].size > 0:
36                  break
37              idx += 1
38          ts = sol.t
39          xs = sol.y
40          tf = sol.t_events[idx][0]
41          xf = sol.y_events[idx][0]
42
43          if self.transes:
44              xf = self.transes[idx](tf, xf)
45
46          if self.nxts:
47              r = self.nxts[idx].run(xf, [tf, t_span[1]], t_density)
48              return [np.hstack([ts, r[0]]), np.hstack([xs, r[1]])]
49
```

```
50          while True:
51              t_eval = np.linspace(tf, t_span[1], np.int((t_span[1]-tf)*
                    t_density))
52              sol = solve_ivp(self.dynamic, [tf, t_span[1]], xf, t_eval =
                    t_eval, events = events)
53              ts = np.hstack([ts, sol.t])
54              xs = np.hstack([xs, sol.y])
55              if sol.status == 0:
56                  break
57              idx = 0
58              for event in self.events:
59                  if sol.t_events[idx].size > 0:
60                      break
61                  idx += 1
62              tf_new = sol.t_events[idx][0]
63              xf = sol.y_events[idx][0]
64              if self.transes:
65                  xf = self.transes[idx](tf, xf)
66              if tf_new-tf < 1e-6:
67                  break
68              tf = tf_new
69          return [ts, xs]
```

ConditionalDynamic 类

```
1  import numpy as np
2  from scipy.integrate import solve_ivp
3
4
5  class ConditionalDynamic:
6
7      def __init__(self, dynamic = None, event = None, trans = None, nxt =
           None):
8          self.dynamic = dynamic
9          self.event = event
```

```
10          self.trans = trans
11          self.nxt = nxt
12
13      def run(self, x0, t_span, t_density = 1000):
14          if not self.dynamic:
15              raise Exception("Dynamic has not been set!")
16
17          t_eval = np.linspace(t_span[0], t_span[1], np.int((t_span[1]-t_span
                [0])*t_density))
18          if self.event:
19              self.event.terminal = True
20              self.event.direction = True
21              events = [self.event]
22          else:
23              events = None
24          sol = solve_ivp(self.dynamic, t_span, x0, t_eval = t_eval, events =
                events)
25
26          if not sol.success:
27              raise Exception(sol.message)
28
29          if sol.status == 0:
30              return [sol.t, sol.y]
31
32          ts = sol.t
33          xs = sol.y
34          tf = sol.t_events[0][0]
35          xf = sol.y_events[0][0]
36          if self.trans:
37              xf = self.trans(tf, xf)
38
39          if self.nxt:
40              r = self.nxt.run(xf, [tf, t_span[1]], t_density)
41              return [np.hstack([ts, r[0]]), np.hstack([xs, r[1]])]
42
43          while True:
44              t_eval = np.linspace(tf, t_span[1], np.int((t_span[1]-tf)*
                    t_density))
```

```
45              sol = solve_ivp(self.dynamic, [tf, t_span[1]], xf, t_eval =
                    t_eval, events = events)
46              ts = np.hstack([ts, sol.t])
47              xs = np.hstack([xs, sol.y])
48              if sol.status == 0:
49                  break
50              tf_new = sol.t_events[0][0]
51              xf = sol.y_events[0][0]
52              if self.trans:
53                  xf = self.trans(tf, xf)
54              if tf_new-tf < 1e-6:
55                  break
56              tf = tf_new
57          return [ts, xs]
```