Simulation Engineering Exercises 04

Name: Chenfeng Zhu
Matrikelnummer: 450485
University: TU-Clausthal
Program: ITIS
Language: JAVA

1. Improvement I of Circus Trapeze - Multi Step Solvers

Assumptions:
The mass of the string is ignored.
Friction within the string is ignored.
The air resistance of the activity is ignored.
In this simple simulation, the length of the string does not change.
There are 3 types of methods (Eurler, RK2 and RK4) to calculate the movements.

Mathematical Model:
The equations for the model:
$f(x) = C_0 + f'(x) \cdot dx$
$f'(x) = C_1 + f''(x) \cdot dx$
$f''(x) = C_2 \cdot \sin\big(f(x)\big), \qquad C_2 = -\frac{g}{L}$
Solutions of Ordinary Differential Equations:

a) Euler:
$k_1 = \mathrm{h} \cdot \mathrm{f}(x_n y_n)$
$y_{n+1} = y_n + \mathrm{h} \cdot \mathrm{f}(x_n y_n)$

b) Runge-Kutta 2 (Heun's Method):
$y_{n+1} = y_n + (a_1 k_1 + a_2 k_2) \cdot \mathrm{h}$
$k_1 = \mathrm{h} \cdot \mathrm{f}(x_n y_n), \quad k_2 = \mathrm{h} \cdot \mathrm{f}(x_n + p_1 h, y_n + q_1 k_1 h)$
$a_1 + a_2 = 1, \quad a_2 p_1 = 1/2, \quad a_2 q_1 = 1/2$
Set $a_1 = 1/2,\ a_2 = 1/2$
Then, we get: $y_{n+1} = y_n + (k_1 + k_2)/2$

c) Runge-Kutta 4:
$k_1 = \mathrm{h} \cdot \mathrm{f}(x_n y_n)$
$k_2 = \mathrm{h} \cdot \mathrm{f}(x_n + h/2, y_n + k_1/2)$
$k_3 = \mathrm{h} \cdot \mathrm{f}(x_n + h/2, y_n + k_2/2)$
$k_4 = \mathrm{h} \cdot \mathrm{f}(x_n + h, y_n + k_3)$
$y_{n+1} = y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6}$

Main code:

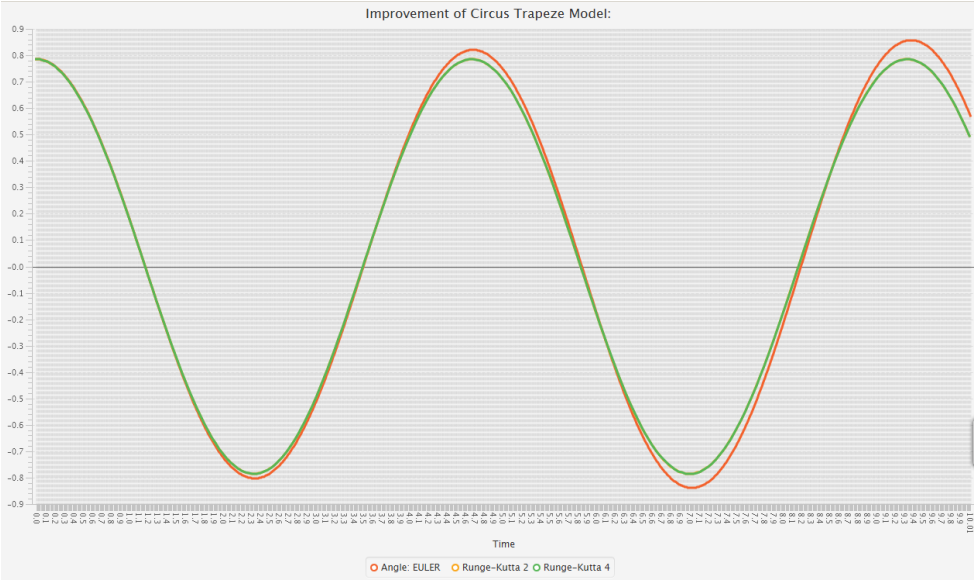
```
public void start() {
    theta.setValues(Math.toRadians(45));
    velocity.setValues(0);
    sl.init(step_time);
    for (;;) {
        theta.setDer(velocity.getValues());
        theta.setTime(sl.getTime());
        // System.out.println(theta.getTime() + ": " + theta.getValues());
        velocity.setDer((-g / l * Math.sin(theta.getValues())));
        this.saveData(theta);
        if (sl.getTime() > this.end_time) {
            break;
        }
        sl.integrate();
    }
}
```

```
switch (this.getMethod().getMethod()) {
case MethodType.EULER:
    this.values += sl.getStepTime() * der;
    // sl.getList().add(this.clone());
    break;
case MethodType.RK2:
    if ((sl.getFrameCount() & 1) == 0) {
        this.prev_state = this.clone();
        this.values += 2 * sl.getStepTime() * der;
    } else {
        // System.out.println(der +","+ this.prev_state.getDer());
        this.values = this.prev_state.getValues() + sl.getStepTime()
                * (this.prev_state.getDer() + der);
        // sl.getList().add(this.clone());
    }
    break;
case MethodType.RK4:
    switch (sl.getFrameCount() & 3) {
    case 0:
        this.prev_state = this.clone();
        this.values += 2 * sl.getStepTime() * der;
        this.temp_prev_der[0] = der;
        break;
    case 1:
        this.values = this.prev_state.getValues() + 2 * sl.getStepTime() * der;
        this.temp_prev_der[1] = der;
        break;
    case 2:
        this.values = this.prev_state.getValues() + 4 * sl.getStepTime() * der;
        this.temp_prev_der[2] = der;
        break;
    case 3:
        // System.out.println(temp_prev_der[0] + "," + temp_prev_der[1]
        // + "," + temp_prev_der[2] + "," + der);
        this.values = this.prev_state.getValues()
                + (4 * sl.getStepTime() / 6)
                * (temp_prev_der[0] + 2 * temp_prev_der[1] + 2 * temp_prev_der[2] + der);
        // sl.getList().add(this.clone());
        break;
    }
    break;
```

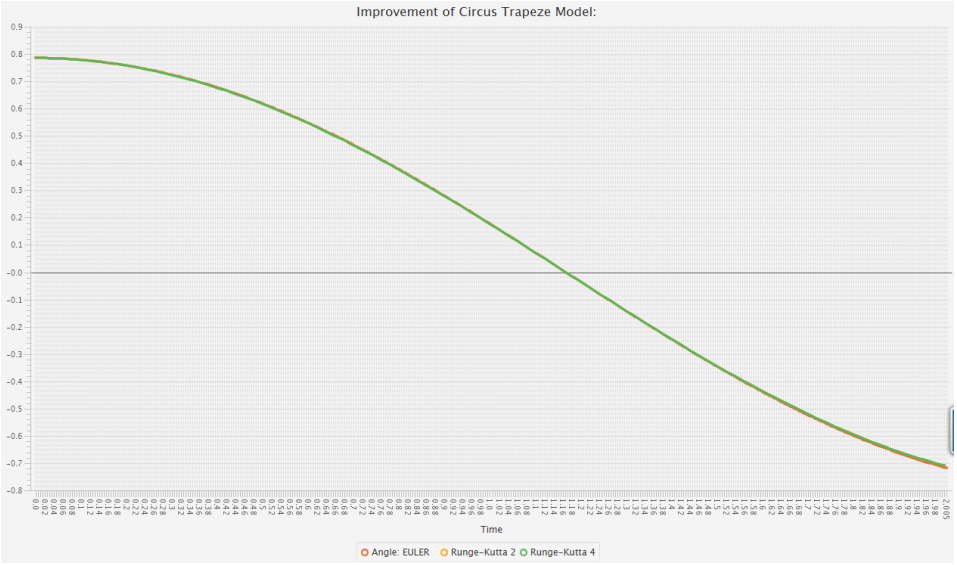
```
public void saveData(AcrobatStateImpr state) {
    switch (state.getMethod().getMethod()) {
    case MethodType.EULER:
        state_list.add(state.clone());
        break;
    case MethodType.RK2:
        if ((sl.getFrameCount() & 1) == 0) {
            state_list.add(state.clone());
        }
        break;
    case MethodType.RK4:
        if ((sl.getFrameCount() & 3) == 0) {
            state_list.add(state.clone());
        }
        break;
    }
}
```

Graph:

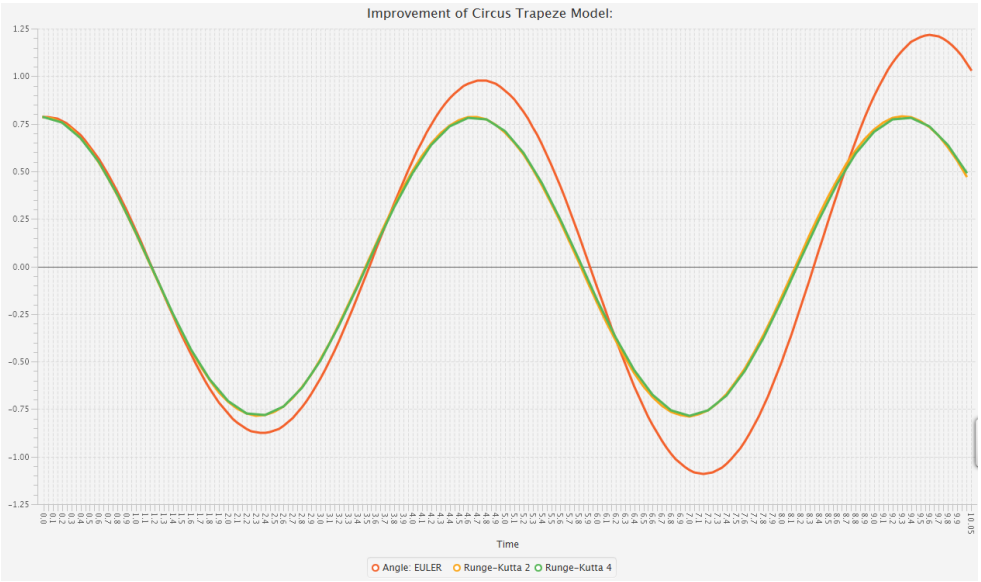
a) Start_time: 0; end_time: 10; step_time: 0.01.



b) Start_time: 0; end_time: 2; step_time: 0.005.



c) Start_time: 0; end_time: 10; step_time: 0.05.



Output results:

Test the simulation 3 times. Show several data at the top and data in the end for the first simulation.

a) Euler:

Time: 0.0, Theta: 0.7853981633974483,0.0.
Time: 0.01, Theta: 0.7853981633974483,-0.7940789908505675.
Time: 0.02, Theta: 0.7852595704683357,-1.588157981701135.
Time: 0.03, Theta: 0.7849823846101106,-2.3821269111924317.
Time: 0.04, Theta: 0.784566625032104,-3.1758756722156347.
Time: 0.05, Theta: 0.7840123301609624,-3.9692940814609456.
Time: 0.06, Theta: 0.7833195576539482,-4.7622718490782106.
Time: 0.07, Theta: 0.7824883844175337,-5.554698548517229.
Time: 0.08, Theta: 0.7815189066312599,-6.346463586615315.
Time: 0.09, Theta: 0.7804112397768163,-7.137456173999608.
.....
Time: 9.9, Theta: 0.6587740304093284,-42.322948109862885.
Time: 9.91, Theta: 0.6513872824726502,-43.010388518737095.
Time: 9.92, Theta: 0.6438805535505109,-43.691250781401145.
Time: 9.93, Theta: 0.636254991746011,-44.36538999230439.
Time: 9.94, Theta: 0.6285117704530404,-45.032660831345.
Time: 9.95, Theta: 0.6206520884286318,-45.69291762057537.
Time: 9.96, Theta: 0.6126771698554161,-46.346014383704585.
Time: 9.97, Theta: 0.6045882643936932,-46.99180490836688.
Time: 9.98, Theta: 0.5963866472226348,-47.63014281111947.
Time: 9.99, Theta: 0.5880736190701433,-48.26088160512723.
Time: 10.0, Theta: 0.579650506230897,-48.88387477048606.

b) RK2:

Time: 0.0, Theta: 0.7853981633974483,0.0.

Time: 0.02, Theta: 0.7851209775392232,-1.588157981701135.
Time: 0.04, Theta: 0.7842894968071952,-3.1754352285615113.
Time: 0.06, Theta: 0.7829041824275134,-4.760949848769941.
Time: 0.08, Theta: 0.7809658037191993,-6.343818011361693.
Time: 0.1, Theta: 0.7784754389382275,-7.923152998297766.
Time: 0.12, Theta: 0.7754344764487218,-9.498064282441831.
Time: 0.14, Theta: 0.7718446162107322,-11.06765663998349.
Time: 0.16, Theta: 0.7677078715710802,-12.631029305781485.
Time: 0.18, Theta: 0.7630265713408209,-14.187275180003393.
Time: 0.2, Theta: 0.7578033621399634,-15.735480094317213.
……
Time: 9.8, Theta: 0.6386307375323024,-35.11578312616234.
Time: 9.82, Theta: 0.6261393469648627,-36.44351043290668.
Time: 9.84, Theta: 0.6131884418264315,-37.748081389455294.
Time: 9.86, Theta: 0.5997862883413393,-39.02844452197956.
Time: 9.88, Theta: 0.5859415204624793,-40.28354504457305.
Time: 9.9, Theta: 0.5716631407893894,-41.512326294031155.
Time: 9.92, Theta: 0.5569605209608932,-42.71373130416416.
Time: 9.94, Theta: 0.5418434014741398,-43.88670451596967.
Time: 9.96, Theta: 0.5263218908833562,-45.030193618874016.
Time: 9.98, Theta: 0.5104064643334958,-46.14315151711452.
Time: 10.0, Theta: 0.49410796138623453,-47.2245384141813.

c) RK4:

Time: 0.0, Theta: 0.7853981633974483,0.0.
Time: 0.04, Theta: 0.7842896249066619,-3.175141568978802.
Time: 0.08, Theta: 0.7809664700228086,-6.343231918390409.
Time: 0.12, Theta: 0.775436090864058,-9.497188961043651.
Time: 0.16, Theta: 0.7677108432655286,-12.629869977330701.
Time: 0.2, Theta: 0.7578080988899295,-15.734044050927618.
……
Time: 9.8, Theta: 0.6392372713921253,-35.0461623675212.
Time: 9.84, Theta: 0.6138454989730481,-37.68051159055973.
Time: 9.88, Theta: 0.5866478332157916,-40.21826275585609.
Time: 9.92, Theta: 0.5577146562982611,-42.65097683232198.
Time: 9.96, Theta: 0.5271222467528236,-44.970209746418725.
Time: 10.0, Theta: 0.4949527637546163,-47.167568869627274.

Conclusion:

From the graph, we can see that the step size is less, the error is less.

From the results, we can see that the one with RK2 and the one with RK4 are close, with only a difference of 0.0009 after 10s. The one with Euler is a little away from them with a difference of over 0.08 after 10s.

RK4 is the most accurate.

2. Improvement II of Circus Trapeze - Adaptive Step Size

Assumptions:

It is the same as the first one.

In the condition that there is an acceptable error threshold, the step size would change dynamically until the error between the results with different step sizes is less than the error threshold.

Solution:

Set an error threshold. If the threshold <= 0, the step size won't be adaptive.

Compare the state with one step size and the one with half a step size. If the difference is larger than the threshold, then compare the one with half and the one with a quarter. If the difference is less than the threshold, then take the step size.

Main code:

```
/**
 * Adapt the step size.
 *
 * @param list
 */
public void adaptStepSize(StateList list) {
    if (this.e_threshold_ass <= 0) {
        return;
    }
    switch (list.getState().getMethod().getMethod()) {
    case MethodType.EULER:
        testStepSize(list);
        break;
    case MethodType.RK2:
        if ((sl.getFrameCount() & 1) == 0) {
            testStepSize(list);
        }
        break;
    case MethodType.RK4:
        if ((sl.getFrameCount() & 3) == 0) {
            testStepSize(list);
        }
        break;
    }
}
```

```
switch (list.stepCount) {
case 0: // save the current state.
    temp_theta[0] = theta.clone();
    temp_velocity = velocity.clone();
    temp_framecount[0] = list.getFrameCount();
    break;
case 1: // save the state after one steps for halve
    if (flag_ass_direction == -1) {
        temp_theta[1] = theta.clone();
        temp_framecount[1] = list.getFrameCount();
        list.halveStepTime();
        list.setFrameCount(temp_framecount[0]);
        list.setState(temp_velocity);
    }
    break;
case 2: // save the state after two steps for double
    if (flag_ass_direction == 1) {
        temp_theta[1] = theta.clone();
        temp_framecount[1] = list.getFrameCount();
        list.doubleStepTime();
        list.setFrameCount(temp_framecount[0]);
        list.setState(temp_velocity);
    }
    break;
case 3: // save the state after increase or decrease the step size.
    temp_theta[2] = theta.clone();
    temp_framecount[2] = list.getFrameCount();
    double error = Math.abs(temp_theta[2].getValues() - temp_theta[1].getValues());
    if (error > e_threshold_ass) {
        // error can not be accepted.
        list.setFrameCount(temp_framecount[0]);
        list.setState(temp_velocity);
    } else {
        // if the error is accepted, then continue.
        if (error / e_threshold_ass < 0.001) {
            // if the error is too little,
            // then try use a larger step on next time.
            list.doubleStepTime();
        }
        flag = true;
    }
```

Graph:

NULL.

Output results:

Test the simulation 3 times. Step size is 0.01.

a) EULER:

b) RK2:

c) RK4:

3. Improvement of Ice Cream Shop Simulation

Assumptions:

a) Shops open at 11 AM
b) A new customer arrives randomly (based on Normal Distribution) between every
   1 to 3 minutes till 1 PM
   2 to 10 minutes from 1 PM to 5PM
   1 to 2 minutes from 5 PM to 7PM
   2 to 5 minutes from 7 PM to 8PM
c) Customers ask randomly (based on Normal Distribution) for
   1 to 5 scoops of ice till noon
   3 to 6 scoops of ice from noon to 3 PM
   1 to 4 scoops of ice from 3 PM to 8 PM
d) Each ice takes 30 seconds

Normal Distribution: 95% of values are within 2 standard deviations of the mean.

Mathematical Model:

Normal Distribution:

$$\mathrm{P(x)} = \left(\frac{1}{\sqrt{2\pi}\sigma}\right) e^{-(x-\mu)^2/2\sigma^2} \qquad -\infty \leq x \leq +\infty$$

Solution:

a) A new customer arrives randomly (based on Normal Distribution) between every
   1 to 3 minutes till 1 PM: $\mu = 2.0;\ \sigma^2 = 0.5$
   2 to 10 minutes from 1 PM to 5PM: $\mu = 6.0;\ \sigma^2 = 2.0$
   1 to 2 minutes from 5 PM to 7PM: $\mu = 1.5;\ \sigma^2 = 0.25$
   2 to 5 minutes from 7 PM to 8PM: $\mu = 3.5;\ \sigma^2 = 0.75$
b) Customers ask randomly (based on Normal Distribution) for
   1 to 5 scoops of ice till noon: $\mu = 3.0;\ \sigma^2 = 1.0$
   3 to 6 scoops of ice from noon to 3 PM: $\mu = 4.5;\ \sigma^2 = 0.75$
   1 to 4 scoops of ice from 3 PM to 8 PM: $\mu = 2.5;\ \sigma^2 = 0.75$

Random Test:

Using *Random.nextGaussian()* to create a random number. The below is a test:

```
{[-1,1]=676, [-2,-1)(1,2]=269, [-3,-2)(2,3]=52, other=3}
{[1,3]=954, other=46}
{[1,5]=990, other=10}
```

In the 1st one, the result of standard distribution is 67.6% (nearly 68%), 94.5% (nearly 95%) and 99.7% (nearly 99%). In the 2nd one, 95.4% of numbers are between 1 and 3 (nearly 95%). In the 3rd one, the number of scoops must be integer. Considering that those between 0.5 and 1.0 would become 1, and those between 5.0 and 5.5 would become 5, the proportion of numbers which between 1 and 5 would increase (over 95%).

That means, the Generator of random is acceptable.

Main code:

```java
/**
 * Generate a number randomly based on Normal Distribution.
 *
 * @param time
 * @return a random number
 */
public double generateND(double time) {
    // it is a real random in JAVA (Math.Random() is pseudo random)
    Random r = new Random();

    // next new customers arrives randomly
    double next_event_time = time;
    switch (Event.getHour(time)) {
    case Event.TIME_11:
    case Event.TIME_12:
        next_event_time += r.nextGaussian() * 0.5 + 2.0; // 1-3 at 11-13
        break;
    case Event.TIME_13:
    case Event.TIME_14:
    case Event.TIME_15:
    case Event.TIME_16:
        next_event_time += r.nextGaussian() * 2.0 + 6.0; // 2-10 at 13-17
        break;
    case Event.TIME_17:
    case Event.TIME_18:
        next_event_time += r.nextGaussian() * 0.25 + 1.5; // 1-2 at 17-19
        break;
    case Event.TIME_19:
        next_event_time += r.nextGaussian() * 0.75 + 3.5; // 2-5 at 19-20
        break;
    }
    return next_event_time;
}

 // next new customers arrives randomly
 // according to the random type.
 double next_event_time = 0;
 switch (sim.getType_random()) {
 case Simulator.RANDOM_RANDOM:
     next_event_time = this.generateRandom(this.time);
     break;
 case Simulator.RANDOM_NORMAL_DISTRIBUTION_1:
 case Simulator.RANDOM_NORMAL_DISTRIBUTION_2:
     next_event_time = this.generateND(this.time);
     break;
 default:
     next_event_time = this.generateRandom(this.time);
     break;
 }
```

```java
/**
 * Generate a number randomly based on Normal Distribution.
 *
 * @param time
 * @return a random number
 */
public int generateND(double time) {
    Random r = new Random();
    int num_scoops = 0;
    // numbers of scoops which customers want are random
    switch (Event.getHour(time)) {
    case Event.TIME_OFF:
        break;
    case Event.TIME_11:
        // 1-5 at 11-12
        num_scoops = (int) Math.round(r.nextGaussian() * 1.0 + 3);
        break;
    case Event.TIME_12:
    case Event.TIME_13:
    case Event.TIME_14:
        // 3-6 at 12-15
        num_scoops = (int) Math.round(r.nextGaussian() * 0.75 + 4.5);
        break;
    case Event.TIME_15:
    case Event.TIME_16:
    case Event.TIME_17:
    case Event.TIME_18:
    case Event.TIME_19:
        // 1-4 at 15-20
        num_scoops = (int) Math.round(r.nextGaussian() * 0.75 + 2.5);
        break;
    }
    return num_scoops;
}
```

```java
int num_scoops = 0;
// numbers of scoops which customers want are random
switch (sim.getType_random()) {
case Simulator.RANDOM_RANDOM:
    num_scoops = this.generateRandom(c.time_generated);
    break;
case Simulator.RANDOM_NORMAL_DISTRIBUTION_1:
case Simulator.RANDOM_NORMAL_DISTRIBUTION_2:
    num_scoops = this.generateND(c.time_generated);
    break;
default:
    num_scoops = this.generateRandom(c.time_generated);
    break;
}
c.setNum_scoops(num_scoops);
```

Graph:

NULL.

Output results:

Test the simulation 3 times.

a) IceCreamShopSimulation_ND01:

Time: 0.0-1440.0 (1 day)
Customer Quantity: 197
Total scoops made: 619
Average scoops per customer: 3.1421319796954315
The customer waiting longest: Customer_61 with 5 scoops (arrived at 779.0465931748322(Day_0 12h:59.0m) with 5 customers ahead, was serviced at 789.189567358623(Day_0 13h:9.2m), left at 791.689567358623(Day_0 13h:11.7m))
Total waiting time: 530.1300219969971
Average waiting time per customer: 2.6910153400862797
Customers at different periods: {1=31, 2=30, 3=10, 4=10, 5=11, 6=11, 7=38, 8=39, 9=17}
Total scoops at different periods: {1=85.0, 2=140.0, 3=46.0, 4=47.0, 5=28.0, 6=26.0, 7=101.0, 8=102.0, 9=44.0}
Total waiting time at different periods: {1=47.88457775936456, 2=237.6069084597442, 3=41.69107181632171, 4=23.5, 5=14.0, 6=13.0, 7=65.52421062838039, 8=64.92325333318627, 9=22.0}
Average scoops at different periods: {1=2.741, 2=4.666, 3=4.6, 4=4.7, 5=2.545, 6=2.363, 7=2.657, 8=2.615, 9=2.588}
Average waiting time at different periods: {1=1.544, 2=7.92, 3=4.169, 4=2.35, 5=1.272, 6=1.181, 7=1.724, 8=1.664, 9=1.294}

b) IceCreamShopSimulation_ND02:

Time: 0.0-14400.0 (10 days)
Customer Quantity: 1949
Total scoops made: 6053
Average scoops per customer: 3.1056952283222166
The customer waiting longest: Customer_61 with 5 scoops (arrived at 778.4064451216758(Day_0 12h:58.4m) with 6 customers ahead, was serviced at 788.679126583279(Day_0 13h:8.7m), left at 791.179126583279(Day_0 13h:11.2m))
Total waiting time: 4843.161497493524
Average waiting time per customer: 2.4849468945579907
Customers at different periods: {1=298, 2=303, 3=104, 4=101, 5=96, 6=99, 7=377, 8=403, 9=168}
Total scoops at different periods: {1=887.0, 2=1363.0, 3=471.0, 4=453.0, 5=244.0, 6=231.0, 7=939.0, 8=1039.0, 9=426.0}
Total waiting time at different periods: {1=494.22860867441204, 2=2102.9748919647004, 3=406.4129962045149, 4=227.87830496335755, 5=122.38513420001254, 6=115.88414806879518, 7=533.8766989641254, 8=624.3741025141676, 9=215.1466119394388}
Average scoops at different periods: {1=2.976, 2=4.498, 3=4.528, 4=4.485, 5=2.541, 6=2.333, 7=2.49, 8=2.578, 9=2.535}
Average waiting time at different periods: {1=1.658, 2=6.94, 3=3.907, 4=2.256, 5=1.274, 6=1.17, 7=1.416, 8=1.549, 9=1.28}

c) IceCreamShopSimulation_ND03:

Time: 0.0-14400.0 (10 days)

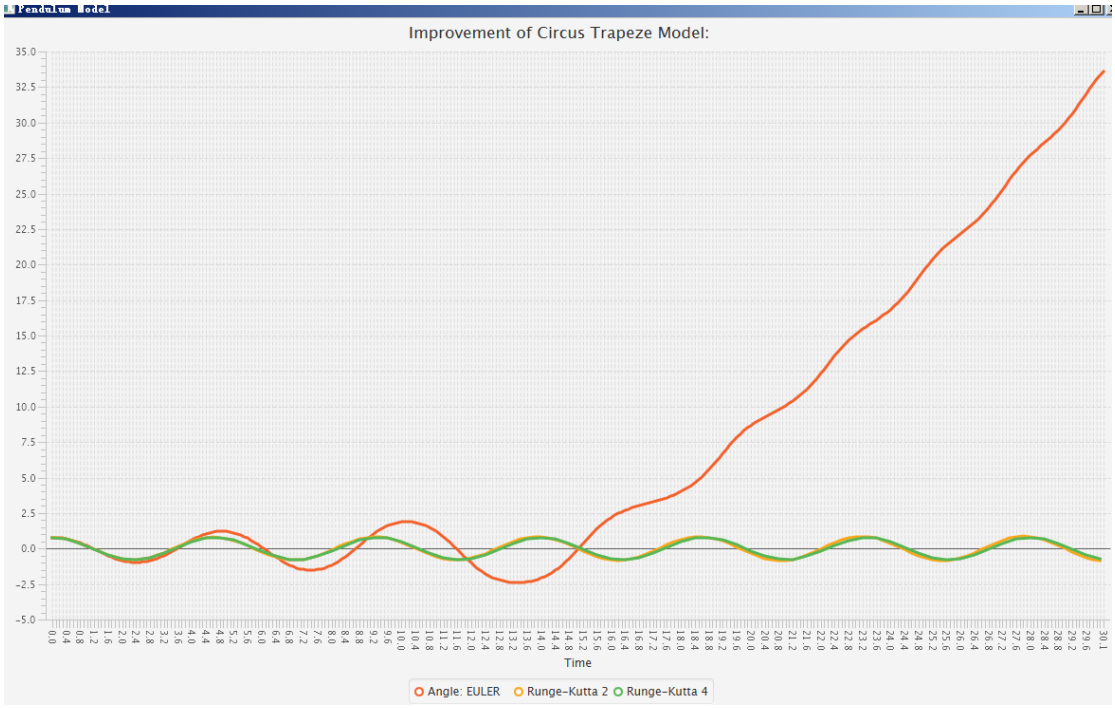
Customer Quantity: 1960
Total scoops made: 6099
Average scoops per customer: 3.111734693877551
The customer waiting longest: Customer_1625 with 5 scoops (arrived at 12296.458121438056(Day_8 12h:56.5m) with 7 customers ahead, was serviced at 12309.993441207469(Day_8 13h:10.0m), left at 12312.493441207469(Day_8 13h:12.5m))
Total waiting time: 4667.167869777639
Average waiting time per customer: 2.381208096825326
Customers at different periods: {1=302, 2=296, 3=103, 4=102, 5=102, 6=106, 7=370, 8=401, 9=178}
Total scoops at different periods: {1=944.0, 2=1345.0, 3=483.0, 4=451.0, 5=256.0, 6=271.0, 7=922.0, 8=978.0, 9=449.0}
Total waiting time at different periods: {1=532.7459748479782, 2=1929.6036063799245, 3=395.4544598442161, 4=226.79680545224778, 5=128.6931324345096, 6=135.5, 7=526.5746540788373, 8=566.3782323750418, 9=225.42100436488295}
Average scoops at different periods: {1=3.125, 2=4.543, 3=4.689, 4=4.421, 5=2.509, 6=2.556, 7=2.491, 8=2.438, 9=2.522}
Average waiting time at different periods: {1=1.764, 2=6.518, 3=3.839, 4=2.223, 5=1.261, 6=1.278, 7=1.423, 8=1.412, 9=1.266}

Conclusion:
One day is a short time and around 200 customers in 1 day which are chosen to be samples are not enough. When we simulated for a longer time, the result is more satisfied. When the random is based on Normal Distribution, the proportion becomes steady.

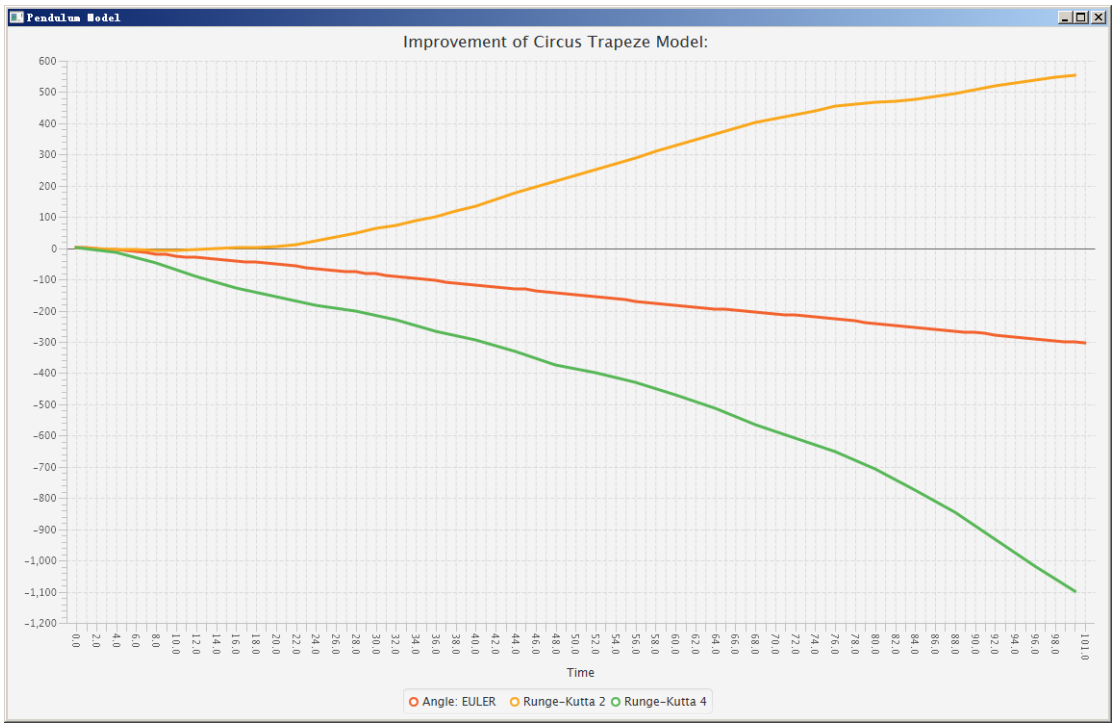
4. Something interesting about ODEs

When the step size is strange, the results would run out of control.

a) Start: 0; end: 30; step: 0.1



b) Start: 0; end: 100; step: 1



Code:

https://github.com/sampig/SimulationEngineering