

第三章

3-10

采用 Python 语言进行计算程序编写，计算思路如下：

(1) 导入参数临界温度 T_c 、临界压力 p_c 、偏心因子 ω 、摩尔质量 M 和理想气体常数 $R = 8.314462618 \text{ J}/(\text{mol} \cdot \text{K})$;

(2) 给定温度 T 计算参数 a 和 b ;

(3) 给定温度 T 和压力 p 计算参数 A 和 B ;

(4) 计算压缩因子 Z 的多项式系数 C_2, C_1, C_0 ;

(5) 使用牛顿法分别计算压缩因子 Z 的液相值 Z_l 和气相值 Z_g ;

(6) 计算比体积 v 的液相值 v_l 和气相值 v_g ;

(7) 在给定压力 p 下，绘制比体积 v 与温度 T 的关系曲线，并标注饱和温度 T_{sat} 。

计算程序如下：

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import os
4
5 # 使用 Times New Roman 作为 matplotlib 全局字体
6 plt.rcParams["font.family"] = "serif"
7 plt.rcParams["font.serif"] = ["Times New Roman"]
8 plt.rcParams["mathtext.fontset"] = "stix"
9
10
11 class PRFluid:
12     def __init__(self, Tc, Pc, omega, M):
13         self.Tc = Tc # 输入K
14         self.Pc = Pc * 1e6 # 输入MPa
15         self.omega = omega # 无量纲
16         self.M = M / 1000 # 输入g/mol
17
18     R = 8.314462618 # J/(mol*K)
19
20     # 计算a和b
21     def params(self, T):
22         kappa = 0.37464 + 1.54226 * self.omega - 0.26992 * self.omega**2
23         Tr = T / self.Tc
24         alpha = (1 + kappa * (1 - Tr**0.5)) ** 2
25         a = 0.45724 * self.R**2 * self.Tc**2 / self.Pc * alpha
```

```

26     b = 0.07780 * self.R * self.Tc / self.Pc
27     return a, b
28
29     # 计算A和B
30     def AB(self, T, p):
31         a, b = self.params(T)
32         A = a * p * 1e6 / (self.R * T) ** 2
33         B = b * p * 1e6 / (self.R * T)
34         return A, B
35
36     # 计算C2, C1, C0
37     def C(self, T, p):
38         A, B = self.AB(T, p)
39         C2 = -(1 - B)
40         C1 = A - 3 * B**2 - 2 * B
41         C0 = -(A * B - B**2 - B**3)
42         return C2, C1, C0
43
44     # 计算压缩因子Z
45     # 液相
46     def Zl(self, T, p):
47         C2, C1, C0 = self.C(T, p)
48         # 牛顿法求解Z
49         Zl = 0.001 # 初始猜测值
50         for _ in range(100):
51             f = Zl**3 + C2 * Zl**2 + C1 * Zl + C0
52             df = 3 * Zl**2 + 2 * C2 * Zl + C1
53             Zl_new = Zl - f / df
54             if abs(Zl_new - Zl) < 1e-6:
55                 break
56             Zl = Zl_new
57         return Zl
58
59     # 气相
60     def Zg(self, T, p):
61         C2, C1, C0 = self.C(T, p)
62         # 牛顿法求解Z
63         Zg = 1.0 # 初始猜测值
64         for _ in range(100):
65             f = Zg**3 + C2 * Zg**2 + C1 * Zg + C0

```

```

66         df = 3 * Zg**2 + 2 * C2 * Zg + C1
67         Zg_new = Zg - f / df
68         if abs(Zg_new - Zg) < 1e-6:
69             break
70         Zg = Zg_new
71     return Zg
72
73     # 计算比体积v
74     # 液相
75     def vl(self, T, p):
76         Zl = self.Zl(T, p)
77         vl = Zl * self.R * T / (p * 1e6 * self.M)
78         return vl
79
80     # 气相
81     def vg(self, T, p):
82         Zg = self.Zg(T, p)
83         vg = Zg * self.R * T / (p * 1e6 * self.M)
84         return vg
85
86     # 画v-T图
87     def plot_Tv(
88         self,
89         fluid_name, # 流体名称
90         p, # 压力 Pa
91         Tsat, # 饱和温度 K
92         T_min, # 温度范围最小值 K
93         T_max, # 温度范围最大值 K
94         nT=220, # 温度点数
95     ):
96         T_grid = np.linspace(T_min, T_max, nT) # 温度网格
97         v_grid = np.empty_like(T_grid) # 比体积网格
98         # 计算比体积
99         for i, T in enumerate(T_grid):
100             if T < Tsat:
101                 v_grid[i] = self.vl(T, p)
102             elif T > Tsat:
103                 v_grid[i] = self.vg(T, p)
104             else:
105                 v_grid[i] = 0.5 * (self.vl(T, p) + self.vg(T, p))

```

```

106 fig, ax = plt.subplots() # 创建图像和坐标轴
107 # 主曲线
108 ax.plot(v_grid, T_grid, linewidth=2, label=fluid_name)
109 xmin, xmax = np.nanmin(v_grid), np.nanmax(v_grid)
110 # Tsat 虚线
111 ax.hlines(Tsat, xmin, xmax, linestyle="--", label=r"$T_{\mathrm{sat}}$"
112         )
113 # 标注 Tsat
114 yt = list(ax.get_yticks())
115 # 加入Tsat并排序
116 if not any(abs(t - Tsat) < 1e-8 for t in yt):
117     yt.append(Tsat)
118 yt = np.array(sorted(yt))
119 # 生成刻度标签: 对 Tsat 使用仅数值标签 (两位小数), 其它刻度保留数字格式
120 # (根据范围选择小数位)
121 deltaT = T_grid.max() - T_grid.min()
122 labels = []
123 for t in yt:
124     if abs(t - Tsat) < 1e-8 or abs(t - Tsat) < 1e-6 * max(1.0, deltaT):
125         labels.append(f"{Tsat:.2f}")
126     else:
127         # 根据温度范围决定格式, 避免过多小数
128         if deltaT > 50:
129             labels.append(f"{t:.0f}")
130         else:
131             labels.append(f"{t:.2f}")
132 ax.set_yticks(yt)
133 ax.set_yticklabels(labels)
134 # 轴标签
135 ax.set_xlabel(r"$v$ (m3/kg)")
136 ax.set_ylabel(r"$T$ (K)")
137 # 标题
138 ax.set_title(f"{fluid_name} $v$-$T$ at $p$ = {p:.1f} MPa")
139 ax.grid(True)
140 ax.set_xscale("log") # 使用对数刻度
141 ax.legend(loc="upper left", frameon=True, fancybox=True, framealpha=0.9)
142
143 # 固定保存路径为脚本同目录下的 figs 文件夹
144 base_dir = os.path.dirname(os.path.abspath(__file__))
145 fig_dir = os.path.join(base_dir, "figs")

```

```

144     os.makedirs(fig_dir, exist_ok=True)
145
146     # 文件名固定为"流体名称.png"
147     filename = f"{fluid_name}.png"
148     savepath = os.path.join(fig_dir, filename)
149
150     # 保存图像，固定参数
151     fig.savefig(savepath, dpi=300, bbox_inches="tight", transparent=False)
152     plt.close(fig)
153
154
155 # 实际计算
156 R290 = PRFluid(369.89, 4.2512, 0.1521, 44.096)
157 R290.plot_Tv("R290", 1.4, 317.86, 200, 450)
158
159 R600a = PRFluid(407.81, 3.629, 0.184, 58.122)
160 R600a.plot_Tv("R600a", 0.6, 314.12, 200, 450)

```

R290 在 1.4MPa 下的 T - v 图和 R600a 在 0.6MPa 下的 T - v 图如下：

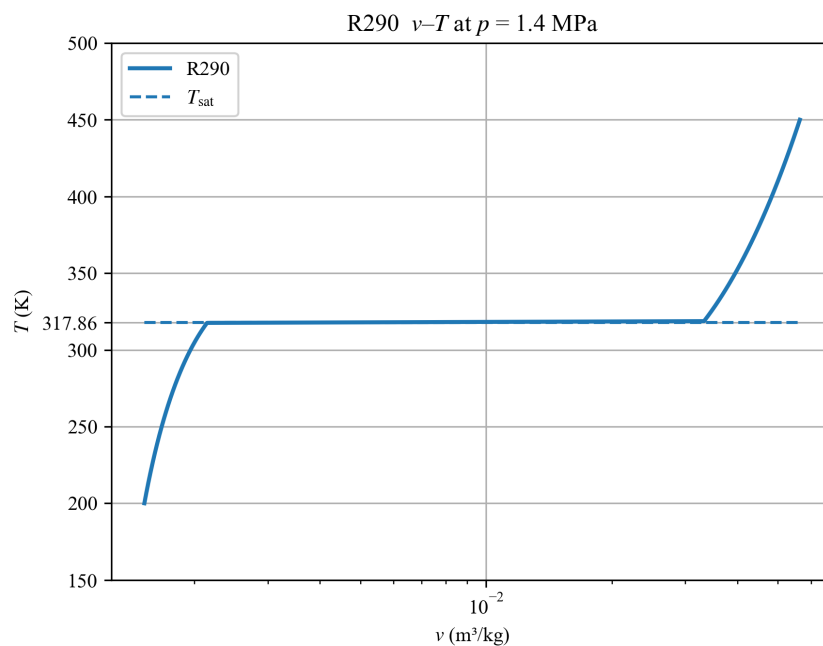


图 1: R290 在 1.4MPa 下的 T - v 图

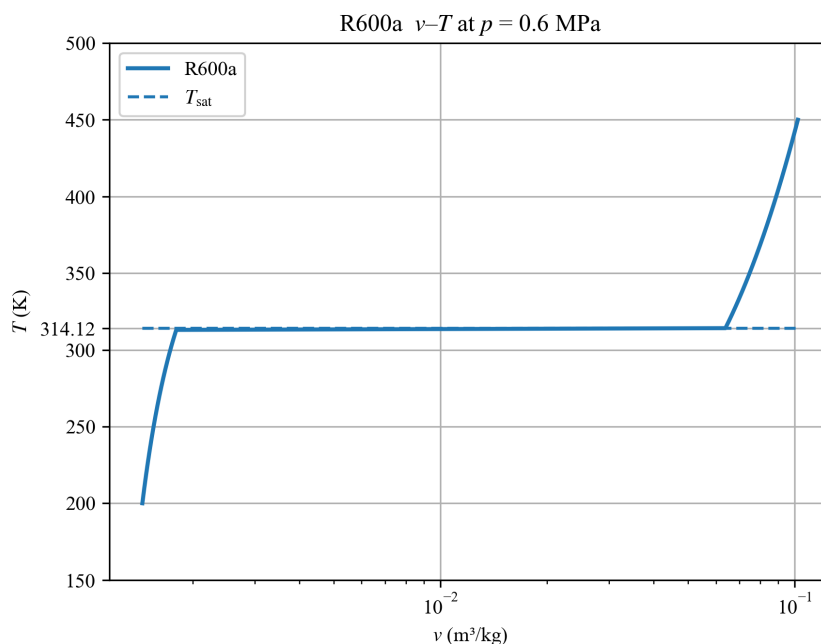


图 2: R600a 在 0.6MPa 下的 T - v 图

3-13

查物性库得，对于 R134a，各参数为： $T_c = 374.21\text{K}$ ， $p_c = 4.0593\text{MPa}$ ， $\omega = 0.326$ ， $M = 102.03\text{g/mol}$ 。

对于 R1234yf，各参数为： $T_c = 367.85\text{K}$ ， $p_c = 3.3822\text{MPa}$ ， $\omega = 0.276$ ， $M = 114.04\text{g/mol}$ ；

对于 R1234ze(E)，各参数为： $T_c = 382.75\text{K}$ ， $p_c = 3.6349\text{MPa}$ ， $\omega = 0.313$ ， $M = 114.04\text{g/mol}$ ；

压力为 0.1MPa ，温度为 $35^\circ\text{C}=308.15\text{K}$ 时，以上三种制冷剂均为气相。利用题 3-10 程序进行计算 v_g ，计算结果为：

$$v_{\text{R134a}} = 0.24679\text{m}^3/\text{kg}, \quad v_{\text{R1234yf}} = 0.22031\text{m}^3/\text{kg}, \quad v_{\text{R1234ze(E)}} = 0.22007\text{m}^3/\text{kg}$$

可以看出，三种制冷剂的比体积相差不大，R134a 的比体积略大于另外两种。

3-15

在题 3-10 的程序基础上，更改 `__init__` 函数与 a、b 计算函数 params，更改如下：

```
1 class FRFluid2:
2     def __init__(self, Tc1, Tc2, pc1, pc2, omega1, omega2, M1, M2, x1, kij):
3         self.Tc1 = Tc1 # K
4         self.Tc2 = Tc2 # K
5         self.pc1 = pc1 * 1e6 # Pa, 输入MPa, 改为乘法
6         self.pc2 = pc2 * 1e6 # Pa, 输入MPa, 改为乘法
```

```

7      self.omega1 = omega1 # 无量纲
8      self.omega2 = omega2 # 无量纲
9      self.M1 = M1 / 1e3 # kg/mol, 输入g/mol
10     self.M2 = M2 / 1e3 # kg/mol, 输入g/mol
11     self.x1 = x1 # 组分1的摩尔分率
12     self.x2 = 1 - x1 # 组分2的摩尔分率
13     self.kij = kij # 组分间的二元交互作用参数
14
15     R = 8.314462618 # J/(mol*K)
16
17     # 计算a和b
18     def params(self, T):
19         kappa1 = 0.37464 + 1.54226 * self.omega1 - 0.26992 * self.omega1**2
20         kappa2 = 0.37464 + 1.54226 * self.omega2 - 0.26992 * self.omega2**2
21         Tr1 = T / self.Tc1
22         Tr2 = T / self.Tc2
23         alpha1 = (1 + kappa1 * (1 - Tr1**0.5)) ** 2
24         alpha2 = (1 + kappa2 * (1 - Tr2**0.5)) ** 2
25         a1 = 0.45724 * self.R**2 * self.Tc1**2 / self.pc1 * alpha1
26         a2 = 0.45724 * self.R**2 * self.Tc2**2 / self.pc2 * alpha2
27         b1 = 0.07780 * self.R * self.Tc1 / self.pc1
28         b2 = 0.07780 * self.R * self.Tc2 / self.pc2
29         a = (
30             self.x1**2 * a1
31             + self.x2**2 * a2
32             + 2 * self.x1 * self.x2 * (a1 * a2) ** 0.5 * (1 - self.kij)
33         )
34         b = self.x1 * b1 + self.x2 * b2
35         return a, b

```

在压力 $p = 0.1\text{MPa}$ 、 0.2MPa 、 0.3MPa ，温度 $T = 300\text{K}$ 时，不同的 k_{ij} 条件下，混合制冷剂 R290/R600a 的比体积计算结果与计算偏差如表 1 所示，表中计算偏差是相对于 $k_{ij} = 0.064$ 时的比体积计算结果而言的。可以看出， k_{ij} 取 0.1、0 和 -0.1 时，计算结果与 $k_{ij} = 0.064$ 时的比体积计算结果偏差逐渐增大，且偏差均小于 1%。

p (MPa)	k_{ij}	v (m ³ /kg)	误差 (%)
0.1	0.064	0.47838	
	0.1	0.47859	0.04390
	0	0.47802	0.07525
	-0.1	0.47744	0.19650
0.2	0.064	0.23422	
	0.1	0.23443	0.08966
	0	0.23384	0.16224
	-0.1	0.23324	0.41841
0.3	0.064	0.15273	
	0.1	0.15295	0.14405
	0	0.15233	0.26190
	-0.1	0.15171	0.66785

表 1: 不同 k_{ij} 条件下混合制冷剂 R290/R600a 的比体积计算结果与计算偏差

第四章

4-13

在题 3-10 程序的基础上扩充, 采用 Python 语言进行计算程序编写, 计算程序如下:

```

1 import numpy as np
2
3
4 class PRHS:
5     def __init__(self, Tc, pc, omega, M, ps0):
6         self.Tc = Tc # K
7         self.pc = pc * 1e6 # Pa, 输入MPa
8         self.omega = omega # 无量纲
9         self.M = M / 1e3 # kg/mol, 输入g/mol
10        self.ps0 = ps0 * 1e6 # Pa, 输入MPa
11
12        R = 8.314462618 # J/(mol*K)
13
14        # 计算a和b
15        def params(self, T):
16            kappa = 0.37464 + 1.54226 * self.omega - 0.26992 * self.omega**2
17            Tr = T / self.Tc
18            alpha = (1 + kappa * (1 - Tr**0.5)) ** 2

```



```

19     a = 0.45724 * self.R**2 * self.Tc**2 / self.pc * alpha
20     da = (
21         -0.45724
22         * self.R**2
23         * self.Tc**2
24         / self.pc
25         * kappa
26         * (1 + kappa * (1 - Tr**0.5))
27         * (Tr**-0.5)
28         / self.Tc
29     )
30     b = 0.07780 * self.R * self.Tc / self.pc
31     return a, b, da
32
33     # 计算A和B
34     def AB(self, T, p):
35         a, b, da = self.params(T)
36         A = a * p * 1e6 / (self.R * T) ** 2
37         B = b * p * 1e6 / (self.R * T)
38         return A, B
39
40     # 计算C2, C1, C0
41     def C(self, T, p):
42         A, B = self.AB(T, p)
43         C2 = -(1 - B)
44         C1 = A - 3 * B**2 - 2 * B
45         C0 = -(A * B - B**2 - B**3)
46         return C2, C1, C0
47
48     # 计算压缩因子Z
49     # 液相
50     def Zl(self, T, p):
51         C2, C1, C0 = self.C(T, p)
52         # 牛顿法求解Z
53         Zl = 0.001 # 初始猜测值
54         for _ in range(100):
55             f = Zl**3 + C2 * Zl**2 + C1 * Zl + C0
56             df = 3 * Zl**2 + 2 * C2 * Zl + C1
57             Zl_new = Zl - f / df
58             if abs(Zl_new - Zl) < 1e-6:

```

```

59         break
60         Zl = Zl_new
61     return Zl
62
63     # 气相
64     def Zg(self, T, p):
65         C2, C1, C0 = self.C(T, p)
66         # 牛顿法求解Z
67         Zg = 1.0 # 初始猜测值
68         for _ in range(100):
69             f = Zg**3 + C2 * Zg**2 + C1 * Zg + C0
70             df = 3 * Zg**2 + 2 * C2 * Zg + C1
71             Zg_new = Zg - f / df
72             if abs(Zg_new - Zg) < 1e-6:
73                 break
74             Zg = Zg_new
75         return Zg
76
77     # 计算比体积v
78     # 液相
79     def vl(self, T, p):
80         Zl = self.Zl(T, p)
81         vl = Zl * self.R * T / (p * 1e6) # p从MPa转换为Pa
82         return vl
83
84     # 气相
85     def vg(self, T, p):
86         Zg = self.Zg(T, p)
87         vg = Zg * self.R * T / (p * 1e6) # p从MPa转换为Pa
88         return vg
89
90     # 计算焓的余函数
91     # 液相
92     def h_res_l(self, T, p):
93         a, b, da = self.params(T)
94         Zl = self.Zl(T, p)
95         vl = self.vl(T, p)
96         hr_l = (T * da - a) / (b * np.sqrt(8)) * np.log(
97             (vl - 0.414 * b) / (vl + 2.414 * b)
98             ) + self.R * T * (1 - Zl)

```

```

99     return hr_l
100
101     # 气相
102     def h_res_g(self, T, p):
103         a, b, da = self.params(T)
104         Zg = self.Zg(T, p)
105         vg = self.vg(T, p)
106         hr_g = (T * da - a) / (b * np.sqrt(8)) * np.log(
107             (vg - 0.414 * b) / (vg + 2.414 * b)
108         ) + self.R * T * (1 - Zg)
109         return hr_g
110
111     # 计算熵的余函数
112     # 液相
113     def s_res_l(self, T, p):
114         a, b, da = self.params(T)
115         vl = self.vl(T, p)
116         sr_l = (
117             -self.R * np.log((vl - b) / vl)
118             - self.R * np.log(vl / (self.R * T / (p * 1e6)))
119             + da / (b * np.sqrt(8)) * np.log((vl - 0.414 * b) / (vl + 2.414 * b)
120             )
121         )
122         return sr_l
123
124     # 气相
125     def s_res_g(self, T, p):
126         a, b, da = self.params(T)
127         vg = self.vg(T, p)
128         sr_g = (
129             -self.R * np.log((vg - b) / vg)
130             - self.R * np.log(vg / (self.R * T / (p * 1e6)))
131             + da / (b * np.sqrt(8)) * np.log((vg - 0.414 * b) / (vg + 2.414 * b)
132             )
133         )
134         return sr_g
135
136     # 计算c_p积分
137     def cp(self, T, A, B, C, D):
138         cp = (

```

```

137         A * (T - 273.15)
138         + B / 2 * (T**2 - 273.15**2)
139         + C / 3 * (T**3 - 273.15**3)
140         + D / 4 * (T**4 - 273.15**4)
141     )
142     return cp
143
144     # 计算c_p/T积分
145     def cpT(self, T, A, B, C, D):
146         cp = (
147             A * np.log(T / 273.15)
148             + B * (T - 273.15)
149             + C / 2 * (T**2 - 273.15**2)
150             + D / 3 * (T**3 - 273.15**3)
151         )
152         return cp
153
154     # 计算焓和熵
155     # 液相
156     def h_l(self, T, A, B, C, D, p):
157         h_r_ps_0 = self.h_res_l(273.15, self.ps0)
158         cp0 = self.cp(T, A, B, C, D)
159         h_res_l = self.h_res_l(T, p)
160         hl = 200 * 1e3 + cp0 + (h_r_ps_0 - h_res_l) / self.M # J/kg
161         return hl
162
163     def s_l(self, T, A, B, C, D, p):
164         s_r_ps_0 = self.s_res_l(273.15, self.ps0)
165         cpT = self.cpT(T, A, B, C, D)
166         sr_l = self.s_res_l(T, p)
167         sl = (
168             1e3 + cpT + (s_r_ps_0 - self.R * np.log(p / self.ps0) - sr_l) / self
169             .M
170             ) # J/(kg*K)
171         return sl
172
173     # 气相
174     def h_g(self, T, A, B, C, D, p):
175         h_r_ps_0 = self.h_res_l(273.15, self.ps0)
176         cp0 = self.cp(T, A, B, C, D)

```

```

176     h_res_g = self.h_res_g(T, p)
177     hg = 200 * 1e3 + cp0 + (h_r_ps_0 - h_res_g) / self.M # J/kg
178     return hg
179
180     def s_g(self, T, A, B, C, D, p):
181         s_r_ps_0 = self.s_res_l(273.15, self.ps0)
182         cpT = self.cpT(T, A, B, C, D)
183         sr_g = self.s_res_g(T, p)
184         sg = (
185             1e3 + cpT + (s_r_ps_0 - self.R * np.log(p / self.ps0) - sr_g) / self
186             .M
187             ) # J/(kg*K)
188         return sg

```

4-15