

高等工程热力学编程部分作业

何飏 3123101186

整个编程部分作业均通过 Python 语言完成，程序均在 Windows 10 操作系统下的 Anaconda3 环境中编写和运行，Python 版本为 3.13.2，所用主要第三方库包括 Numpy 和 Matplotlib，用于数值计算和绘图。

编程思路大致如下：

- (1) 导入物质相关参数如临界温度、临界压力、偏心因子、摩尔质量等；
- (2) 利用导入相关参数计算 PR 方程所需相关参数如 a 、 b 、 A 、 B 等；
- (3) 使用牛顿法根据 PR 方程计算压缩因子 Z ，计算时分别考虑液相和气相两种情况，液相初值取 0.001，气相初值取 1.1；
- (4) 根据不同题目利用计算得到的压缩因子计算比体积、焓、熵等热力学性质，或绘制相图等。

然而，使用牛顿法进行压缩因子求解依赖初值，由于牛顿法就近收敛，初值不同会收敛到不同根，如液相迭代收敛到气相根，而且在接近临界压力或温度时，迭代慢，可以考虑采用解析解直接一次性拿到全部实根，再按相态选择。

第三章

3-10

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

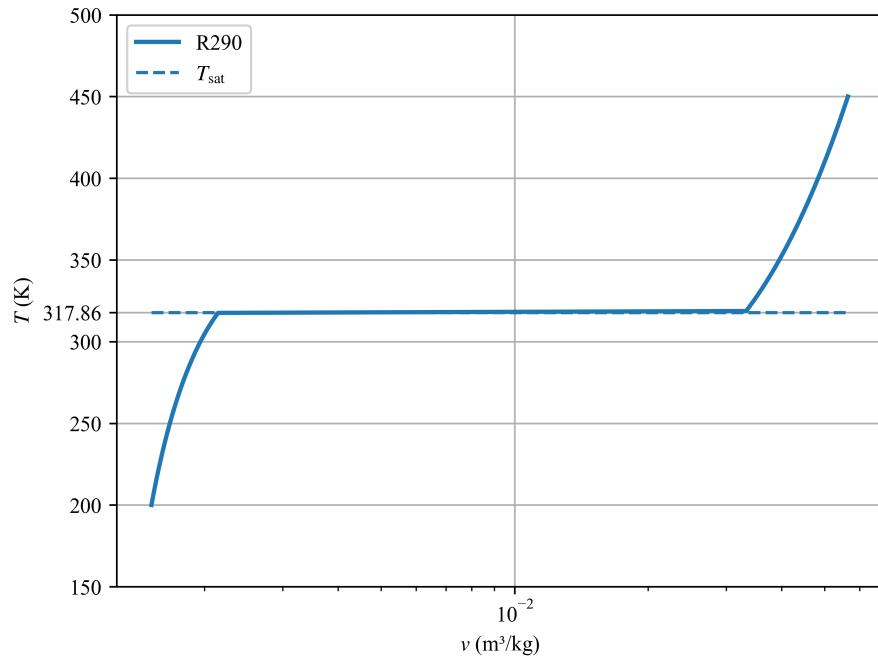


图 1: R290 在 1.4MPa 下的 $T-v$ 图

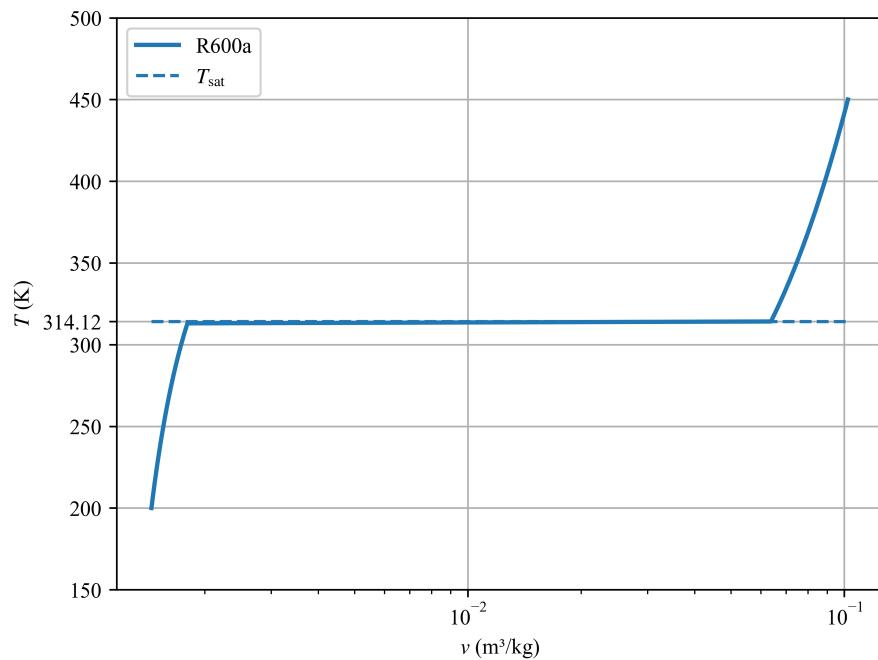


图 2: R600a 在 0.6MPa 下的 $T-v$ 图

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 class PR310:
11     def __init__(self, Tc, Pc, omega, M):
12         self.Tc = Tc # 输入K
13         self.Pc = Pc * 1e6 # 输入MPa
14         self.omega = omega # 无量纲
15         self.M = M / 1000 # 输入g/mol
16
17     R = 8.314462618 # J/(mol*K)
18
19     # 计算a和b
20     def params(self, T):
21         kappa = 0.37464 + 1.54226 * self.omega - 0.26992 * self.omega**2
22         Tr = T / self.Tc
23         alpha = (1 + kappa * (1 - Tr**0.5)) ** 2
24         a = 0.45724 * self.R**2 * self.Tc**2 / self.Pc * alpha
25         b = 0.07780 * self.R * self.Tc / self.Pc
26         return a, b
27
28     # 计算A和B
29     def AB(self, T, p):
30         a, b = self.params(T)
31         A = a * p * 1e6 / (self.R * T) ** 2
32         B = b * p * 1e6 / (self.R * T)
33         return A, B
34
35     # 计算C2, C1, C0
36     def C(self, T, p):
37         A, B = self.AB(T, p)
38         C2 = -(1 - B)
39         C1 = A - 3 * B**2 - 2 * B
40         C0 = -(A * B - B**2 - B**3)
41         return C2, C1, C0
42
43     # 计算压缩因子Z
44     # 液相
45     def Zl(self, T, p):
46         C2, C1, C0 = self.C(T, p)

```

```

47     # 牛顿法求解Z
48     Zl = 0.001 # 初始猜测值
49     for _ in range(100):
50         f = Zl**3 + C2 * Zl**2 + C1 * Zl + C0
51         df = 3 * Zl**2 + 2 * C2 * Zl + C1
52         Zl_new = Zl - f / df
53         if abs(Zl_new - Zl) < 1e-6:
54             break
55         Zl = Zl_new
56     return Zl
57
58     # 气相
59     def Zg(self, T, p):
60         C2, C1, C0 = self.C(T, p)
61         # 牛顿法求解Z
62         Zg = 1.1 # 初始猜测值
63         for _ in range(100):
64             f = Zg**3 + C2 * Zg**2 + C1 * Zg + C0
65             df = 3 * Zg**2 + 2 * C2 * Zg + C1
66             Zg_new = Zg - f / df
67             if abs(Zg_new - Zg) < 1e-6:
68                 break
69             Zg = Zg_new
70     return Zg
71
72     # 计算比体积v
73     # 液相
74     def vl(self, T, p):
75         Zl = self.Zl(T, p)
76         vl = Zl * self.R * T / (p * 1e6 * self.M)
77         return vl
78
79     # 气相
80     def vg(self, T, p):
81         Zg = self.Zg(T, p)
82         vg = Zg * self.R * T / (p * 1e6 * self.M)
83         return vg
84
85     # 画图
86     def plot_Tv(
87         self,
88         fluid_name, # 流体名称
89         p, # 压力 Pa
90         Tsat, # 饱和温度 K
91         T_min, # 温度范围最小值 K
92         T_max, # 温度范围最大值 K
93         nT=220, # 温度点数

```

```

94 ):
95     T_grid = np.linspace(T_min, T_max, nT) # 温度网格
96     v_grid = np.empty_like(T_grid) # 比体积网格
97     # 计算比体积
98     for i, T in enumerate(T_grid):
99         if T < Tsat:
100             v_grid[i] = self.vl(T, p)
101         elif T > Tsat:
102             v_grid[i] = self.vg(T, p)
103         else:
104             v_grid[i] = 0.5 * (self.vl(T, p) + self.vg(T, p))
105     fig, ax = plt.subplots() # 创建图像和坐标轴
106     # 主曲线
107     ax.plot(v_grid, T_grid, linewidth=2, label=fluid_name)
108     xmin, xmax = np.nanmin(v_grid), np.nanmax(v_grid)
109     # Tsat 虚线
110     ax.hlines(Tsat, xmin, xmax, linestyles="--", label=r"$T_{\mathrm{sat}}$")
111     # 标注 Tsat
112     yt = list(ax.get_yticks())
113     # 加入Tsat并排序
114     if not any(abs(t - Tsat) < 1e-8 for t in yt):
115         yt.append(Tsat)
116     yt = np.array(sorted(yt))
117     # 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 (根据范围选择小
118     # 数位)
119     deltaT = T_grid.max() - T_grid.min()
120     labels = []
121     for t in yt:
122         if abs(t - Tsat) < 1e-8 or abs(t - Tsat) < 1e-6 * max(1.0, deltaT):
123             labels.append(f"{Tsat:.2f}")
124         else:
125             # 根据温度范围决定格式, 避免过多小数
126             if deltaT > 50:
127                 labels.append(f"{t:.0f}")
128             else:
129                 labels.append(f"{t:.2f}")
130     ax.set_yticks(yt)
131     ax.set_yticklabels(labels)
132     # 轴标签
133     ax.set_xlabel(r"$v$ (m3/kg)")
134     ax.set_ylabel(r"$T$ (K)")
135     ax.grid(True)
136     ax.set_xscale("log") # 使用对数刻度
137     ax.legend(loc="upper left", frameon=True, fancybox=True, framealpha=0.9)
138     # 固定保存路径为脚本同目录下的 figs 文件夹
139     base_dir = os.path.dirname(os.path.abspath(__file__))

```

```

140     fig_dir = os.path.join(base_dir, "figs")
141     os.makedirs(fig_dir, exist_ok=True)
142
143     # 文件名固定为"流体名称.png"
144     filename = f"{fluid_name}.png"
145     savepath = os.path.join(fig_dir, filename)
146
147     # 保存图像, 固定参数
148     fig.savefig(savepath, dpi=600, bbox_inches="tight", transparent=False)
149     plt.close(fig)
150
151 R290 = PR310(369.89, 4.2512, 0.1521, 44.096)
152 R290.plot_Tv("R290", 1.4, 317.86, 200, 450)
153
154 R600a = PR310(407.81, 3.629, 0.184, 58.122)
155 R600a.plot_Tv("R600a", 0.6, 314.12, 200, 450)

```

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, 调用题 3-10 中程序计算三种制冷剂的 v_g 。计算结果为: $v_{\text{R134a}} = 0.24679\text{m}^3/\text{kg}$, $v_{\text{R1234yf}} = 0.22031\text{m}^3/\text{kg}$, $v_{\text{R1234ze(E)}} = 0.22007\text{m}^3/\text{kg}$

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, 故采用 R1234yf 和 R1234ze(E) 作为 R134a 的替代品是合理的。

程序如下:

```

1 import PR310 # 导入PR310模块
2
3 R134a = PR310.PR310(374.21, 4.0593, 0.326, 102.03)
4 R1234yf = PR310.PR310(367.85, 3.382, 0.276, 114.04)
5 R1234zeE = PR310.PR310(382.51, 3.635, 0.313, 114.04)
6
7 print(R134a.vg(308.15, 0.1))
8 print(R1234yf.vg(308.15, 0.1))
9 print(R1234zeE.vg(308.15, 0.1))

```

3-15

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%。

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

p (MPa)	k_{ij}	v (m^3/mol)	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

计算程序如下：

```
1 class PR315:  
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  
6         self.pc2 = pc2 * 1e6  
7         self.omega1 = omega1  
8         self.omega2 = omega2  
9         self.M1 = M1 / 1e3  
10        self.M2 = M2 / 1e3  
11        self.x1 = x1  
12        self.x2 = 1 - x1  
13        self.kij = kij  
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
36
37 # 计算A和B
38 def AB(self, T, p):
39     a, b = self.params(T)
40     A = a * p * 1e6 / (self.R * T) ** 2
41     B = b * p * 1e6 / (self.R * T)
42     return A, B
43
44 # 计算C2, C1, C0
45 def C(self, T, p):
46     A, B = self.AB(T, p)
47     C2 = -(1 - B)
48     C1 = A - 3 * B**2 - 2 * B
49     C0 = -(A * B - B**2 - B**3)
50     return C2, C1, C0
51
52 # 计算压缩因子Z
53 # 液相
54 def Zl(self, T, p):
55     C2, C1, C0 = self.C(T, p)
56     # 牛顿法求解Z
57     Zl = 0.001 # 初始猜测值
58     for _ in range(100):
59         f = Zl**3 + C2 * Zl**2 + C1 * Zl + C0
60         df = 3 * Zl**2 + 2 * C2 * Zl + C1
61         Zl_new = Zl - f / df
62         if abs(Zl_new - Zl) < 1e-6:
63             break
64         Zl = Zl_new
65     return Zl
66
67 # 气相

```

```

68 def Zg(self, T, p):
69     C2, C1, C0 = self.C(T, p)
70     # 牛顿法求解Z
71     Zg = 1.1 # 初始猜测值
72     for _ in range(100):
73         f = Zg**3 + C2 * Zg**2 + C1 * Zg + C0
74         df = 3 * Zg**2 + 2 * C2 * Zg + C1
75         Zg_new = Zg - f / df
76         if abs(Zg_new - Zg) < 1e-6:
77             break
78         Zg = Zg_new
79     return Zg
80
81 # 计算比体积v
82 # 液相
83 def vl(self, T, p):
84     Zl = self.Zl(T, p)
85     vl = (
86         Zl * self.R * T / (p * 1e6 * (self.x1 * self.M1 + self.x2 * self.M2))
87     ) # p从MPa转换为Pa
88     return vl
89
90 # 气相
91 def vg(self, T, p):
92     Zg = self.Zg(T, p)
93     vg = (
94         Zg * self.R * T / (p * 1e6 * (self.x1 * self.M1 + self.x2 * self.M2))
95     ) # p从MPa转换为Pa
96     return vg
97
98
99 R290_R600a_1 = PR315(
100     369.89, 407.81, 4.2512, 3.629, 0.1521, 0.184, 44.096, 58.122, 0.5, 0.064
101 )
102 R290_R600a_2 = PR315(
103     369.89, 407.81, 4.2512, 3.629, 0.1521, 0.184, 44.096, 58.122, 0.5, 0.1
104 )
105 R290_R600a_3 = PR315(
106     369.89, 407.81, 4.2512, 3.629, 0.1521, 0.184, 44.096, 58.122, 0.5, 0
107 )
108 R290_R600a_4 = PR315(
109     369.89, 407.81, 4.2512, 3.629, 0.1521, 0.184, 44.096, 58.122, 0.5, -0.1
110 )
111 print(R290_R600a_1.vg(300, 0.1))
112 print(R290_R600a_2.vg(300, 0.1))
113 print(R290_R600a_3.vg(300, 0.1))
114 print(R290_R600a_4.vg(300, 0.1))

```

```
115 print(R290_R600a_1.vg(300, 0.2))  
116 print(R290_R600a_2.vg(300, 0.2))  
117 print(R290_R600a_3.vg(300, 0.2))  
118 print(R290_R600a_4.vg(300, 0.2))  
119 print(R290_R600a_1.vg(300, 0.3))  
120 print(R290_R600a_2.vg(300, 0.3))  
121 print(R290_R600a_3.vg(300, 0.3))  
122 print(R290_R600a_4.vg(300, 0.3))
```

第四章

4-13

利用程序分别计算在 1.4MPa 下不同温度 T 下 R290 的液相焓和熵，以及在 0.6MPa 下不同温度 T 下 R600a 的液相焓和熵，计算结果与标准值对比如表 2 和表 3 所示，可以看出，计算结果与标准值误差均小于 1%。

表 2: 1.4MPa 下不同温度 T 下 R290 的液相焓和熵计算结果与标准值对比

T (K)	h (kJ/kg)	h 标准(kJ/kg)	s (kJ/(kg · K))	s 标准(kJ/(kg · K))	h 误差%	s 误差%
260	168.686	170.083	0.876	0.881	0.821	0.567
270	192.542	194.346	0.966	0.973	0.928	0.719
280	217.443	219.262	1.057	1.063	0.830	0.723
290	243.544	244.914	1.149	1.153	0.559	0.347
300	271.043	271.376	1.242	1.243	0.123	0.080

表 3: 0.6MPa 下不同温度 T 下 R600a 的液相焓和熵计算结果与标准值对比

T (K)	h (kJ/kg)	h 标准(kJ/kg)	s (kJ/(kg · K))	s 标准(kJ/(kg · K))	h 误差%	s 误差%
260	171.745	171.556	0.891	0.891	0.110	0
270	193.349	193.946	0.973	0.975	0.308	0.205
280	215.677	216.839	1.054	1.058	0.536	0.378
290	238.773	240.279	1.135	1.140	0.627	0.438
300	262.694	264.277	1.216	1.222	0.158	0.491

程序如下：

```
1 import numpy as np
2
3 class PR413:
4     def __init__(self, Tc, pc, omega, M, ps0):
5         self.Tc = Tc # K
6         self.pc = pc * 1e6
7         self.omega = omega
8         self.M = M / 1e3
9         self.ps0 = ps0
10
11     R = 8.314462618 # J/(mol*K)
12
13     # 计算a和b
14     def params(self, T):
15         kappa = 0.37464 + 1.54226 * self.omega - 0.26992 * self.omega**2
16         Tr = T / self.Tc
```

```

17     alpha = (1 + kappa * (1 - Tr**0.5)) ** 2
18     a = 0.45724 * self.R**2 * self.Tc**2 / self.pc * alpha
19     da = (
20         -0.45724
21         * self.R**2
22         * self.Tc**2
23         / self.pc
24         * kappa
25         * (1 + kappa * (1 - Tr**0.5))
26         * (Tr**-0.5)
27         / self.Tc
28     )
29     b = 0.07780 * self.R * self.Tc / self.pc
30     return a, b, da
31
32 # 计算A和B
33 def AB(self, T, p):
34     a, b, da = self.params(T)
35     A = a * p * 1e6 / (self.R * T) ** 2
36     B = b * p * 1e6 / (self.R * T)
37     return A, B
38
39 # 计算C2, C1, C0
40 def C(self, T, p):
41     A, B = self.AB(T, p)
42     C2 = -(1 - B)
43     C1 = A - 3 * B**2 - 2 * B
44     C0 = -(A * B - B**2 - B**3)
45     return C2, C1, C0
46
47 # 计算压缩因子Z
48 # 液相
49 def Zl(self, T, p):
50     C2, C1, C0 = self.C(T, p)
51     # 牛顿法求解Z
52     Zl = 0.001 # 初始猜测值
53     for _ in range(100):
54         f = Zl**3 + C2 * Zl**2 + C1 * Zl + C0
55         df = 3 * Zl**2 + 2 * C2 * Zl + C1
56         Zl_new = Zl - f / df
57         if abs(Zl_new - Zl) < 1e-6:
58             break
59         Zl = Zl_new
60     return Zl
61
62 # 气相
63 def Zg(self, T, p):

```

```

64     C2, C1, C0 = self.C(T, p)
65     # 牛顿法求解Z
66     Zg = 1.1 # 初始猜测值
67     for _ in range(100):
68         f = Zg**3 + C2 * Zg**2 + C1 * Zg + C0
69         df = 3 * Zg**2 + 2 * C2 * Zg + C1
70         Zg_new = Zg - f / df
71         if abs(Zg_new - Zg) < 1e-6:
72             break
73     Zg = Zg_new
74     return Zg
75
76     # 计算比体积v
77     # 液相
78     def vl(self, T, p):
79         Zl = self.Zl(T, p)
80         v1 = Zl * self.R * T / (p * 1e6)
81         return v1
82
83     # 气相
84     def vg(self, T, p):
85         Zg = self.Zg(T, p)
86         vg = Zg * self.R * T / (p * 1e6)
87         return vg
88
89     # 计算焓的余函数
90     # 液相
91     def h_res_l(self, T, p):
92         a, b, da = self.params(T)
93         Zl = self.Zl(T, p)
94         v1 = self.vl(T, p)
95         hr_l = (T * da - a) / (b * np.sqrt(8)) * np.log(
96             (v1 - 0.414 * b) / (v1 + 2.414 * b)
97         ) + self.R * T * (1 - Zl)
98         return hr_l
99
100    # 气相
101    def h_res_g(self, T, p):
102        a, b, da = self.params(T)
103        Zg = self.Zg(T, p)
104        vg = self.vg(T, p)
105        hr_g = (T * da - a) / (b * np.sqrt(8)) * np.log(
106            (vg - 0.414 * b) / (vg + 2.414 * b)
107        ) + self.R * T * (1 - Zg)
108        return hr_g
109
110    # 计算熵的余函数

```

```

111 # 液相
112 def s_res_l(self, T, p):
113     a, b, da = self.params(T)
114     vl = self.vl(T, p)
115     sr_l = (
116         -self.R * np.log((vl - b) / vl)
117         - self.R * np.log(vl / (self.R * T / (p * 1e6)))
118         + da / (b * np.sqrt(8)) * np.log((vl - 0.414 * b) / (vl + 2.414 * b))
119     )
120     return sr_l
121
122 # 气相
123 def s_res_g(self, T, p):
124     a, b, da = self.params(T)
125     vg = self.vg(T, p)
126     sr_g = (
127         -self.R * np.log((vg - b) / vg)
128         - self.R * np.log(vg / (self.R * T / (p * 1e6)))
129         + da / (b * np.sqrt(8)) * np.log((vg - 0.414 * b) / (vg + 2.414 * b))
130     )
131     return sr_g
132
133 # 计算c_p积分
134 def cp(self, T, A, B, C, D):
135     cp = (
136         A * (T - 273.15)
137         + B / 2 * (T**2 - 273.15**2)
138         + C / 3 * (T**3 - 273.15**3)
139         + D / 4 * (T**4 - 273.15**4)
140     )
141     return cp
142
143 # 计算c_p/T积分
144 def cpT(self, T, A, B, C, D):
145     cp = (
146         A * np.log(T / 273.15)
147         + B * (T - 273.15)
148         + C / 2 * (T**2 - 273.15**2)
149         + D / 3 * (T**3 - 273.15**3)
150     )
151     return cp
152
153 # 计算焓和熵
154 # 液相
155 def h_l(self, T, A, B, C, D, p):
156     h_r_ps_0 = self.h_res_l(273.15, self.ps0)
157     cp0 = self.cp(T, A, B, C, D)

```

```

158     h_res_l = self.h_res_l(T, p)
159     hl = 200 * 1e3 + cp0 + (h_r_ps_0 - h_res_l) / self.M # J/kg
160     return hl
161
162 def s_l(self, T, A, B, C, D, p):
163     s_r_ps_0 = self.s_res_l(273.15, self.ps0)
164     cpT = self.cpT(T, A, B, C, D)
165     sr_l = self.s_res_l(T, p)
166     sl = (
167         1e3 + cpT + (s_r_ps_0 - self.R * np.log(p / self.ps0) - sr_l) / self.M
168     ) # J/(kg*K)
169     return sl
170
171 # 气相
172 def h_g(self, T, A, B, C, D, p):
173     h_r_ps_0 = self.h_res_l(273.15, self.ps0) # 使用液相作为基准
174     cp0 = self.cp(T, A, B, C, D)
175     h_res_g = self.h_res_g(T, p)
176     hg = 200 * 1e3 + cp0 + (h_r_ps_0 - h_res_g) / self.M # J/kg
177     return hg
178
179 def s_g(self, T, A, B, C, D, p):
180     s_r_ps_0 = self.s_res_l(273.15, self.ps0) # 使用液相作为基准
181     cpT = self.cpT(T, A, B, C, D)
182     sr_g = self.s_res_g(T, p)
183     sg = (
184         1e3 + cpT + (s_r_ps_0 - self.R * np.log(p / self.ps0) - sr_g) / self.M
185     ) # J/(kg*K)
186     return sg
187
188
189 R290 = PR413(369.89, 4.2512, 0.1521, 44.096, 0.47446)
190 R600a = PR413(407.81, 3.629, 0.184, 58.122, 0.15696)
191
192 print(R290.h_l(300, -95.80, 6.945, -3.597 * 1e-3, 7.290 * 1e-7, 1.4))
193 print(R290.s_l(300, -95.80, 6.945, -3.597 * 1e-3, 7.290 * 1e-7, 1.4))
194 print(R600a.h_l(300, -23.91, 6.605, -3.176 * 1e-3, 4.981 * 1e-7, 0.6))
195 print(R600a.s_l(300, -23.91, 6.605, -3.176 * 1e-3, 4.981 * 1e-7, 0.6))

```

4-15

取二元作用系数 $k_{ij} = 0.064$, 在 $p=1.0\text{MPa}$ 下不同温度下计算 R290/R600a(50%/50%) 混合制冷剂的焓和熵, 计算结果如表 4 所示, 结果表明, 计算结果与标准值误差很小。

程序如下:

```

1 import numpy as np

```

表 4: 1.0MPa 下不同温度 T 下 R290/R600a(50%/50%) 混合制冷剂的液相焓和熵计算结果与标准值对比

T (K)	h (kJ/kg)	h 标准(kJ/kg)	s (kJ/(kg · K))	s 标准(kJ/(kg · K))	h 误差%	s 误差%
260	170.450	170.056	0.885	0.889	0.232	0.450
270	193.011	193.144	0.970	0.979	0.069	0.919
280	216.459	216.813	1.055	1.066	0.163	1.032
290	240.885	241.124	1.141	1.150	0.099	0.783
300	266.428	266.154	1.228	1.231	0.103	0.244

```

2
3 class PR415:
4     def __init__(self, Tc1, pc1, omega1, M1, x1, Tc2, pc2, omega2, M2, kij, ps0):
5         self.Tc1 = Tc1 # K
6         self.pc1 = pc1 * 1e6
7         self.omega1 = omega1
8         self.M1 = M1 / 1e3
9         self.x1 = x1
10
11        self.Tc2 = Tc2 # K
12        self.pc2 = pc2 * 1e6
13        self.omega2 = omega2
14        self.M2 = M2 / 1e3
15        self.x2 = 1 - x1
16
17        self.ps0 = ps0
18
19        self.kij = kij
20
21 R = 8.314462618 # J/(mol*K)
22
23 # 计算a和b
24 def params(self, T):
25     kappa1 = 0.37464 + 1.54226 * self.omega1 - 0.26992 * self.omega1**2
26     kappa2 = 0.37464 + 1.54226 * self.omega2 - 0.26992 * self.omega2**2
27     Tr1 = T / self.Tc1
28     Tr2 = T / self.Tc2
29     alpha1 = (1 + kappa1 * (1 - Tr1**0.5)) ** 2
30     alpha2 = (1 + kappa2 * (1 - Tr2**0.5)) ** 2
31     a1 = 0.45724 * self.R**2 * self.Tc1**2 / self.pc1 * alpha1
32     a2 = 0.45724 * self.R**2 * self.Tc2**2 / self.pc2 * alpha2
33     da1 = (
34         -0.45724
35         * self.R**2
36         * self.Tc1**2

```

```

37         / self.pc1
38         * kappa1
39         * (1 + kappa1 * (1 - Tr1**0.5))
40         * (Tr1**-0.5)
41         / self.Tc1
42     )
43     da2 = (
44         -0.45724
45         * self.R**2
46         * self.Tc2**2
47         / self.pc2
48         * kappa2
49         * (1 + kappa2 * (1 - Tr2**0.5))
50         * (Tr2**-0.5)
51         / self.Tc2
52     )
53     b1 = 0.07780 * self.R * self.Tc1 / self.pc1
54     b2 = 0.07780 * self.R * self.Tc2 / self.pc2
55
56     a = (
57         self.x1**2 * a1
58         + self.x2**2 * a2
59         + 2 * self.x1 * self.x2 * (a1 * a2) ** 0.5 * (1 - self.kij)
60     )
61     b = self.x1 * b1 + self.x2 * b2
62     da = (
63         self.x1**2 * da1
64         + self.x2**2 * da2
65         + self.x1
66         * self.x2
67         * (1 - self.kij)
68         * ((a2 / a1) ** 0.5 * da1 + (a1 / a2) ** 0.5 * da2)
69     )
70     return a, b, da
71
72 # 计算A和B
73 def AB(self, T, p):
74     a, b, da = self.params(T)
75     A = a * p * 1e6 / (self.R * T) ** 2
76     B = b * p * 1e6 / (self.R * T)
77     return A, B
78
79 # 计算C2, C1, C0
80 def C(self, T, p):
81     A, B = self.AB(T, p)
82     C2 = -(1 - B)
83     C1 = A - 3 * B**2 - 2 * B

```

```

84     C0 = -(A * B - B**2 - B**3)
85     return C2, C1, C0
86
87 # 计算压缩因子Z
88 # 液相
89 def Zl(self, T, p):
90     C2, C1, C0 = self.C(T, p)
91     # 牛顿法求解Z
92     Zl = 0.001 # 初始猜测值
93     for _ in range(100):
94         f = Zl**3 + C2 * Zl**2 + C1 * Zl + C0
95         df = 3 * Zl**2 + 2 * C2 * Zl + C1
96         Zl_new = Zl - f / df
97         if abs(Zl_new - Zl) < 1e-6:
98             break
99         Zl = Zl_new
100    return Zl
101
102 # 气相
103 def Zg(self, T, p):
104     C2, C1, C0 = self.C(T, p)
105     # 牛顿法求解Z
106     Zg = 1.1 # 初始猜测值
107     for _ in range(100):
108         f = Zg**3 + C2 * Zg**2 + C1 * Zg + C0
109         df = 3 * Zg**2 + 2 * C2 * Zg + C1
110         Zg_new = Zg - f / df
111         if abs(Zg_new - Zg) < 1e-6:
112             break
113         Zg = Zg_new
114     return Zg
115
116 # 计算比体积v
117 # 液相
118 def vl(self, T, p):
119     Zl = self.Zl(T, p)
120     vl = Zl * self.R * T / (p * 1e6)
121     return vl
122
123 # 气相
124 def vg(self, T, p):
125     Zg = self.Zg(T, p)
126     vg = Zg * self.R * T / (p * 1e6)
127     return vg
128
129 # 计算焓的余函数
130 # 液相

```

```

131 def h_res_l(self, T, p):
132     a, b, da = self.params(T)
133     Zl = self.Zl(T, p)
134     vl = self.vl(T, p)
135     hr_l = (T * da - a) / (b * np.sqrt(8)) * np.log(
136         (vl - 0.414 * b) / (vl + 2.414 * b)
137     ) + self.R * T * (1 - Zl)
138     return hr_l
139
140 # 气相
141 def h_res_g(self, T, p):
142     a, b, da = self.params(T)
143     Zg = self.Zg(T, p)
144     vg = self.vg(T, p)
145     hr_g = (T * da - a) / (b * np.sqrt(8)) * np.log(
146         (vg - 0.414 * b) / (vg + 2.414 * b)
147     ) + self.R * T * (1 - Zg)
148     return hr_g
149
150 # 计算熵的余函数
151 # 液相
152 def s_res_l(self, T, p):
153     a, b, da = self.params(T)
154     vl = self.vl(T, p)
155     sr_l = (
156         -self.R * np.log((vl - b) / vl)
157         - self.R * np.log(vl / (self.R * T / (p * 1e6)))
158         + da / (b * np.sqrt(8)) * np.log((vl - 0.414 * b) / (vl + 2.414 * b))
159     )
160     return sr_l
161
162 # 气相
163 def s_res_g(self, T, p):
164     a, b, da = self.params(T)
165     vg = self.vg(T, p)
166     sr_g = (
167         -self.R * np.log((vg - b) / vg)
168         - self.R * np.log(vg / (self.R * T / (p * 1e6)))
169         + da / (b * np.sqrt(8)) * np.log((vg - 0.414 * b) / (vg + 2.414 * b))
170     )
171     return sr_g
172
173 # 计算c_p积分
174 def cp(self, T, A1, A2, B1, B2, C1, C2, D1, D2):
175     cp = (
176         (A1 + A2) * 0.5 * (T - 273.15)
177         + (B1 + B2) * 0.5 / 2 * (T**2 - 273.15**2)

```

```

178         + (C1 + C2) * 0.5 / 3 * (T**3 - 273.15**3)
179         + (D1 + D2) * 0.5 / 4 * (T**4 - 273.15**4)
180     )
181     return cp
182
183 # 计算c_p/T积分
184 def cpT(self, T, A1, A2, B1, B2, C1, C2, D1, D2):
185     cp = (
186         (A1 + A2) * 0.5 * np.log(T / 273.15)
187         + (B1 + B2) * 0.5 * (T - 273.15)
188         + (C1 + C2) * 0.5 / 2 * (T**2 - 273.15**2)
189         + (D1 + D2) * 0.5 / 3 * (T**3 - 273.15**3)
190     )
191     return cp
192
193 # 计算焓和熵
194 # 液相
195 def h_l(self, T, A1, A2, B1, B2, C1, C2, D1, D2, p):
196     h_r_ps_0 = self.h_res_l(273.15, self.ps0)
197     cp0 = self.cp(T, A1, A2, B1, B2, C1, C2, D1, D2)
198     h_res_l = self.h_res_l(T, p)
199     hl = (
200         200 * 1e3
201         + cp0
202         + (h_r_ps_0 - h_res_l) / (self.x1 * self.M1 + self.x2 * self.M2)
203     ) # J/kg
204     return hl
205
206 def s_l(self, T, A1, A2, B1, B2, C1, C2, D1, D2, p):
207     s_r_ps_0 = self.s_res_l(273.15, self.ps0)
208     cpT = self.cpT(T, A1, A2, B1, B2, C1, C2, D1, D2)
209     sr_l = self.s_res_l(T, p)
210     sl = (
211         1e3
212         + cpT
213         + (s_r_ps_0 - self.R * np.log(p / self.ps0) - sr_l)
214         / (self.x1 * self.M1 + self.x2 * self.M2)
215     ) # J/(kg*K)
216     return sl
217
218 # 气相
219 def h_g(self, T, A1, A2, B1, B2, C1, C2, D1, D2, p):
220     h_r_ps_0 = self.h_res_l(273.15, self.ps0) # 使用液相作为基准
221     cp0 = self.cp(T, A1, A2, B1, B2, C1, C2, D1, D2)
222     h_res_g = self.h_res_g(T, p)
223     hg = (
224         200 * 1e3

```

```

225         + cp0
226         + (h_r_ps_0 - h_res_g) / (self.x1 * self.M1 + self.x2 * self.M2)
227     ) # J/kg
228     return hg
229
230 def s_g(self, T, A1, A2, B1, B2, C1, C2, D1, D2, p):
231     s_r_ps_0 = self.s_res_l(273.15, self.ps0) # 使用液相作为基准
232     cpT = self.cpT(T, A1, A2, B1, B2, C1, C2, D1, D2)
233     sr_g = self.s_res_g(T, p)
234     sg = (
235         1e3
236         + cpT
237         + (s_r_ps_0 - self.R * np.log(p / self.ps0) - sr_g)
238         / (self.x1 * self.M1 + self.x2 * self.M2)
239     ) # J/(kg*K)
240     return sg
241
242
243 R290R600a = PR415(
244     Tc1=369.89,
245     pc1=4.2512,
246     omega1=0.1521,
247     M1=44.096, # R290
248     x1=0.5,
249     Tc2=407.81,
250     pc2=3.629,
251     omega2=0.184,
252     M2=58.122, # R600a
253     kij=0.064,
254     ps0=0.32979,
255 )
256
257 # 300K下计算比焓和比熵
258 print(
259     R290R600a.h_l(
260         300,
261         -95.80,
262         -23.91,
263         6.945,
264         6.605,
265         -3.597 * 1e-3,
266         -3.176 * 1e-3,
267         7.290 * 1e-7,
268         4.981 * 1e-7,
269         1.0,
270     )
271 )

```

```
272 print(            
273     R290R600a.s_1(  
274         300,  
275         -95.80,  
276         -23.91,  
277         6.945,  
278         6.605,  
279         -3.597 * 1e-3,  
280         -3.176 * 1e-3,  
281         7.290 * 1e-7,  
282         4.981 * 1e-7,  
283         1.0,  
284     )  
285 )
```

第六章

6-11

推导过程见作业手写部分，1.4MPa 下 R290/R600a 混合制冷剂的 $\hat{\phi}$ -T 图和 \hat{f} -T 图如下：

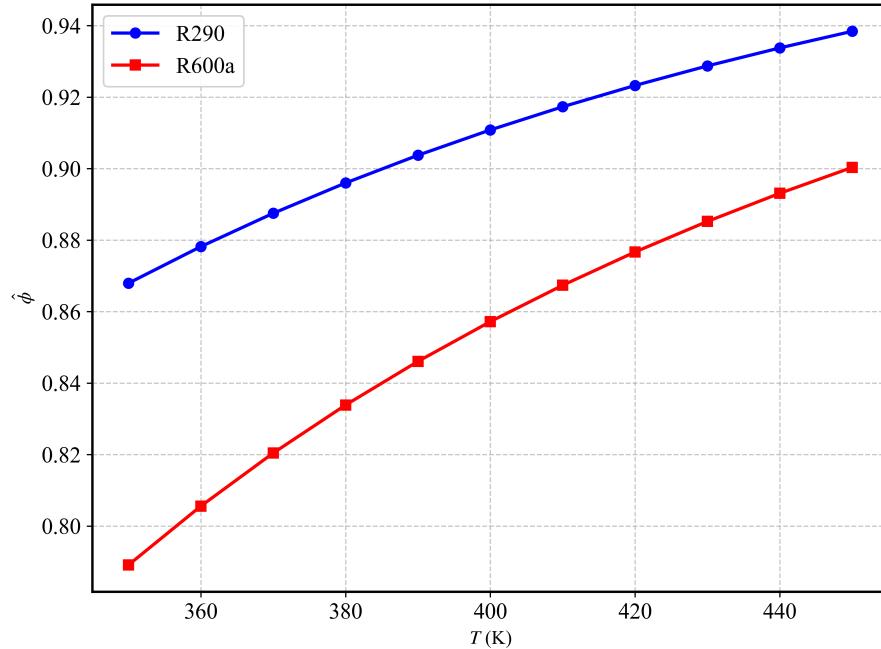


图 3: 1.4MPa 下 R290/R600a 混合制冷剂的 $\hat{\phi}$ -T 图

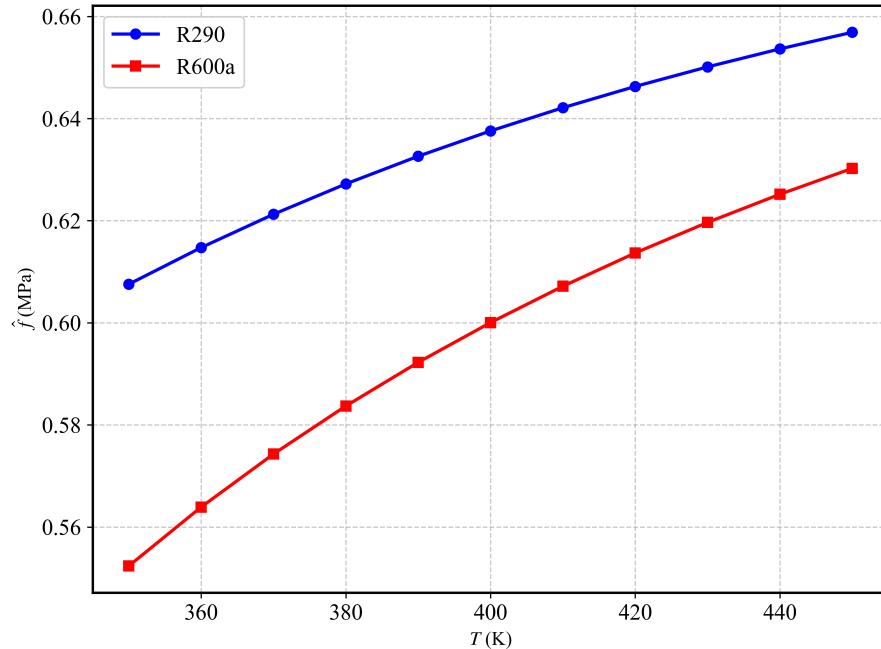


图 4: 1.4MPa 下 R290/R600a 混合制冷剂的 \hat{f} -T 图

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 plt.rcParams["font.size"] = 14 # 增大全局字体
10 plt.rcParams["axes.linewidth"] = 1.5 # 增粗坐标轴
11
12
13 class PR611:
14     def __init__(self, Tc1, pc1, omega1, Tc2, pc2, omega2, x1, kij):
15         self.Tc1 = Tc1
16         self.pc1 = pc1 * 1e6
17         self.omega1 = omega1
18         self.Tc2 = Tc2
19         self.pc2 = pc2 * 1e6
20         self.omega2 = omega2
21         self.x1 = x1
22         self.x2 = 1 - x1
23         self.kij = kij
24
25 R = 8.314462618 # J/(mol · K)
26
27 # 计算a和b
28 def params(self, T):
29     kappa1 = 0.37464 + 1.54226 * self.omega1 - 0.26992 * self.omega1**2
30     kappa2 = 0.37464 + 1.54226 * self.omega2 - 0.26992 * self.omega2**2
31     Tr1 = T / self.Tc1
32     Tr2 = T / self.Tc2
33     alpha1 = (1 + kappa1 * (1 - Tr1**0.5)) ** 2
34     alpha2 = (1 + kappa2 * (1 - Tr2**0.5)) ** 2
35     a1 = 0.45724 * self.R**2 * self.Tc1**2 / self.pc1 * alpha1
36     a2 = 0.45724 * self.R**2 * self.Tc2**2 / self.pc2 * alpha2
37     da1 = (
38         -0.45724
39         * self.R**2
40         * self.Tc1**2
41         / self.pc1
42         * kappa1
43         * (1 + kappa1 * (1 - Tr1**0.5))
44         * (Tr1**-0.5)
45         / self.Tc1
46     )

```

```

47     da2 = (
48         -0.45724
49         * self.R**2
50         * self.Tc2**2
51         / self.pc2
52         * kappa2
53         * (1 + kappa2 * (1 - Tr2**0.5))
54         * (Tr2**-0.5)
55         / self.Tc2
56     )
57     b1 = 0.07780 * self.R * self.Tc1 / self.pc1
58     b2 = 0.07780 * self.R * self.Tc2 / self.pc2
59
60     a = (
61         self.x1**2 * a1
62         + self.x2**2 * a2
63         + 2 * self.x1 * self.x2 * (a1 * a2) ** 0.5 * (1 - self.kij)
64     )
65     b = self.x1 * b1 + self.x2 * b2
66     da = (
67         self.x1**2 * da1
68         + self.x2**2 * da2
69         + self.x1
70         * self.x2
71         * (1 - self.kij)
72         * ((a2 / a1) ** 0.5 * da1 + (a1 / a2) ** 0.5 * da2)
73     )
74     return a1, a2, a, b1, b2, b, da
75
76 # 计算A和B
77 def AB(self, T, p):
78     a1, a2, a, b1, b2, b, da = self.params(T)
79     A = a * p * 1e6 / (self.R * T) ** 2
80     B = b * p * 1e6 / (self.R * T)
81     return A, B
82
83 # 计算C2, C1, C0
84 def C(self, T, p):
85     A, B = self.AB(T, p)
86     C2 = -(1 - B)
87     C1 = A - 3 * B**2 - 2 * B
88     C0 = -(A * B - B**2 - B**3)
89     return C2, C1, C0
90
91 # 计算压缩因子Z
92 # 气相
93 def Zg(self, T, p):

```

```

94     C2, C1, C0 = self.C(T, p)
95     # 牛顿法求解Z
96     Zg = 1.1 # 初始猜测值
97     for _ in range(100):
98         f = Zg**3 + C2 * Zg**2 + C1 * Zg + C0
99         df = 3 * Zg**2 + 2 * C2 * Zg + C1
100        Zg_new = Zg - f / df
101        if abs(Zg_new - Zg) < 1e-6:
102            break
103        Zg = Zg_new
104    return Zg
105
106    # 计算比体积v
107    # 气相
108    def vg(self, T, p):
109        Zg = self.Zg(T, p)
110        vg = Zg * self.R * T / (p * 1e6)
111        return vg # m3/mol
112
113    # 计算逸度系数
114    # 气相
115    def phi_g(self, T, p):
116        Zg = self.Zg(T, p)
117        a1, a2, a, b1, b2, b, da = self.params(T)
118        A, B = self.AB(T, p)
119        phi_g1 = np.exp(
120            (b1 / b) * (Zg - 1)
121            - np.log(Zg - B)
122            - A
123            / (B * np.sqrt(8))
124            *
125            (
126                2 * (self.x2 * (1 - self.kij) * (a1 * a2) ** 0.5 + self.x1 * a1) / a
127                - b1 / b
128            )
129            * np.log((Zg + 2.414 * B) / (Zg - 0.414 * B))
130        )
131        phi_g2 = np.exp(
132            (b2 / b) * (Zg - 1)
133            - np.log(Zg - B)
134            - A
135            / (B * np.sqrt(8))
136            *
137            (
138                2 * (self.x1 * (1 - self.kij) * (a1 * a2) ** 0.5 + self.x2 * a2) / a
139                - b2 / b
140            )
141            * np.log((Zg + 2.414 * B) / (Zg - 0.414 * B))
142        )

```

```

141     return phi_g1, phi_g2
142
143 # 计算逸度
144 # 气相
145 def f_g(self, T, p): # MPa
146     phi_g1, phi_g2 = self.phi_g(T, p)
147     f_g1 = self.x1 * phi_g1 * p
148     f_g2 = self.x2 * phi_g2 * p
149     return f_g1, f_g2
150
151 # 绘制溶液气相f-T、phi-T图
152 def plot_fT(self, fluid_name, p, T_min, T_max, nT=11):
153     T_grid = np.linspace(T_min, T_max, nT) # 温度网格
154     phi_grid1 = np.empty_like(T_grid) # 组分1逸度系数网格
155     phi_grid2 = np.empty_like(T_grid) # 组分2逸度系数网格
156     f_grid1 = np.empty_like(T_grid) # 组分1逸度网格
157     f_grid2 = np.empty_like(T_grid) # 组分2逸度网格
158
159     base_dir = os.path.dirname(os.path.abspath(__file__))
160     fig_dir = os.path.join(base_dir, "figs")
161     os.makedirs(fig_dir, exist_ok=True)
162
163 # 计算逸度系数和逸度
164 for i, T in enumerate(T_grid):
165     phi_g1, phi_g2 = self.phi_g(T, p)
166     f_g1, f_g2 = self.f_g(T, p)
167     phi_grid1[i] = phi_g1
168     phi_grid2[i] = phi_g2
169     f_grid1[i] = f_g1
170     f_grid2[i] = f_g2
171
172 # T-phi图
173 fig_phi = plt.figure(figsize=(8, 6))
174 ax_phi = fig_phi.add_subplot(1, 1, 1)
175 ax_phi.plot(T_grid, phi_grid1, "b-o", linewidth=2, label="R290", markersize=6)
176 ax_phi.plot(T_grid, phi_grid2, "r-s", linewidth=2, label="R600a", markersize=6)
177 ax_phi.set_xlabel(r"$T$ (K)", fontsize=12)
178 ax_phi.set_ylabel(r"$\hat{\phi}$", fontsize=12)
179 ax_phi.grid(True, linestyle="--", alpha=0.7)
180 ax_phi.legend(loc="best", frameon=True, fancybox=True, framealpha=0.9)
181 fig_phi.tight_layout()
182 savepath_phi = os.path.join(fig_dir, f"{fluid_name}_phi.png")
183 fig_phi.savefig(savepath_phi, dpi=600, bbox_inches="tight", transparent=False)
184 plt.close(fig_phi)
185
186 # T-f图
187 fig_f = plt.figure(figsize=(8, 6))

```

```

188     ax_f = fig_f.add_subplot(111)
189     ax_f.plot(T_grid, f_grid1, "b-o", linewidth=2, label="R290", markersize=6)
190     ax_f.plot(T_grid, f_grid2, "r-s", linewidth=2, label="R600a", markersize=6)
191     ax_f.set_xlabel(r"$T$ (K)", fontsize=12)
192     ax_f.set_ylabel(r"$\hat{f}$ (MPa)", fontsize=12)
193     ax_f.grid(True, linestyle="--", alpha=0.7)
194     ax_f.legend(loc="best", frameon=True, fancybox=True, framealpha=0.9)
195     fig_f.tight_layout()
196     savepath_f = os.path.join(fig_dir, f"{fluid_name}_f.png")
197     fig_f.savefig(savepath_f, dpi=600, bbox_inches="tight", transparent=False)
198     plt.close(fig_f)
199
200
201 R290R600a = PR611(
202     Tc1=369.89,
203     pc1=4.2512,
204     omega1=0.1521,
205     Tc2=407.81,
206     pc2=3.629,
207     omega2=0.184,
208     x1=0.5,
209     kij=0.064,
210 )
211
212 R290R600a.plot_fT("R290R600a", 1.4, 350, 450, 11)

```

第七章

7-3

式 (7.13) 为:

$$\ln p_r = \ln T_r [A_1 + A_2 \tau^{1.89} + A_3 \tau^{5.67}]$$

其中 $p_r = p/p_c$, $T_r = T/T_c$, $\tau = 1 - T_r$ 。整理得:

$$p = p_c \left(\frac{T}{T_c} \right)^{A_1 + A_2 (1 - \frac{T}{T_c})^{1.89} + A_3 (1 - \frac{T}{T_c})^{5.67}}$$

由书中表 7.1 可知甲烷、乙烷、丙烷和丁烷项-谭方程的相关常数, 带入求解。绘制的蒸汽压曲线如下图所示:

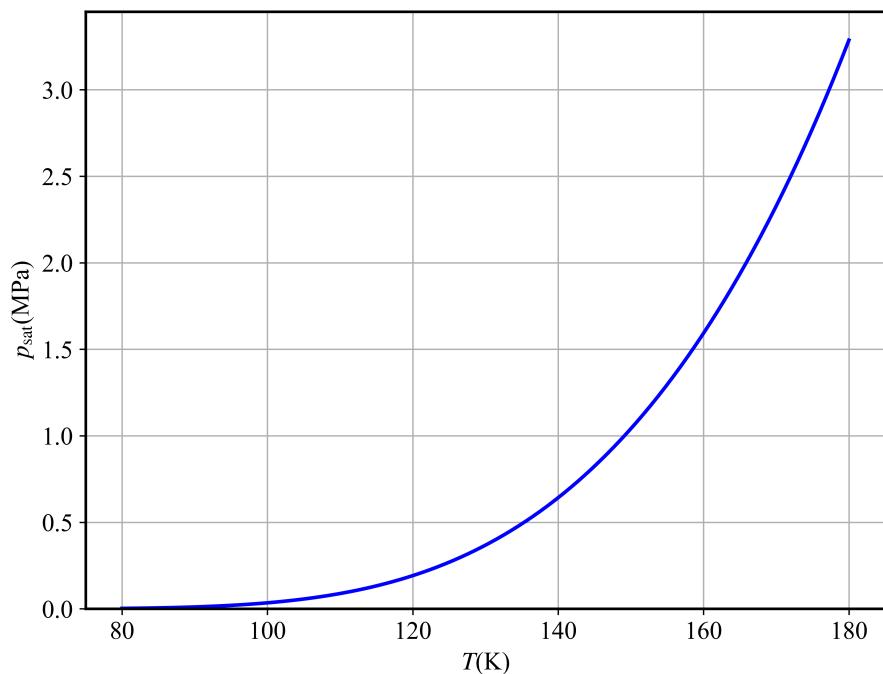


图 5: 甲烷的蒸汽压曲线

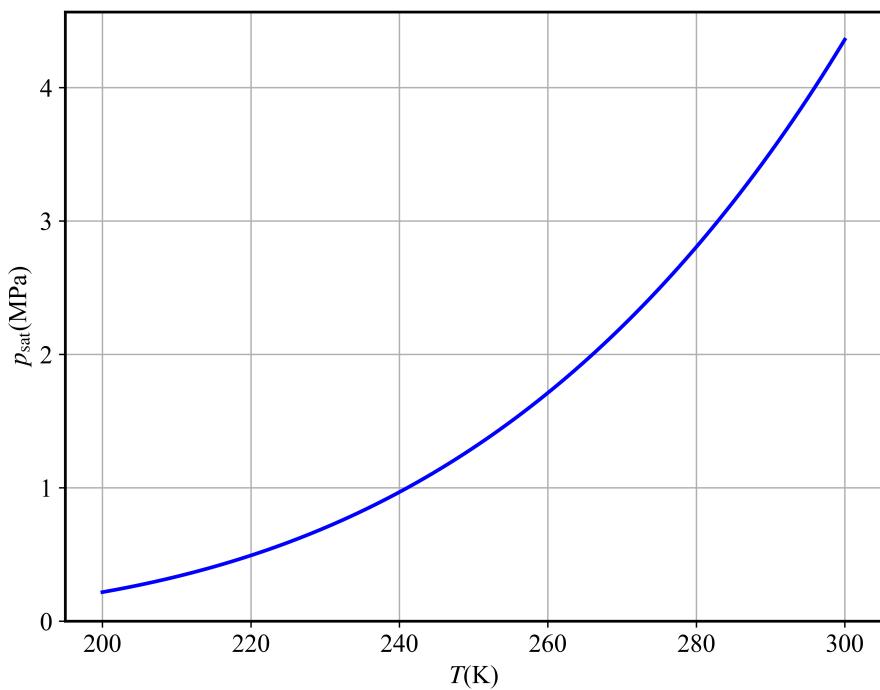


图 6: 乙烷的蒸汽压曲线

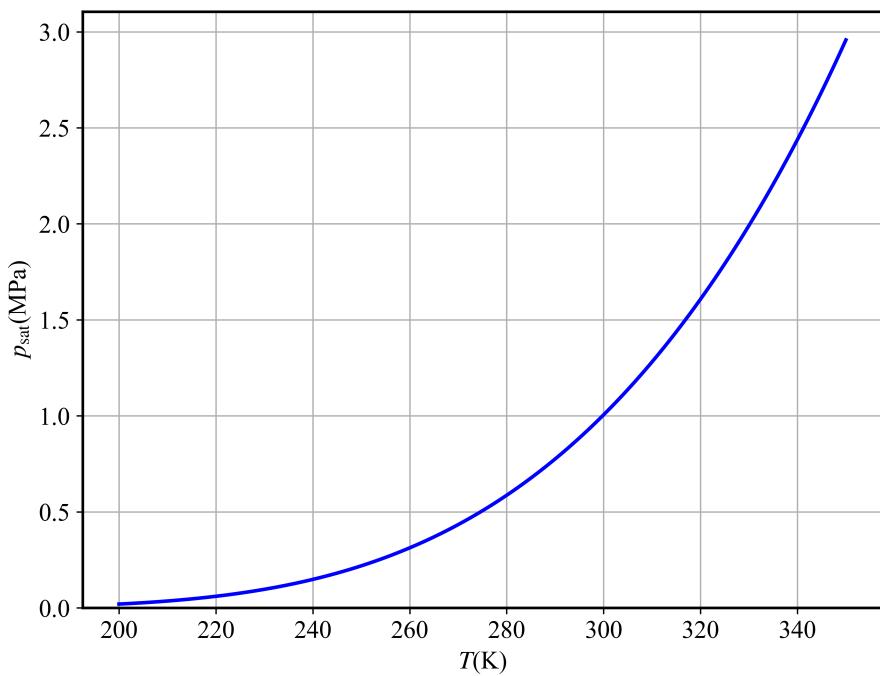


图 7: 丙烷的蒸汽压曲线

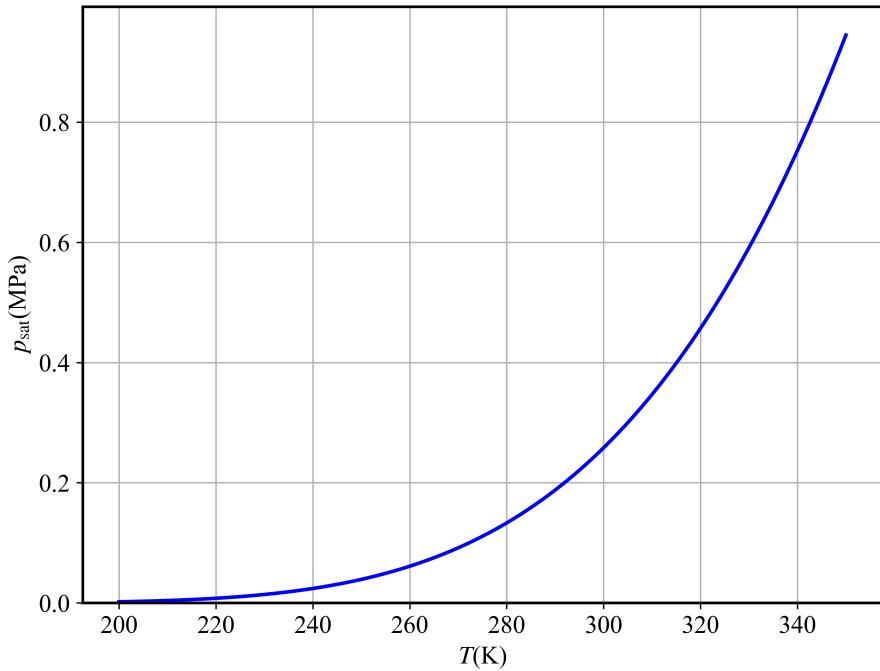


图 8: 丁烷的蒸汽压曲线

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 plt.rcParams["font.size"] = 14 # 增大全局字体
10 plt.rcParams["axes.linewidth"] = 1.5 # 增粗坐标轴
11
12
13 class PR73:
14     def __init__(self, Tc, pc, A1, A2, A3):
15         self.Tc = Tc # K
16         self.pc = pc * 1e6 # Pa, 输入MPa
17         self.A1 = A1
18         self.A2 = A2
19         self.A3 = A3
20
21     def psat(self, T):
22         Tr = T / self.Tc
23         p = self.pc * (Tr) ** (
24             self.A1 + self.A2 * (1 - Tr) ** 1.89 + self.A3 * (1 - Tr) ** 5.67
25         )
26         return p / 1e6 # 输出MPa

```

```

27
28 def plot_psat(self, fluid_name, Tmin, Tmax, nt):
29     T_grid = np.linspace(Tmin, Tmax, nt)
30     psat_grid = np.zeros(nt)
31     for i, T in enumerate(T_grid):
32         psat_grid[i] = self.psat(T)
33
34     base_dir = os.path.dirname(os.path.abspath(__file__))
35     fig_dir = os.path.join(base_dir, "figs")
36     os.makedirs(fig_dir, exist_ok=True)
37
38     fig = plt.figure(figsize=(8, 6))
39     ax = fig.add_subplot(1, 1, 1)
40     ax.plot(T_grid, psat_grid, "b-", linewidth=2)
41     ax.set_xlim(bottom=0)
42     ax.set_xlabel("T (K)", fontsize=14)
43     ax.set_ylabel(r"$P_{\text{sat}}$ (MPa)", fontsize=14)
44     ax.grid(True)
45
46     # 保存图像
47     savepath = os.path.join(fig_dir, f"{fluid_name}.png")
48     fig.savefig(savepath, dpi=600, bbox_inches="tight", transparent=False)
49     plt.close(fig)
50
51
52 CH4 = PR73(190.551, 4.5992, 5.87304544, 6.23280143, 13.0721578)
53 CH4.plot_psat("甲烷", 80, 180, 100)
54 C2H6 = PR73(305.33, 4.8717, 6.30717658, 7.47042131, 17.0958137)
55 C2H6.plot_psat("乙烷", 200, 300, 100)
56 C3H8 = PR73(369.80, 4.239, 6.50580501, 8.6776247, 18.0116214)
57 C3H8.plot_psat("丙烷", 200, 350, 150)
58 C4H10 = PR73(425.2, 3.8, 6.81692028, 8.77671813, 23.7680492)
59 C4H10.plot_psat("丁烷", 200, 350, 150)

```

7-4

制冷剂 R290、R600a、R1234yf 和 R1234ze(E) 的 p - T 相图如下所示：

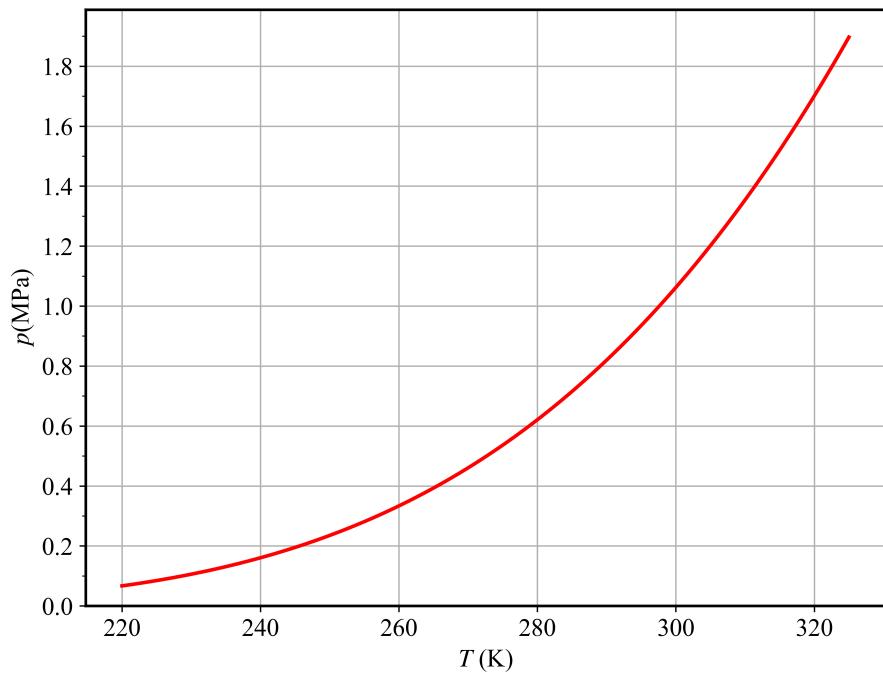


图 9: R290 的 p - T 相图

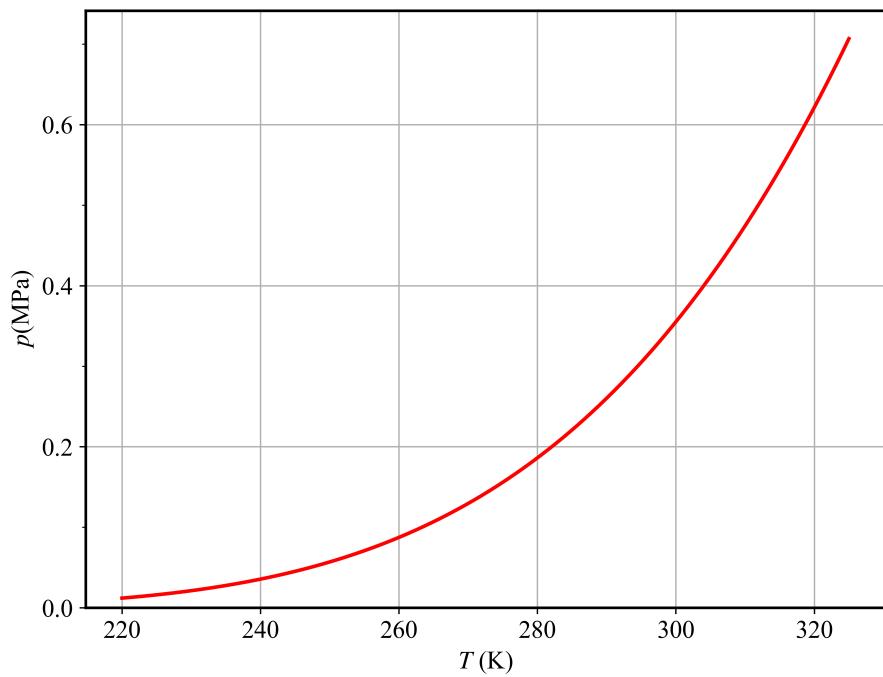


图 10: R600a 的 p - T 相图

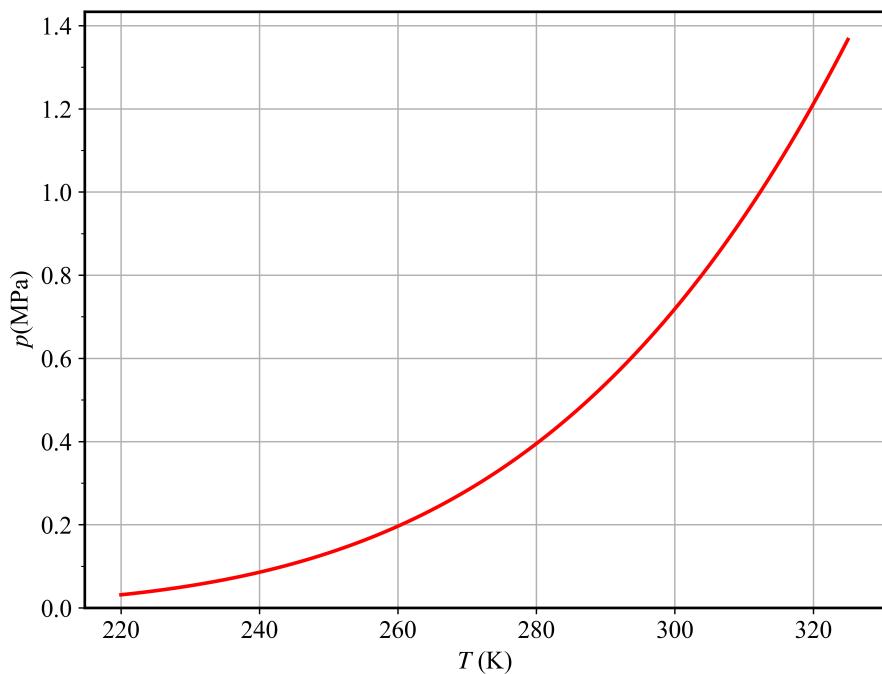


图 11: R1234yf 的 p - T 相图

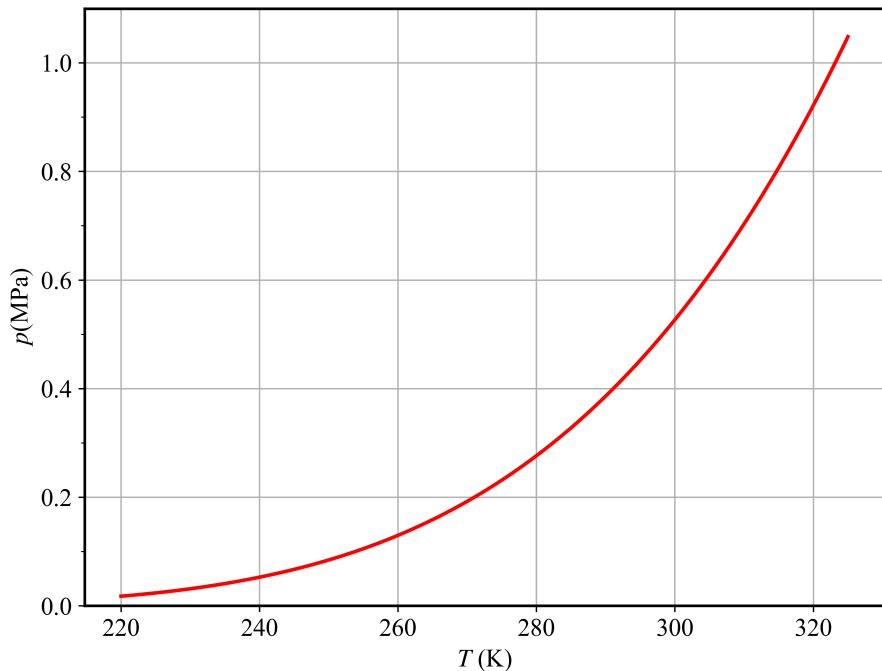


图 12: R1234ze(E) 的 p - T 相图

程序如下：

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from matplotlib.ticker import MultipleLocator
4 import os
5
6 # 使用 Times New Roman 作为 matplotlib 全局字体

```

```

7 plt.rcParams["font.family"] = "serif"
8 plt.rcParams["font.serif"] = ["Times New Roman"]
9 plt.rcParams["mathtext.fontset"] = "stix"
10 plt.rcParams["font.size"] = 14 # 增大全局字体
11 plt.rcParams["axes.linewidth"] = 1.5 # 增粗坐标轴
12
13
14 class PR74:
15     def __init__(self, Tc, pc, omega):
16         self.Tc = Tc
17         self.pc = pc * 1e6
18         self.omega = omega
19
20     R = 8.314462618 # J/(mol*K)
21
22     def params(self, T):
23         kappa = 0.37464 + 1.54226 * self.omega - 0.26992 * self.omega**2
24         Tr = T / self.Tc
25         alpha = (1 + kappa * (1 - Tr**0.5)) ** 2
26         a = 0.45724 * self.R**2 * self.Tc**2 / self.pc * alpha
27         b = 0.07780 * self.R * self.Tc / self.pc
28         da = (
29             -0.45724
30             * self.R**2
31             * self.Tc**2
32             / self.pc
33             * kappa
34             * (1 + kappa * (1 - Tr**0.5))
35             * (Tr**-0.5)
36             / self.Tc
37         )
38         return a, b, da
39
40     def AB(self, T, p):
41         a, b, da = self.params(T)
42         A = a * p * 1e6 / (self.R * T) ** 2
43         B = b * p * 1e6 / (self.R * T)
44         return A, B
45
46     def C(self, T, p):
47         A, B = self.AB(T, p)
48         C2 = -(1 - B)
49         C1 = A - 3 * B**2 - 2 * B
50         C0 = -(A * B - B**2 - B**3)
51         return C2, C1, C0
52
53     def Zl(self, T, p):

```

```

54     C2, C1, C0 = self.C(T, p)
55     Zl = 0.001
56     for _ in range(100):
57         f = Zl**3 + C2 * Zl**2 + C1 * Zl + C0
58         df = 3 * Zl**2 + 2 * C2 * Zl + C1
59         Zl_new = Zl - f / df
60         if abs(Zl_new - Zl) < 1e-6:
61             break
62         Zl = Zl_new
63     return Zl
64
65 def Zg(self, T, p):
66     C2, C1, C0 = self.C(T, p)
67     Zg = 1.1
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 def vl(self, T, p):
78     Zl = self.Zl(T, p)
79     vl = Zl * self.R * T / (p * 1e6)
80     return vl
81
82 def vg(self, T, p):
83     Zg = self.Zg(T, p)
84     vg = Zg * self.R * T / (p * 1e6)
85     return vg
86
87 def phi_l(self, T, p):
88     Zl = self.Zl(T, p)
89     a, b, da = self.params(T)
90     A, B = self.AB(T, p)
91     ln_phi_l = (
92         Zl
93         - 1
94         - np.log(Zl - B)
95         - A
96         / (2 * (2**0.5) * B)
97         * np.log((Zl + (1 + 2**0.5) * B) / (Zl + (1 - 2**0.5) * B))
98     )
99     phi_l = np.exp(ln_phi_l)
100    return phi_l

```

```

101
102     def phi_g(self, T, p):
103         Zg = self.Zg(T, p)
104         a, b, da = self.params(T)
105         A, B = self.AB(T, p)
106         ln_phi_g = (
107             Zg
108             - 1
109             - np.log(Zg - B)
110             - A
111             / (2 * (2**0.5) * B)
112             * np.log((Zg + (1 + 2**0.5) * B) / (Zg + (1 - 2**0.5) * B))
113         )
114         phi_g = np.exp(ln_phi_g)
115         return phi_g
116
117     # 计算饱和压力
118     def psat(self, T):
119         from scipy.optimize import fsolve
120
121         def objective(p):
122             return self.phi_l(T, p) - self.phi_g(T, p)
123
124         p_initial = 0.0001 # 初始猜测值, 单位MPa
125         psat_solution = fsolve(objective, p_initial)
126         return psat_solution[0] # 返回平衡压力, 单位MPa
127
128     # 绘制p-T相图
129     def plot_pT(self, T_min, T_max, savepath):
130         T_grid = np.linspace(T_min, T_max, 100)
131         p_grid = np.zeros_like(T_grid)
132         for i, T in enumerate(T_grid):
133             p_grid[i] = self.psat(T)
134
135         fig = plt.figure(figsize=(8, 6))
136         ax = fig.add_subplot(1, 1, 1)
137         ax.plot(T_grid, p_grid, "r-", linewidth=2)
138         ax.set_xlim(0, 1000)
139         ax.yaxis.set_major_locator(MultipleLocator(0.2))
140         ax.yaxis.set_minor_locator(MultipleLocator(0.1))
141         ax.set_xlabel(r"$T$ (K)")
142         ax.set_ylabel(r"$p$(MPa)")
143         ax.grid(True)
144
145         base_dir = os.path.dirname(os.path.abspath(__file__))
146         fig_dir = os.path.join(base_dir, "figs")
147         os.makedirs(fig_dir, exist_ok=True)

```

```

148     savepath = os.path.join(fig_dir, savepath)
149
150     fig.savefig(savepath, dpi=600, bbox_inches="tight", transparent=False)
151     plt.close(fig)
152
153
154 R290 = PR74(Tc=366.8, pc=4.248, omega=0.152)
155 R290.plot_pT(220, 325, "R290_pT.png")
156 R600a = PR74(Tc=407.8, pc=3.796, omega=0.227)
157 R600a.plot_pT(220, 325, "R600a_pT.png")
158 R1234yf = PR74(Tc=367.85, pc=3.3822, omega=0.276)
159 R1234yf.plot_pT(220, 325, "R1234yf_pT.png")
160 R1234ze = PR74(Tc=382.45, pc=3.6349, omega=0.313)
161 R1234ze.plot_pT(220, 325, "R1234ze(E)_pT.png")

```

7-5

按照书中图 7.22 的思路进行程序编写，绘制的 $p = 0.1\text{MPa}$ 和 $p = 1.0\text{MPa}$ 下溶液 R290/R600a 不同成分的泡点和露点温度的 T - x 相图如下所示：

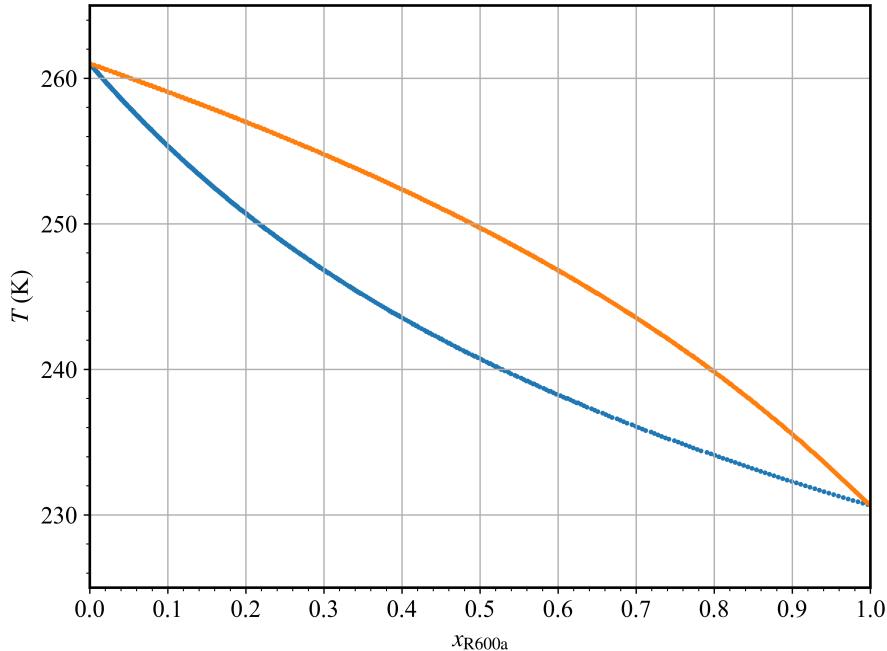


图 13: R290/R600a 在 0.1MPa 下的 T - x 相图

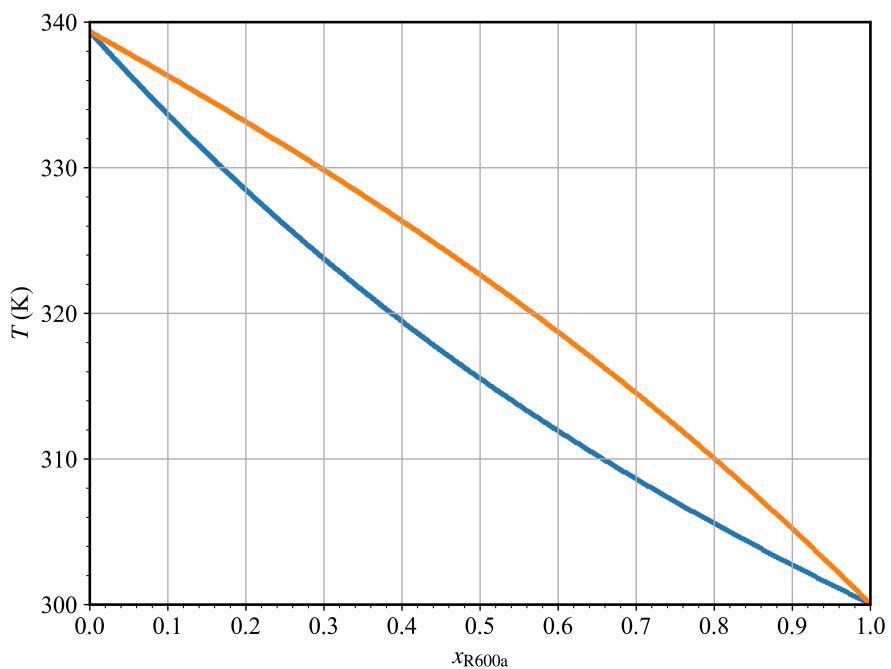


图 14: R290/R600a 在 1.0MPa 下的 T - x 相图

程序如下：

```

1 import os
2 import numpy as np
3 import matplotlib.pyplot as plt
4 from matplotlib.ticker import MultipleLocator
5
6 plt.rcParams["font.family"] = "serif"
7 plt.rcParams["font.serif"] = ["Times New Roman"]
8 plt.rcParams["mathtext.fontset"] = "stix"
9 plt.rcParams["font.size"] = 14
10 plt.rcParams["axes.linewidth"] = 1.5
11
12
13 class PR75:
14     def __init__(self, Tc1, pc1, omega1, Tc2, pc2, omega2, kij):
15         self.Tc1 = float(Tc1)
16         self.pc1 = float(pc1) * 1e6
17         self.omega1 = float(omega1)
18         self.Tc2 = float(Tc2)
19         self.pc2 = float(pc2) * 1e6
20         self.omega2 = float(omega2)
21         self.kij = float(kij)
22
23     R = 8.314462618 # J/(mol · K)
24
25     # 计算 a_i, b_i 以及混合 a, b; mu1 为该相中组分1摩尔分数
26     def params(self, T, mu1):

```

```

27     kappa1 = 0.37464 + 1.54226 * self.omega1 - 0.26992 * self.omega1**2
28     kappa2 = 0.37464 + 1.54226 * self.omega2 - 0.26992 * self.omega2**2
29     Tr1 = T / self.Tc1
30     Tr2 = T / self.Tc2
31     alpha1 = (1 + kappa1 * (1 - np.sqrt(Tr1))) ** 2
32     alpha2 = (1 + kappa2 * (1 - np.sqrt(Tr2))) ** 2
33     a1 = 0.45724 * (self.R * self.Tc1) ** 2 / self.pc1 * alpha1
34     a2 = 0.45724 * (self.R * self.Tc2) ** 2 / self.pc2 * alpha2
35     b1 = 0.07780 * self.R * self.Tc1 / self.pc1
36     b2 = 0.07780 * self.R * self.Tc2 / self.pc2
37     a = (
38         (mu1**2) * a1
39         + (1 - mu1) ** 2 * a2
40         + 2 * mu1 * (1 - mu1) * np.sqrt(a1 * a2) * (1 - self.kij)
41     )
42     b = mu1 * b1 + (1 - mu1) * b2
43     return a1, a2, a, b1, b2, b
44
45 # 计算 A 与 B
46 def AB(self, T, p, mu1):
47     a1, a2, a, b1, b2, b = self.params(T, mu1)
48     p_Pa = p * 1e6
49     A = a * p_Pa / (self.R**2 * T**2)
50     B = b * p_Pa / (self.R * T)
51     return A, B
52
53 # PR 三次的系数
54 def C(self, T, p, mu1):
55     A, B = self.AB(T, p, mu1)
56     C2 = B - 1.0
57     C1 = A - 3.0 * B**2 - 2.0 * B
58     C0 = -(A * B - B**2 - B**3)
59     return C2, C1, C0
60
61 # 液相 Z
62 def Zl(self, T, p, mu1):
63     C2, C1, C0 = self.C(T, p, mu1)
64     Z = 1.0e-3
65     for _ in range(10000):
66         f = Z**3 + C2 * Z**2 + C1 * Z + C0
67         df = 3.0 * Z**2 + 2.0 * C2 * Z + C1
68         Z_new = Z - f / df
69         if abs(Z_new - Z) < 1e-6:
70             Z = Z_new
71             break
72         Z = Z_new
73     return Z

```

```

74
75 # 气相 Z
76 def Zg(self, T, p, mu1):
77     C2, C1, C0 = self.C(T, p, mu1)
78     Z = 1.1
79     for _ in range(10000):
80         f = Z**3 + C2 * Z**2 + C1 * Z + C0
81         df = 3.0 * Z**2 + 2.0 * C2 * Z + C1
82         Z_new = Z - f / df
83         if abs(Z_new - Z) < 1e-6:
84             Z = Z_new
85             break
86     Z = Z_new
87     return Z
88
89 # 气相逸度系数
90 def phi_g(self, T, p, mu1):
91     Zg = self.Zg(T, p, mu1)
92     a1, a2, a, b1, b2, b = self.params(T, mu1)
93     A, B = self.AB(T, p, mu1)
94     # 组分1
95     lnphi1 = (
96         b1 / b * (Zg - 1.0)
97         - np.log(Zg - B)
98         - A
99         / (2.0 * np.sqrt(2.0) * B)
100        *
101            2.0 * ((1 - mu1) * (1 - self.kij) * np.sqrt(a1 * a2) + mu1 * a1) / a
102            - b1 / b
103        )
104        * np.log((Zg + 2.414 * B) / (Zg - 0.414 * B))
105    )
106    # 组分2
107    lnphi2 = (
108        b2 / b * (Zg - 1.0)
109        - np.log(Zg - B)
110        - A
111        / (2.0 * np.sqrt(2.0) * B)
112        *
113            2.0 * (mu1 * (1 - self.kij) * np.sqrt(a1 * a2) + (1 - mu1) * a2) / a
114            - b2 / b
115        )
116        * np.log((Zg + 2.414 * B) / (Zg - 0.414 * B))
117    )
118    return np.exp(lnphi1), np.exp(lnphi2)
119
120 # 液相逸度系数

```

```

121 def phi_1(self, T, p, mu1):
122     Zl = self.Zl(T, p, mu1)
123     a1, a2, a, b1, b2, b = self.params(T, mu1)
124     A, B = self.AB(T, p, mu1)
125
126     lnphi1 = (
127         b1 / b * (Zl - 1.0)
128         - np.log(Zl - B)
129         - A
130         / (2.0 * np.sqrt(2.0) * B)
131         *
132             2.0 * ((1 - mu1) * (1 - self.kij) * np.sqrt(a1 * a2) + mu1 * a1) / a
133             - b1 / b
134         )
135
136     * np.log((Zl + 2.414 * B) / (Zl - 0.414 * B))
137
138 )
139
140     lnphi2 = (
141         b2 / b * (Zl - 1.0)
142         - np.log(Zl - B)
143         - A
144         / (2.0 * np.sqrt(2.0) * B)
145         *
146             2.0 * (mu1 * (1 - self.kij) * np.sqrt(a1 * a2) + (1 - mu1) * a2) / a
147             - b2 / b
148         )
149
150     * np.log((Zl + 2.414 * B) / (Zl - 0.414 * B))
151
152     return np.exp(lnphi1), np.exp(lnphi2)
153
154
155 def plot_Tx(self, p, T0):
156     T_list = []
157     x_bub_list = [] # 泡点: x1 vs T
158     y_dew_list = [] # 露点: y1 vs T
159
160     # y1 扫描
161     y1_values = np.linspace(0.0, 1.0, 1001)
162
163     for y1 in y1_values:
164         y1 = float(y1)
165         y2 = 1.0 - y1
166
167         # 初值: 液相 x 猜 0.1/0.9; T 从 T0 逐步增加
168         T = float(T0)
169         x1 = 0.1
170         s = 0.0 # s = Σ k_i y_i

```

```

168     # 外层调温: 使 s → 1 (露点判据)
169     # 注: 气相侧 ^v 用 y1; 液相侧 ^l 用 x1
170     it_guard = 0
171     while abs(s - 1.0) >= 1e-3:
172         T += 0.1
173         phi_g1, phi_g2 = self.phi_g(T, p, y1) # 气相
174         phi_l1, phi_l2 = self.phi_l(T, p, x1) # 液相
175
176         k1 = phi_g1 / phi_l1 # = 1/K1
177         k2 = phi_g2 / phi_l2 # = 1/K2
178
179         denom = k1 * y1 + k2 * y2
180         if denom <= 1e-16:
181             break
182
183         x1 = (k1 * y1) / denom
184         s_prev = s
185         s = denom
186
187         # 细化循环: 仅重算液相侧
188         inner_guard = 0
189         while abs(s - s_prev) > 1e-6:
190             phi_l1, phi_l2 = self.phi_l(T, p, x1)
191             k1 = phi_g1 / phi_l1
192             k2 = phi_g2 / phi_l2
193             denom = k1 * y1 + k2 * y2
194             if denom <= 1e-16:
195                 break
196             x1 = (k1 * y1) / denom
197             s_prev = s
198             s = denom
199
200             inner_guard += 1
201             if inner_guard > 2000: # 防止极端情况
202                 break
203
204             it_guard += 1
205             if it_guard > 20000: # 防止极端情况
206                 break
207
208         # 收集边界点
209         T_list.append(T)
210         x_bub_list.append(x1) # 泡点边界 (液相组成)
211         y_dew_list.append(y1) # 露点边界 (气相组成)
212
213         fig, ax = plt.subplots(1, figsize=(8, 6))
214         ax.scatter(x_bub_list, T_list, s=3, label=r"bubble: $T$ - $x$")

```

```

215     ax.scatter(y_dew_list, T_list, s=3, label=r"dew: $T\$ - $y\$")
216     ax.set_xlim(0.0, 1.0)
217     ax.xaxis.set_major_locator(MultipleLocator(0.1))
218     ax.xaxis.set_minor_locator(MultipleLocator(0.02))
219     ax.yaxis.set_major_locator(MultipleLocator(10))
220     ax.yaxis.set_minor_locator(MultipleLocator(2))
221     ax.grid(True)
222
223     ax.set_xlabel(r"$x_{\mathrm{R600a}}$")
224     ax.set_ylabel(r"$T$ (K)")
225
226     base_dir = os.path.dirname(os.path.abspath(__file__))
227     fig_dir = os.path.join(base_dir, "figs")
228     os.makedirs(fig_dir, exist_ok=True)
229     filename = f"{p:.3f}MPa.png"
230     savepath = os.path.join(fig_dir, filename)
231     fig.savefig(savepath, dpi=600, bbox_inches="tight", transparent=False)
232     plt.close(fig)
233
234
235 R290R600a = PR75(
236     Tc1=369.89,
237     pc1=4.2512,
238     omega1=0.1521, # R290
239     Tc2=407.81,
240     pc2=3.629,
241     omega2=0.184, # R600a
242     kij=0.01,
243 )
244 R290R600a.plot_Tx(p=0.1, T0=215.0)
245 R290R600a.plot_Tx(p=1.0, T0=290.0)

```