丁基橡胶聚合反应器机理模型开发说明文档

2025 年 4 月 13 日

目录

一、依赖环境说明 3
二、API架构 4
三、API说明 4
1、 Component 类 4
2、 Polymer 类 （父类：Component） 5
3、 Segment 类（父类：Component） 5
4、 ComponentManager 类 6
5、 Flow 类 7
6、 CSTRSingleLiqPhase 类 9
7、 PFRSingleLiqPhase 类 11
8、 Mixer 类 14
9、 Spliter类 16
10、ReactionSet 类 17
11、基类：PropertiesMethod 20
12、DeconvolutionHelper 类 22
13、Utility.py 24
四、 使用流程说明 28
（1）全局组分定义 28
（2）反应集定义 32
（3）定义反应源项（source difinition） 36
（4）定义反应器模型 Block 38

一、依赖环境说明

（1）开发环境：Python 3.8.0

（2）主要依赖：

软件包及其版本信息表

包名	版本号	说明
matplotlib	3.7.5	用于创建静态、动态或交互式的可视化图表
multidict	6.1.0	实现多值字典数据结构，允许一个键对应多个值
numpy	1.24.4	提供多维数组运算、线性代数运算
pandas	2.0.3	数据分析和处理库，提供 DataFrame 等数据结构 依赖 NumPy
pillow	10.4.0	Python 图像处理库
ply	3.11	Python Lex-Yacc 实现-用于 Pyomo 的解析功能
Pyomo	6.8.0	优化建模语言 支持线性规划、非线性规划等优化问题- 依赖 ply 和 six
scipy	1.10.1	科学计算工具包提供数值积分、优化、线性代数等 功能，依赖 NumPy
six	1.16.0	Python 2 和 3 的兼容库 用于确保代码在不同 Python 版本上运行
Pcsaft *		（自编 pcsaft 物性计算模型包，需添加引用）

备注：除了安装 Pyomo 外，要求解模型方程，还需要安装 IPOPT 求解器：
GitHub - coin-or/Ipopt: COIN-OR Interior Point Optimizer IPOPT

二、API 架构

为适应之后各种项目和反应机理需要，采取模块化设计，除了作为主函数入口的 main.py 外，其余功能都被封装成对应的 python 类文件

文件名称	类名	说明
Component.py	Component 物质组分类	提供对于普通物质组分、聚合物、链段的实例化对象的数据结构
ComponentManager.pt	GlobalComponentManager 全局组分管理类	提供对全局组分的管理和初始化
Flow.py	Flow 流股类	流股对象的数据结构
Reactor.py	Reactor 反应器类	提供两种基础单相反应器模型类：CstrSingleLiqPhase 和 PFRSingleLiqPhase
OperationUnit.py	OperationUnit 单元操作类	混合器和分流器
Reactions.py	ReactionSet 反应集类	对反应机理进行指定，计算反应速率
Propertiesmethod.py	PropertiesMethod 物性管理类	提供不同物性计算模型的抽象类
Deconvolution.py	DeconvolutionHelper 分峰工具类	提供对分子量分布进行混合高斯分峰
Utility.py	Utility 公用功能类	对反应器求解结果的聚合物相关性质如数均分子量等进行后处理

三、API 说明

注：（派生类中父类的成员变量不再列出）

1、Component 类

成员变量	说明
Name	组分名称
CAS	CAS 号
Type	组分类型，用于区分普通物质（0）、聚合物（1）、链段（2）
Polymer_mole_flow_relative	是否参与混合物流股摩尔流量计算以及形式标识符
MW	分子量

构造函数参数	说明
formular	即组分名称
cas	CAS 号

type	组分类型枚举类 CompType
comp_mw	组分分子量

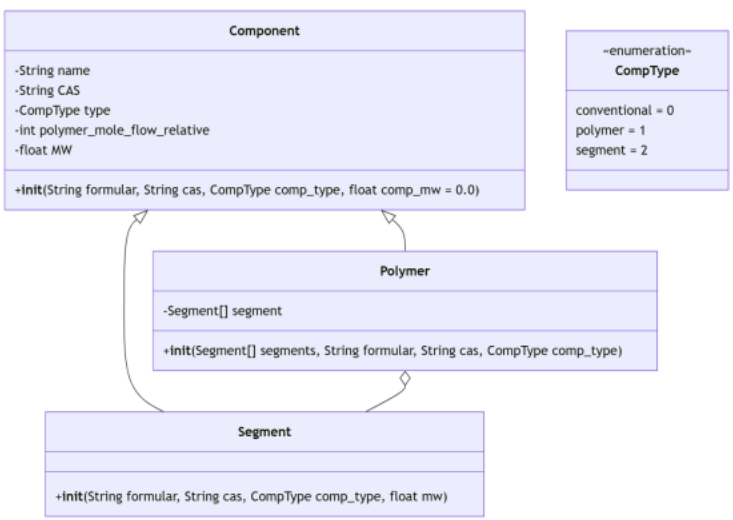
2、Polymer 类 （父类：Component）

成员变量	说明
Segment	聚合物包含的链段对象

构造函数参数	说明
segments	链段对象
formular	组分名称
cas	CAS 号
Comptype	组分类型 默认参数：CompType.polymer

3、Segment 类（父类：Component）

构造函数参数	说明
formular	组分名称
cas	CAS 号
Comptype	组分类型
Mw	链段分子量



4、ComponentManager 类

成员变量	类型	说明
component_list	List[Component]	存储所有组分对象的列表，包括常规组分和特定组分

方法名	参数	参数说明	功能描述
component_list_gen	clist	List[Component]	输入的组分列表，包含常规组分和聚合物组分
	poly_Type	str	ReactConfig.csv 中的列名，用于读取聚合物配置信息
	poly_site	int	聚合物位点数量，用于生成位点相关的特定组分

方法功能详细说明

component_list_gen 方法的主要功能：

（1）处理常规组分：

遍历输入的组分列表
将类型为 CompType.conventional 的组分添加到 component_list

（2）处理聚合物组分：

从 'ReactConfig.csv' 文件读取聚合物配置信息
根据配置信息生成特定组分
支持两种特殊标记：
如果在 ReactConfig.csv 文件中某一组分涉及多活性位，则其配置信息会以如下形式进行表示：

组分名称+“；”+NSite

第三个分号后的数字：表示是否设计流股摩尔流量计算，如果涉及，应该属于哪一阶矩

例如，在阳离子聚合中，以 A 共聚单体结尾的聚合物活性链的 1 阶矩是一个依赖活性位点，且在流股混合物摩尔流量计算中，需以 1 阶矩量进行计算的摩尔量，

那么其在 ReactConifg.csv 中的表示应当为：

First_mom_live_A;NSite;1

（3）特定组分生成规则：

位点依赖的组分：名称格式为 "原始名称[位点编号]"
支持摩尔流量标识符设置，默认值为-1

ReactConfig.csv 文件结构

```
CATION
ion-pair;NSite
p1_ion_A;NSite
p1_ion_B;NSite
zeroth_mom_live_A;NSite;0
zeroth_mom_live_B;NSite;0
first_mom_live_A;NSite;1
first_mom_live_B;NSite;1
second_mom_live_A;NSite;2
second_mom_live_B;NSite;2
zeroth_mom_dead;NSite;0
first_mom_dead;NSite;1
second_mom_dead;NSite;2
counter_ion
```

5、Flow 类

成员变量	类型	说明
Temperature	float	流股温度
Pressure	float	流股压力
Name	str	流股名称
comp_dict	Dict	组分字典，存储各组分的质量流量、摩尔流量和聚合物流动矩量标识符

comp_dict 结构说明

```
{
  "组分名称": {
    "mass_flow": float,     # 质量流量
    "mole_flow": float,     # 摩尔流量
    "polymer_flow_momentum": int  # 聚合物质量标识符
  }
}
```

构造函数说明

参数名	类型	说明
t	float	初始化温度
p	float	初始化压力
name	str	流股名称

方法说明

方法名	参数	功能描述
get_mole_frac	无	计算并打印总摩尔流量
set_MassFlow_conventional	flow_arr: Dict[str, float]	设置常规组分的质量流量，并自动转换为摩尔流量
mass_2_mole	无	将质量流量转换为摩尔流量

方法详细说明

get_mole_frac

收集所有组分的摩尔流量
计算总摩尔流量并打印

set_MassFlow_conventional

参数 flow_arr 为字典，键为组分名称，值为质量流量
检查输入的组分是否存在于组分字典中
设置对应组分的质量流量

调用 mass_2_mole 进行质量流量到摩尔流量的转换

mass_2_mole

遍历组分字典中的所有组分
仅转换 polymer_flow_momentum 为 -2 的组分
转换公式：摩尔流量 = 质量流量 / 分子量
其他组分的摩尔流量设为 0.0

特殊说明

- comp_dict 在初始化时会为每个组分创建一个包含三个键值对的字典结构
- polymer_flow_momentum 值为 -2 表示参与摩尔流量计算的常规组分
- 质量流量到摩尔流量的转换仅对常规组分进行

6、CSTRSingleLiqPhase 类

成员变量	变量类型	说明
ReactionSet	ReactionSet	反应集合对象
Inflow	Flow	入口流股对象
PropertiesMethod	PropertiesMethod	物性计算方法对象
Temperature	float	反应器温度
Pressure	float	反应器压力
Volume	float	反应器体积
q_spec	float	指定的体积流量（默认为 0）

构造函数说明

参数名	类型	默认值	说明
t	float	-	反应器温度
p	float	-	反应器压力
v	float	-	反应器体积
inflow	Flow	-	入口流股对象
rx_set	ReactionSet	-	反应集合对象

参数名	类型	默认值	说明
prop	PropertiesMethod	-	物性计算方法对象
q_spec	float	0	指定的体积流量

方法说明

方法名	参数	返回值	功能描述
compute_dpn	model	float	计算数均聚合度(DPN)
volume_flow_rate_rule	model	float	计算体积流量
mass_balance	model	List[Equation]	建立组分质量平衡方程组

方法详细说明

compute_dpn

- 计算聚合物的数均聚合度
- 计算逻辑：
- 计算聚合物零阶矩（polymer_flow_momentum = 0）
- 计算聚合物一阶矩（polymer_flow_momentum = 1）
- 计算总体零阶矩和一阶矩
- 返回一阶矩与零阶矩的比值

volume_flow_rate_rule

- 计算反应器出口体积流量
- 处理流程：
  - 如果指定了 **q_spec**，直接返回指定值
  - 否则基于以下步骤计算：
    - 计算聚合物零阶矩流量
    - 计算总体零阶和一阶矩流量
    - 计算物性参数
    - 计算混合物摩尔体积
    - 返回体积流量

mass_balance

- 建立反应器的组分质量平衡方程组
- 方程组包含：
  - 入口物料流量

    - 出口物料流量
    - 反应生成/消耗量
- 计算步骤：
    - 计算各组分浓度
    - 建立质量平衡方程
    - 返回方程组列表

7、PFRSingleLiqPhase 类

成员变量名	类型	说明
ReactionSet	ReactionSet	反应集合对象
Inflow	Flow	入口流股对象
PropertiesMethod	PropertiesMethod	物性计算方法对象
Temperature	float	反应器温度
Pressure	float	反应器压力
Area	float	反应器截面积（计算自直径和管数）
Length	float	反应器长度
dH_rx	float	反应热
Diameter	float	管道直径
U	float	传热系数
T_cool_in	float	冷却剂入口温度
T_cool_out	float	冷却剂出口温度
T_cool	function	冷却剂温度分布函数
qspec	float	指定的体积流量
tubes	int	管道数量

构造函数说明

参数名	类型	默认值	说明
t	float	-	反应器温度

参数名	类型	默认值	说明
p	float	-	反应器压力
l	float	-	反应器长度
D	float	-	特征尺寸
inflow	Flow	-	入口流股对象
rx_set	ReactionSet	-	反应集合对象
prop	PropertiesMethod	-	物性计算方法对象
U	float	-	传热系数
dH_rx	float	-	反应热
T_cool_in	float	-	冷却剂入口温度
T_cool_out	float	-	冷却剂出口温度
diameter	float	-	管道直径
multitubes	int	-	管道数量
qspec	float	0.0	指定的体积流量

方法说明

方法名	参数	返回值	功能描述
initialize_temperature_calculation	无	无	初始化冷却剂温度分布函数
compute_dpn	model, z	float	计算指定位置的数均聚合度
volume_flow_rate_rule	model, z	float	计算指定位置的体积流量
concentration_rule	model, comp, z	float	计算指定组分在指定位置的浓度
cp_mix_rule	model, z	float	计算指定位置的混合物比热容
mass_balance	model,	Equation	建立组分质

方法名	参数	返回值	功能描述
	comp, z		量平衡方程
heat_balance	model, z	Equation	建立热量平衡方程

方法详细说明

initialize_temperature_calculation

- 初始化冷却剂温度分布函数
- 冷流股建立沿程温度线性分布模型：T(z) = T_in + (T_out - T_in) * (z/L)

compute_dpn

- 计算特定位置的数均聚合度
- 计算步骤：
  - 计算聚合物零阶矩流量
  - 计算聚合物一阶矩流量
  - 返回一阶矩与零阶矩的比值（数均聚合度）

volume_flow_rate_rule

- 计算特定位置的体积流量
- 处理流程：
  - 检查是否有指定流量
  - 计算物性参数
  - 计算混合物摩尔体积
  - 返回体积流量

concentration_rule

- 计算特定组分在指定位置的浓度
- 浓度 = 摩尔流量/体积流量

cp_mix_rule

- 计算混合物比热容

考虑所有组分（常规组分和聚合物）的贡献

mass_balance

- 建立组分质量平衡微分方程

- 考虑反应动力学
- 方程形式：dF/dz = rate * Area

heat_balance

- 建立热量平衡微分方程
- 考虑：
  - 反应热效应（Q_rx）
  - 传热效应（Q_transfer）
  - 混合物比热容 cp_mix
  - 方程形式：dT/dz = (Q_rx + Q_transfer)/cp_mix

特殊说明

- 反应器模型考虑了管式反应器的特征（多管、传热等）
- 冷流股温度分布采用线性模型
- 质量平衡和热量平衡采用微分方程形式
- 物性计算考虑了聚合物特性
- 时间单位统一转换为小时（*3600）

8、Mixer 类

成员变量说明

变量名	类型	说明
split_Out_flow	Flow	混合器出口流股对象
inlet_flow_list	List[Flow]	混合器入口流股列表

构造函数说明

参数名	类型	说明
inlet_flow_list	List[Flow]	需要混合的入口流股列表

构造函数详细说明

- 例如创建一个新的出口流股对象，初始条件：
  - 温度：101
  - 压力：101

    - 名称：'mix'
- 存储入口流股列表供后续计算使用

方法说明

方法名	参数	返回值	功能描述
mass_balance	无	无	执行混合器的质量平衡计算

mass_balance 方法详细说明

1、功能描述

- 计算所有入口流股的混合结果
- 更新出口流股的组分摩尔流量

2、计算流程

- 遍历出口流股中的每个组分
- 对于每个组分：
- 初始化混合流量为 0
- 遍历所有入口流股
- 累加每个入口流股中该组分的摩尔流量
- 将累加结果赋值给出口流股对应组分的摩尔流量

3、计算特点

- 基于摩尔流量进行混合计算
- 保持组分列表不变
- 简单加和计算，不考虑混合效应

特殊说明

1、假设条件

- 假设理想混合（无热效应）
- 出口压力和温度为固定值
- 仅考虑摩尔流量的混合

2、限制条件

- 不考虑温度变化
- 不考虑压力变化
- 不考虑混合焓变
- 不考虑组分间的相互作用

3、注意事项

- 所有入口流股必须具有相同的组分列表
- 混合计算仅基于摩尔流量
- 出口流股的温度和压力是固定的，不受入口条件影响

9、Spliter类

变量名	类型	说明
split_Out_flow_list	List[Flow]	分流器出口流股列表
split_frac	List[float]	各出口流股的分流比例列表
inlet_flow	Flow	分流器入口流股对象

构造函数说明

参数名	类型	说明
inlet_flow	Flow	需要分流的入口流股
split_frac	List[float]	各出口的分流比例列表

构造函数详细说明

- 根据分流比例列表的长度创建相应数量的出口流股
- 每个出口流股的初始条件：
  - 温度：101
  - 压力：0
  - 名称：'split_flow_out_' + 序号
- 存储入口流股和分流比例供后续计算使用

方法说明

方法名	参数	返回值	功能描述

方法名	参数	返回值	功能描述
mass_balance	无	无	执行分流器的质量平衡计算

mass_balance 方法详细说明

1、功能描述

- 根据指定的分流比例计算各出口流股的组分流量
- 更新所有出口流股的组分摩尔流量

2、计算流程

- 遍历所有出口流股
- 对于每个出口流股：
  - 遍历所有组分
  - 计算该出口流股中每个组分的摩尔流量：
    - 出口流量 = 入口流量 × 分流比例
  - 更新出口流股中对应组分的摩尔流量

使用示例

```
# 创建入口流股
inlet_flow = Flow(t=101, p=101, name='inlet')
# 定义分流比例（例如：0.3, 0.7 的两路分流）
split_ratios = [0.3, 0.7]
# 创建分流器对象
splitter = Spliter(inlet_flow, split_ratios)
# 执行分流计算
splitter.mass_balance()
# 获取分流后的流股列表
split_flows = splitter.split_Out_flow_list
```

10、ReactionSet 类

成员变量说明

变量名	类型	说明
source_dict	Dict[str, List]	存储每个组分的反应源项信息
ea	Dict[str, float]	存储反应活化能信息

构造函数说明

- 初始化 source_dict：为每个组分创建空列表
- 初始化 ea 为 None

方法说明

方法名	参数	返回值	功能描述
output_rx_matrix	无	无	生成反应矩阵
print_kinetics	无	Dict[str, List[str]]	生成并返回反应动力学表达式
source_define	idx_obj, powerlaw_idx, k_constant, param=None, order=None, is_sink=True	无	定义反应源项
calculate_rate	comp_key, concen, T	float	计算特定组分的反应速率
preview_reaction_equations	无	无	预览反应方程式
kinetics_temperature_correction	K_orin, Ea, T, Tr	Dict	计算温度

方法详细说明

output_rx_matrix

- 生成反应矩阵，表示各组分在反应中的系数
- 规则：
  - -1 表示反应物
  - 1 表示产物
  - 0 表示不参与反应

print_kinetics

- 生成反应动力学表达式
- 返回格式：{组分：[反应表达式列表]}
- 表达式示例：-[A]*[B] 表示 A 和 B 反应

source_define

- 参数说明：
  - idx_obj: 目标组分索引
  - powerlaw_idx: 幂律指数索引
  - k_constant: 反应速率常数信息
  - param: 参与因子 [默认：[1,1]]
  - order: 反应级数 [默认：[1,1]]
  - is_sink: 是否为消耗项 [默认：True]
- 功能：定义组分的反应源项信息

calculate_rate

- 参数说明：
  - comp_key: 组分键名
  - concen: 浓度列表
  - T: 温度
- 功能：计算特定组分的反应速率
- 计算公式：

$$\text{rate} = K * e^{-\frac{Ea}{R} * (\frac{1}{T} - \frac{1}{Tr})} * (c1\text{^}order1) * param1 * (c2\text{^}order2) * param2$$

preview_reaction_equations

- 打印所有反应方程式
- 格式：r[组分]=±k*[组分 1]*[组分 2]

kinetics_temperature_correction

- 参数说明：

- K_orin: 原始反应速率常数
- Ea: 活化能
- T: 目标温度
- Tr: 参考温度

- 功能：根据阿伦尼乌斯方程计算温度修正后的反应速率常数

特殊说明

- 反应源项结构

```
source = [
    powerlaw_idx[0],     # 第一个反应物索引
    powerlaw_idx[1],     # 第二个反应物索引
    param[0],           # 第一个参与因子
    param[1],           # 第二个参与因子
    order[0],           # 第一个反应级数
    order[1],           # 第二个反应级数
    direction,          # 反应方向（1/-1）
    k_constant['value'],# 速率常数值
    k_constant['name'] # 速率常数名称
]
```

11、基类：PropertiesMethod（ABC）

抽象方法

方法名	参数	返回值	说明
calculate_molar_density_mixture	*args, **kwargs	abstract	计算混合物的摩尔密度
calculate_molar_density_pure	*args, **kwargs	abstract	计算纯组分的摩尔密度
calculate_polymer_density	*args, **kwargs	abstract	计算聚合物密度
calculate_mixture_cp	*args, **kwargs	abstract	计算混合物比热容

子类 1：UserMethod

实现的方法

calculate_molar_density_mixture

- 参数：
  - mole_frac: 摩尔分数
  - temperature: 温度
  - pressure: 压力
  - param_list: 参数列表（包含 Tc, Pc, Omega）
- 返回值：(densities, Z_real)
  - densities: 可能的密度值
  - Z_real: 对应的压缩因子

实现说明

- 使用 Peng-Robinson 状态方程
  - 计算步骤：
  - 计算 α 和临界参数
  - 计算混合规则参数
  - 求解三次方程得到压缩因子
  - 计算对应的密度值

calculate_molar_density_pure (未实现)

calculate_polymer_density (未实现)

子类 2：IIR_PCSAFT

实现的方法

calculate_molar_density_mixture

- 参数：
  - temperature: 温度
  - pressure: 压力
  - param: PC-SAFT 参数
  - mole_frac: 摩尔分数
  - polymer_mole_frac_zeroth: 聚合物零阶矩摩尔分数
  - dpn: 数均聚合度
- 返回值：修正后的密度值
- 实现说明：
  - 使用 PC-SAFT 状态方程计算基础密度
  - 应用聚合物密度修正

calculate_mixture_cp

- 参数：
  - temperature: 温度
  - param: 参数对象
  - mole_frac: 摩尔分数
  - 返回值：混合物比热容

- 实现说明：
  - 根据不同组分类型使用不同的比热容计算方法：
    - CPLID='100': 使用 dip100 方法
    - CPLID='506': 使用 dipted506 方法
    - 其他：使用 Bicerano 方法计算固体比热容
  - 使用摩尔分数加权平均得到混合物比热容

使用示例

```
# 使用 UserMethod
user_method = UserMethod()
params = {
    "Tc": 500.0,
    "Pc": 30.0e5,
    "Omega": 0.3
}
densities, Z = user_method.calculate_molar_density_mixture(
    mole_frac=[0.5, 0.5],
    temperature=300,
    pressure=1e5,
    param_list=params
)
# 使用 IIR_PCSAFT
pcsaft_method = IIR_PCSAFT(param=some_params)
density = pcsaft_method.calculate_molar_density_mixture(
    temperature=300,
    pressure=1e5,
    param=some_params,
    mole_frac=[0.3, 0.7],
    polymer_mole_frac_zeroth=0.3,
    dpn=100
)
```

12、DeconvolutionHelper 类

成员变量

变量名	类型	说明	默认值
site_num	int	解卷积峰的数量	3
method	str	解卷积使用的方法	'Gaussian'
MW	numpy.ndarray	分子量数据数组	None
Y	numpy.ndarray	分布数据数组	None

构造函数

参数名	类型	说明	默认值	是否必需
site_num	int	设置解卷积峰的数量	3	否
method	str	设置解卷积方法	'Gaussian'	否

公共方法

load_mwd(path)

- 加载分子量分布数据。
- 参数:
  - path (str): CSV 文件路径
- 说明:
  - 从 CSV 文件中读取分子量分布数据
  - 第二列为分子量数据
  - 最后一列为分布数据

plot_mwd()

- 绘制原始分子量分布图。
- 说明:
  - X 轴：分子量的对数值
  - Y 轴：分布值
- 使用 matplotlib 进行绘制

deconvolution()

- 执行解卷积计算

- 功能:
  - 创建优化模型
  - 设置初始参数值：

```
initial_guess = {
    'A0': max(self.Y)/2, 'A1': max(self.Y)/3, 'A2':
max(self.Y)/3,
    'mu0': 4.7, 'mu1': 1, 'mu2': 1,
    's0': 0.1, 's1': 0.01, 's2': 0.1
}
```

  - 添加约束条件：
    - 面积非负：$A[i] \geq 0$
    - 面积归一化：$\Sigma A[i] = 1$
    - 峰宽度约束：$10^{(2s[i]^2)} = 2.0$
    - 面积上限：$A[i] \leq 1.0$
    - 位置范围：$0 \leq \mu[i] \leq max(MW)$
- 求解优化问题并可视化结果

13、Utility.py

成员变量

变量名	类型	说明	默认值
species_list	list	流出物种列表	None
IB_consume	float	异丁烯消耗量	None
IP_consume	float	异戊二烯消耗量	None
zeroth_mom_live_tot	float	总活性链零阶矩	None
first_mom_live_tot	float	总活性链一阶矩	None
second_mom_live_tot	float	总活性链二阶矩	None
zeroth_mom_dead_tot	float	总死链零阶矩	None
first_mom_dead_tot	float	总死链一阶矩	None
second_mom_dead_tot	float	总死链二阶矩	None

函数参数

参数名	类型	说明	是否必需

solutions	object	反应器解决方案对象	是
flow_IN	object	入口流对象	是
polymer_meta	object	聚合物元数据对象	是

函数说明

计算聚合物反应器中的各项性质参数。主要包括：

1. 矩计算

- 零阶矩（ZMOM）：表示聚合物链数
- 一阶矩（FMOM）：表示聚合物总单体数
- 二阶矩（SMOM）：用于计算分子量分布

2. 分子量计算

```
MWN = DPN * segment_MW  # 数均分子量
MWW = DPW * segment_MW  # 重均分子量
PDI = MWW / MWN         # 分散度
```

3. 返回值说明（polymer_prop 字典）

主要属性包括：

- 总体性质：
  - ZMOM, FMOM, SMOM：总矩
  - MWN, MWW, PDI：分子量特征
  - DPN, DPW：聚合度
- 活性链性质（L 前缀）：
  - LZMOM, LFMOM, LSMOM
- 死链性质（D 前缀）：
  - DZMOM, DFMOM, DSMOM
- 分段组成（SFRAC）：
  - IB-seg：异丁烯段分数
  - IP-seg：异戊二烯段分数
- 各活性位点性质（S 前缀）：
  - SZMOM, SFMOM, SSMOM：各位点矩
  - SMWN, SMWW, SPDI：各位点分子量特征
  - SPFRAC：各位点产量分数

使用示例

```
# 创建反应器解决方案
solutions = reactor.solve()
# 计算聚合物性质
polymer_properties = calculate_polymer_properties_reactor(
    solutions,
    flow_IN,
    polymer_meta
)
# 获取数均分子量
Mn = polymer_properties['MWN']
```

Polymer Properties 计算项说明

属性名	计算公式	物理含义	单位	说明
ZMOM	zeroth_mom_live_tot + zeroth_mom_dead_tot	总零阶矩	mol	表示聚合物链总数
FMOM	first_mom_live_tot + first_mom_dead_tot	总一阶矩	mol	表示聚合物总单体数
SMOM	second_mom_live_tot + second_mom_dead_tot	总二阶矩	mol	用于计算分子量分布
LZMOM	Σ(zeroth_mom_live_A[i] + zeroth_mom_live_B[i])	活性链零阶矩	mol	活性聚合物链数
LFMOM	Σ(first_mom_live_A[i] + first_mom_live_B[i])	活性链一阶矩	mol	活性链单体数
LSMOM	Σ(second_mom_live_A[i] + second_mom_live_B[i])	活性链二阶矩	mol	活性链二阶矩
DZMOM	Σ(zeroth_mom_dead[i])	死链零阶矩	mol	死链聚合

属性名	计算公式	物理含义	单位	说明
				物链数
DFMOM	Σ(first_mom_dead[i])	死链一阶矩	mol	死链单体数
DSMOM	Σ(second_mom_dead[i])	死链二阶矩	mol	死链二阶矩
SZMOM[i]	zeroth_mom_live_A[i] + zeroth_mom_live_B[i] + zeroth_mom_dead[i]	位点 i 的总零阶矩	mol	各位点链总数
SFMOM[i]	first_mom_live_A[i] + first_mom_live_B[i] + first_mom_dead[i]	位点 i 的总一阶矩	mol	各位点单体总数
SSMOM[i]	second_mom_live_A[i] + second_mom_live_B[i] + second_mom_dead[i]	位点 i 的总二阶矩	mol	各位点二阶矩
LSZMOM[i]	zeroth_mom_live_A[i] + zeroth_mom_live_B[i]	位点 i 的活性链零阶矩	mol	各位点活性链数
LSFMOM[i]	first_mom_live_A[i] + first_mom_live_B[i]	位点 i 的活性链一阶矩	mol	各位点活性链单体数
LSSMOM[i]	second_mom_live_A[i] + second_mom_live_B[i]	位点 i 的活性链二阶矩	mol	各位点活性链二阶矩
SFRAC['IB-seg']	IB_consume / (IB_consume + IP_consume)	异丁烯段分数	-	异丁烯组分比例
SFRAC['IP-seg']	IP_consume / (IB_consume + IP_consume)	异戊二烯段分数	-	异戊二烯

属性名	计算公式	物理含义	单位	说明
				组分比例
DPN	first_mom_tot / zeroth_mom_tot	数均聚合度	-	平均链长
DPW	second_mom_tot / first_mom_tot	重均聚合度	-	重均链长
SDPN[i]	SFMOM[i] / SZMOM[i]	位点 i 的数均聚合度	-	各位点平均链长
SDPW[i]	SSMOM[i] / SFMOM[i]	位点 i 的重均聚合度	-	各位点重均链长
MWN	DPN * segment_MW	数均分子量	g/mol	平均分子量
MWW	DPW * segment_MW	重均分子量	g/mol	重均分子量
PDI	MWW / MWN	分散度指数	-	分子量分布宽度
SMWN[i]	SDPN[i] * segment_MW	位点 i 的数均分子量	g/mol	各位点平均分子量
SMWW[i]	SDPW[i] * segment_MW	位点 i 的重均分子量	g/mol	各位点重均分子量
SPDI[i]	SMWW[i] / SMWN[i]	位点 i 的分散度	-	各位点分子量分布
SPFRAC[i]	SFMOM[i] / Σ(SFMOM)	位点 i 的产量分数		

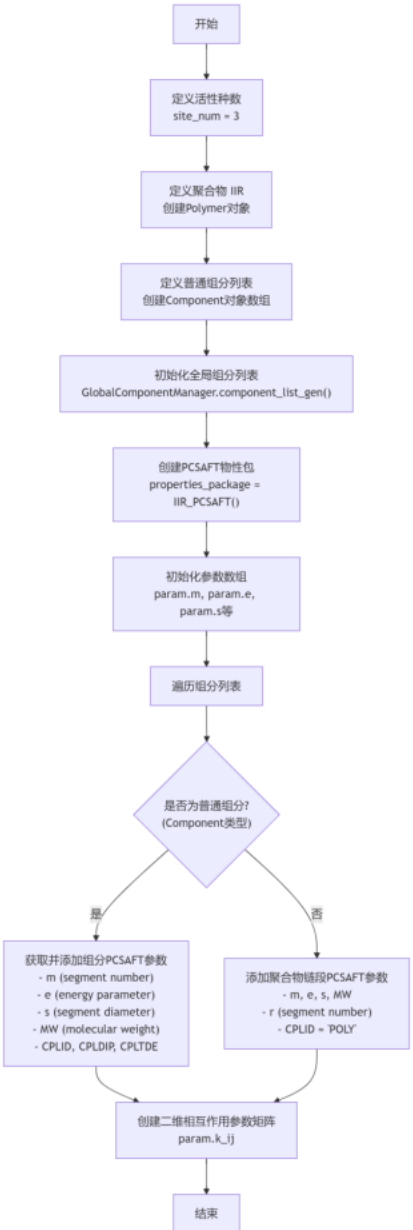
四、使用流程说明

（1）全局组分定义

```
# 定义活性种数
site_num = 3

# 定义聚合物
IIR = Polymer([Segment('IB-R', 'IB-R-seg', CompType.segment, 56.1075)], 'IIR',
              'IIR', CompType.polymer)
# 定义普通组分
component_list = [Component("IB", '115-11-7', CompType.conventional, 56.1075),
                  Component("IP", '78-79-5', CompType.conventional, 68.1185),
                  Component("HCL", '7647-01-0', CompType.conventional, 36.4606),
                  Component("EADC", '563-43-9', CompType.conventional, 126.949),
                  Component("HEXANE", '110-54-3', CompType.conventional, 86.1772),
                  Component("CH3CL",'74-87-3', CompType.conventional, 50.4875), IIR]

# 调用 GlobalComponentManager.component_list_gen()函数，初始化全局组分列表
componentmanager.GlobalComponentManager.component_list_gen(component_list, 'CATION',
site_num)

# 装填 PCSAFT 物性参数
param = Param
properties_package = IIR_PCSAFT(param)
param.m = np.array([])
param.e = np.array([])
param.s = np.array([])
param.r = np.array([])
param.MW = np.array([])
param.CPLID = np.array([])
param.CPLDIP = np.empty((0, 7))
param.CPLTDE = np.empty((0, 9))
for c in GlobalComponentManager.component_list:

    if type(c) is Component:
        param.m = np.append(param.m,
np.float32(properties_package.pcsaft.retrive_param_from_DB(c.CAS, 'PCFTM')))
        param.e = np.append(param.e,
np.float32(properties_package.pcsaft.retrive_param_from_DB(c.CAS, 'PCFTU')))
        param.s = np.append(param.s,
np.float32(properties_package.pcsaft.retrive_param_from_DB(c.CAS, 'PCFTV')))
        param.MW = np.append(param.MW,
```

```
np.float32(properties_package.pcsaft.retrive_param_from_DB(c.CAS, 'MW')))
        param.CPLID = np.append(param.CPLID,
properties_package.pcsaft.retrive_param_from_DB(c.CAS, 'CPLID'))
        param.CPLDIP = np.vstack([param.CPLDIP,

[np.float32(properties_package.pcsaft.retrive_param_from_DB(c.CAS, 'CPLDIP' +
str(i)))
                                  for i in range(1, 8)]])
        param.CPLTDE = np.vstack([param.CPLTDE,

[np.float32(properties_package.pcsaft.retrive_param_from_DB(c.CAS, 'CPLTDE' +
str(i)))
                                  for i in range(1, 10)]])

# 装填聚合物链段 PCSAFT 参数
param.m = np.append(param.m,
np.float32(properties_package.pcsaft.retrive_param_from_DB('seg-IB-R', 'PCFTR')))
param.e = np.append(param.e,
np.float32(properties_package.pcsaft.retrive_param_from_DB('seg-IB-R', 'PCFTU')))
param.s = np.append(param.s,
np.float32(properties_package.pcsaft.retrive_param_from_DB('seg-IB-R', 'PCFTV')))
param.MW = np.append(param.MW,
np.float32(properties_package.pcsaft.retrive_param_from_DB('seg-IB-R', 'MW')))
param.r = np.append(param.r,
np.float32(properties_package.pcsaft.retrive_param_from_DB('seg-IB-R', 'PCFTR')))
param.CPLID = np.append(param.CPLID, 'POLY')
param.k_ij = np.zeros([len(component_list), len(component_list)])
```

流程图（见下页）


开始
定义活性种数
site_num = 3
定义聚合物 IIR
创建Polymer对象
定义普通组分列表
创建Component对象数组
初始化全局组分列表
GlobalComponentManager.component_list_gen()
创建PCSAFT物性包
properties_package =
IIR_PCSAFT()
初始化参数数组
param.m, param.e,
param.s等
遍历组分列表
是否为普通组分?
(Component类型)
是
否
获取并添加组分PCSAFT参数
- m (segment number)
- e (energy parameter)
- s (segment diameter)
- MW (molecular weight)
- CPLID, CPLDIP, CPLTDE
添加聚合物链段PCSAFT参数
- m, e, s, MW
- r (segment number)
- CPLID = 'POLY'
创建二维相互作用参数矩阵
param.k_ij
结束

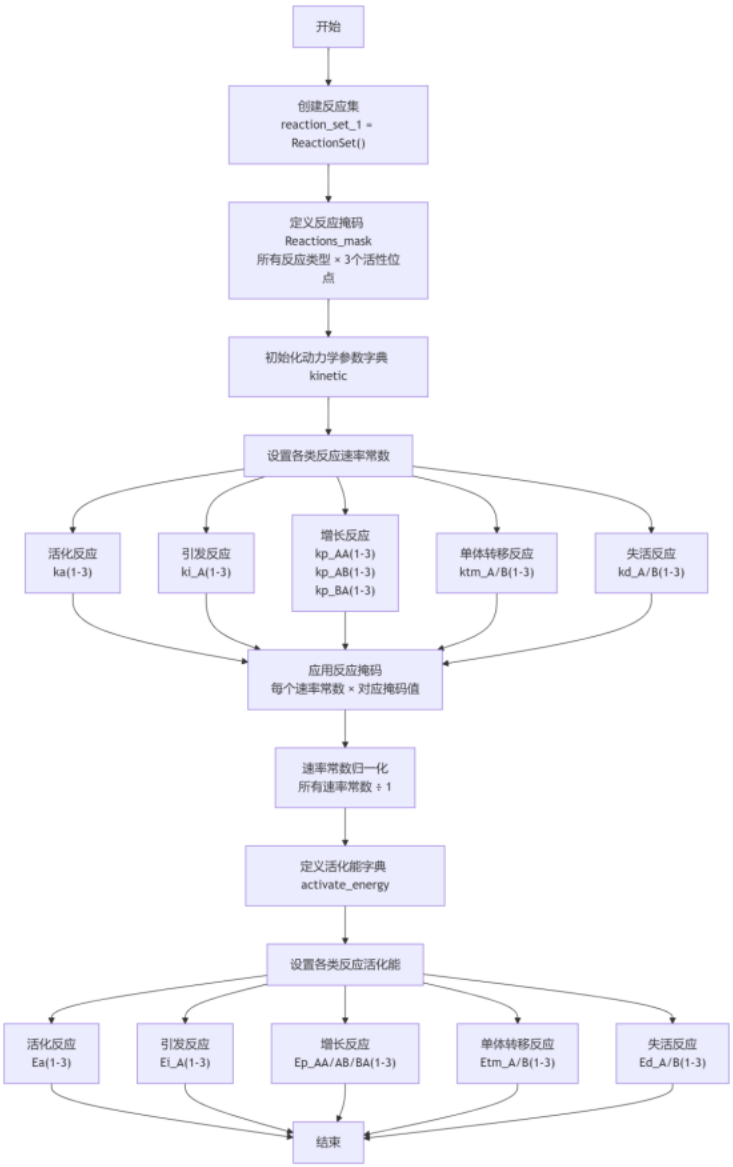
（2）反应集定义

```
# 创建反应集对象

reaction_set_1 = ReactionSet()

# 定义基元反应 mask，1 代表激活，0 代表不激活

Reactions_mask = {
    'ka(1)': 1,
    'ka(2)': 1,
    'ka(3)': 1,
    'ki_A(1)': 1,
    'ki_A(2)': 1,
    'ki_A(3)': 1,
    'kp_AA(1)': 1,
    'kp_AA(2)': 1,
    'kp_AA(3)': 1,
    'kp_AB(1)': 1,
    'kp_AB(2)': 1,
    'kp_AB(3)': 1,
    'kp_BA(1)': 1,
    'kp_BA(2)': 1,
    'kp_BA(3)': 1,
    'ktm_A(1)': 1,
    'ktm_A(2)': 1,
    'ktm_A(3)': 1,
    'ktm_B(1)': 1,
    'ktm_B(2)': 1,
    'ktm_B(3)': 1,
    'kd_A(1)': 1,
    'kd_A(2)': 1,
    'kd_A(3)': 1,
    'kd_B(1)': 1,
    'kd_B(2)': 1,
    'kd_B(3)': 1
}
# 定义基元反应动力学指前因子 [1/sec]

kinetic = {'ka(1)': 66 * Reactions_mask['ka(1)'],
           'ka(2)': 66 * Reactions_mask['ka(2)'],
           'ka(3)': 66 * Reactions_mask['ka(3)'],
           'ki_A(1)': 66 * Reactions_mask['ki_A(1)'],
           'ki_A(2)': 66 * Reactions_mask['ki_A(2)'],
```

```
          'ki_A(3)': 66 * Reactions_mask['ki_A(3)'],
          'kp_AA(1)': 316.938 * Reactions_mask['kp_AA(1)'],
          'kp_AA(2)': 183.358 * Reactions_mask['kp_AA(2)'],
          'kp_AA(3)': 24.7036 * Reactions_mask['kp_AA(3)'],
          'kp_AB(1)': 55.4641 * Reactions_mask['kp_AB(1)'],
          'kp_AB(2)': 32.0877 * Reactions_mask['kp_AB(2)'],
          'kp_AB(3)': 4.32314 * Reactions_mask['kp_AB(3)'],
          'kp_BA(1)': 63.3876 * Reactions_mask['kp_BA(1)'],
          'kp_BA(2)': 36.6717 * Reactions_mask['kp_BA(2)'],
          'kp_BA(3)': 4.94073 * Reactions_mask['kp_BA(3)'],
          'ktm_A(1)': 0.035 * Reactions_mask['ktm_A(1)'],
          'ktm_A(2)': 0.105 * Reactions_mask['ktm_A(2)'],
          'ktm_A(3)': 0.113333 * Reactions_mask['ktm_A(3)'],
          'ktm_B(1)': 0.035 * Reactions_mask['ktm_B(1)'],
          'ktm_B(2)': 0.105 * Reactions_mask['ktm_B(2)'],
          'ktm_B(3)': 0.113333 * Reactions_mask['ktm_B(3)'],
          'kd_A(1)': 0.0042 * Reactions_mask['kd_A(1)'],
          'kd_A(2)': 0.0042 * Reactions_mask['kd_A(2)'],
          'kd_A(3)': 0.0042 * Reactions_mask['kd_A(3)'],
          'kd_B(1)': 0.0042 * Reactions_mask['kd_B(1)'],
          'kd_B(2)': 0.0042 * Reactions_mask['kd_B(2)'],
          'kd_B(3)': 0.0042 * Reactions_mask['kd_B(3)']
          }

# 全局缩放 （测试时可用，默认除以 1）
for c in kinetic:
    kinetic[c] = kinetic[c] / 1

# 定义活化能 [kJ/mol]
activate_energy = {

    'Ea(1)': 35,
    'Ea(2)': 35,
    'Ea(3)': 35,
    'Ei_A(1)': 35,
    'Ei_A(2)': 35,
    'Ei_A(3)': 35,
    'Ep_AA(1)': 45,
    'Ep_AA(2)': 45,
    'Ep_AA(3)': 45,
    'Ep_AB(1)': 45,
    'Ep_AB(2)': 45,
    'Ep_AB(3)': 45,
    'Ep_BA(1)': 45,
```

```
    'Ep_BA(2)': 45,
    'Ep_BA(3)': 45,
    'Etm_A(1)': 75,
    'Etm_A(2)': 75,
    'Etm_A(3)': 75,
    'Etm_B(1)': 75,
    'Etm_B(2)': 75,
    'Etm_B(3)': 75,
    'Ed_A(1)': 2,
    'Ed_A(2)': 2,
    'Ed_A(3)': 2,
    'Ed_B(1)': 2,
    'Ed_B(2)': 2,
    'Ed_B(3)': 2,
}
```

流程图（见下页）


开始
创建反应集
reaction_set_1 =
ReactionSet()
定义反应掩码
Reactions_mask
所有反应类型 × 3个活性位
点
初始化动力学参数字典
kinetic
设置各类反应速率常数
活化反应
ka(1-3)
引发反应
ki_A(1-3)
增长反应
kp_AA(1-3)
kp_AB(1-3)
kp_BA(1-3)
单体转移反应
ktm_A/B(1-3)
失活反应
kd_A/B(1-3)
应用反应掩码
每个速率常数 × 对应掩码值
速率常数归一化
所有速率常数 ÷ 1
定义活化能字典
activate_energy
设置各类反应活化能
活化反应
Ea(1-3)
引发反应
Ei_A(1-3)
增长反应
Ep_AA/AB/BA(1-3)
单体转移反应
Etm_A/B(1-3)
失活反应
Ed_A/B(1-3)
结束

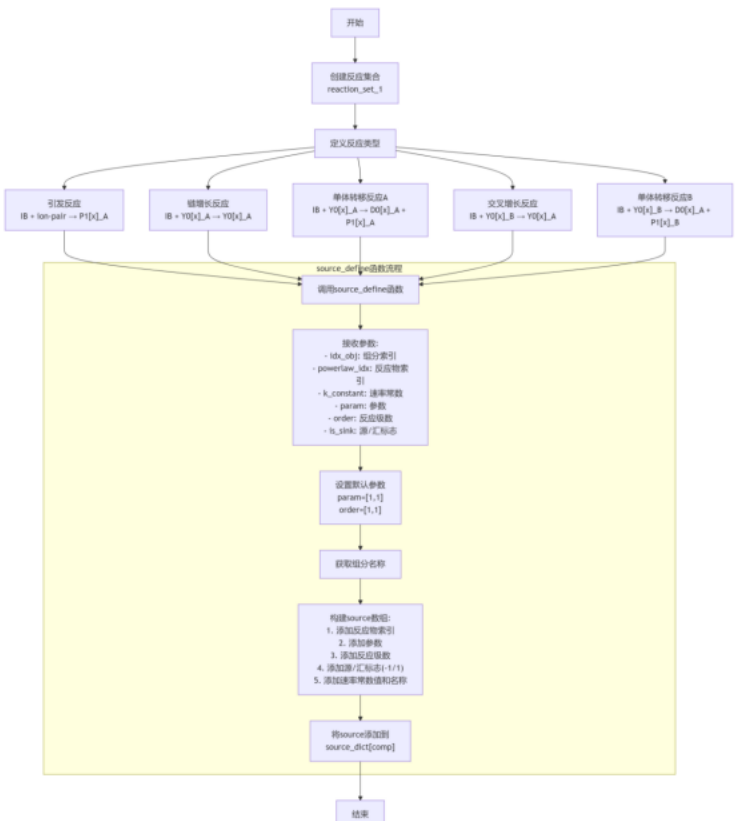
（3）定义反应源项（source difinition）

以 IB 为例

```
# IB + ion-pair ---> P1[1]_A
reaction_set_1.source_define(0, [0, 6], {'name': 'Ki_A(1)', 'value':
kinetic['ki_A(1)']}, [1, 1], [1, 1], True)
# IB + ion-pair ---> P1[2]_A
reaction_set_1.source_define(0, [0, 7], {'name': 'Ki_A(2)', 'value':
kinetic['ki_A(2)']}, [1, 1], [1, 1], True)
# IB + ion-pair ---> P1[3]_A
reaction_set_1.source_define(0, [0, 8], {'name': 'Ki_A(3)', 'value':
kinetic['ki_A(3)']}, [1, 1], [1, 1], True)
# IB + Y0[1]_A --->  Y0[1]_A
reaction_set_1.source_define(0, [0, 15], {'name': 'Kp_AA(1)', 'value':
kinetic['kp_AA(1)']}, [1, 1], [1, 1], True)
# IB + Y0[2]_A --->  Y0[2]_A
reaction_set_1.source_define(0, [0, 16], {'name': 'Kp_AA(2)', 'value':
kinetic['kp_AA(2)']}, [1, 1], [1, 1], True)
# IB + Y0[3]_A --->  Y0[3]_A
reaction_set_1.source_define(0, [0, 17], {'name': 'Kp_AA(3)', 'value':
kinetic['kp_AA(3)']}, [1, 1], [1, 1], True)
# IB + Y0[1]_A ---> D0[1]_A+P1[1]_A
reaction_set_1.source_define(0, [0, 15], {'name': 'Ktm_A(1)', 'value':
kinetic['ktm_A(1)']}, [1, 1], [1, 1], True)
# IB + Y0[2]_A ---> D0[2]_A+P1[1]_A
reaction_set_1.source_define(0, [0, 16], {'name': 'Ktm_A(2)', 'value':
kinetic['ktm_A(2)']}, [1, 1], [1, 1], True)
# IB + Y0[3]_A ---> D0[3]_A+P1[3]_A
reaction_set_1.source_define(0, [0, 17], {'name': 'Ktm_A(3)', 'value':
kinetic['ktm_A(3)']}, [1, 1], [1, 1], True)
# IB + Y0[1]_B ---> Y0[1]_A
reaction_set_1.source_define(0, [0, 18], {'name': 'Kp_BA(1)', 'value':
kinetic['kp_BA(1)']}, [1, 1], [1, 1], True)
# IB + Y0[2]_B ---> Y0[1]_A
reaction_set_1.source_define(0, [0, 19], {'name': 'Kp_BA(2)', 'value':
kinetic['kp_BA(2)']}, [1, 1], [1, 1], True)
# IB + Y0[3]_B ---> Y0[1]_A
reaction_set_1.source_define(0, [0, 20], {'name': 'Kp_BA(3)', 'value':
kinetic['kp_BA(3)']}, [1, 1], [1, 1], True)
# IB + Y0[1]_B ---> D0[1]_A+P1[1]_B
reaction_set_1.source_define(0, [0, 18], {'name': 'Ktm_B(1)', 'value':
kinetic['ktm_B(1)']}, [1, 1], [1, 1], True)
# IB + Y0[2]_B ---> D0[2]_A+P1[1]_B
```

```
reaction_set_1.source_define(0, [0, 19], {'name': 'Ktm_B(2)', 'value':
kinetic['ktm_B(2)']}, [1, 1], [1, 1], True)
# IB + Y0[3]_B ——> D0[3]_A+P1[3]_B
reaction_set_1.source_define(0, [0, 20], {'name': 'Ktm_B(3)', 'value':
kinetic['ktm_B(3)']}, [1, 1], [1, 1], True)
```

流程图



（4）定义反应器模型 Block

```
reaction_set_1.ea = activate_energy
# print(reaction_set_1.source_dict)

# 定义流入反应器流股
flow_toR130 = Flow(100, 103, 'R130_InLet')

# 定义流股组分的质量流量 [kg/Hr]
flow_toR130.set_MassFlow_conventional(
    {'IB': 4244.38, 'IP': 95.178, 'HCL': 0.211972, 'EADC': 3.28092, 'HEXANE': 0.0,
'CH3CL': 8641.22})

# 定义求解变量名称
component_index = [component for component in flow_toR130.comp_dict]

# 定义初值（由于 CH3CL 是惰性组分）
init_values = {component: flow_toR130.comp_dict[component]['mole_flow'] if component
== 'CH3CL' else 5 for
               component in flow_toR130.comp_dict}
init_values_r = {component: 60000. for
                 component in flow_toR130.comp_dict}

# 定义撕裂流股最大收敛次数
Max_tear_iter = 2500

# 定义撕裂流股对象
tear_flow = Flow(179, 101325, 'tear_flow')
# for c in tear_flow.comp_dict:
tear_flow.comp_dict['CH3CL']['mole_flow'] = 81000.

# 定义撕裂循环
for i in range(Max_tear_iter):

    # 定义全局计算模型
    global_model = pyo.ConcreteModel()
    # # global_model.tear_flow = pyo.Var(component_index,
    #                                  initialize=init_values,
    #                                  domain=pyo.NonNegativeReals)
    # global_model.tear_flow = pyo.Var(component_index, domain=pyo.NonNegativeReals,
    #                                  initialize=init_values_r)
    # 定义反应器模型 1 的 Block
    global_model.reactor_1 = pyo.Block()
```

```
    # 定义 reactor1 的求解变量
     global_model.reactor_1.outflow = pyo.Var(component_index,
initialize=init_values,
                                             domain=pyo.NonNegativeReals)

    # 定义去反应器 1 的流股对象
    flow_to_reactor1 = Flow(T1, 101325, 'to_reactor1')

    # R1 的进料流股 = 纯 R1 进料 + 撕裂流股进料
    for c in flow_to_reactor1.comp_dict:
        flow_to_reactor1.comp_dict[c]['mole_flow'] =
flow_toR130.comp_dict[c]['mole_flow'] + tear_flow.comp_dict[c][
            'mole_flow']

    # 实例化反应器模型数据结构
    reactor1_model = CstrSingleLiqPhase(T1, 101325, 2, flow_to_reactor1,
reaction_set_1, properties_package)

    #定义 reactor1 的流出流股
    Out_flow_reactor1 = Flow(reactor1_model.Temperature, reactor1_model.Pressure,
'OutFlow_reactor1')

    # 流股连接方程
    for c in Out_flow_reactor1.comp_dict:
        Out_flow_reactor1.comp_dict[c]['mole_flow'] =
global_model.reactor_1.outflow[c]

    # 定义 DPN 的计算表达式
    global_model.reactor_1.dpn = pyo.Expression(rule=reactor1_model.compute_dpn)
    # 定义体积流量计算表达式
    global_model.reactor_1.Q =
pyo.Expression(rule=reactor1_model.volume_flow_rate_rule)

    # 添加质量守恒约束
    global_model.reactor_1.mass_balance = pyo.ConstraintList()
    for eq in reactor1_model.mass_balance(global_model.reactor_1):
        global_model.reactor_1.mass_balance.add(eq == 0.)

## Reactor2
=====================================================================
    global_model.reactor_2 = pyo.Block()
    global_model.reactor_2.outflow = pyo.Var(component_index, initialize=init_values,
domain=pyo.NonNegativeReals)
```

```
    reactor2_model = CstrSingleLiqPhase(T1, 101325, 1.3, Out_flow_reactor1,
reaction_set_1, properties_package)
    Out_flow_reactor2 = Flow(reactor2_model.Temperature, reactor2_model.Pressure,
'OutFlow_reactor2')
    for c in Out_flow_reactor2.comp_dict:
        Out_flow_reactor2.comp_dict[c]['mole_flow'] =
global_model.reactor_2.outflow[c]

    global_model.reactor_2.dpn = pyo.Expression(rule=reactor2_model.compute_dpn)
    global_model.reactor_2.Q =
pyo.Expression(rule=reactor2_model.volume_flow_rate_rule)
    global_model.reactor_2.mass_balance = pyo.ConstraintList()
    for eq in reactor2_model.mass_balance(global_model.reactor_2):
        global_model.reactor_2.mass_balance.add(eq == 0.)

# Reactor 3
=======================================================================

    global_model.reactor_3 = pyo.Block()
    global_model.reactor_3.outflow = pyo.Var(component_index, initialize=init_values,
domain=pyo.NonNegativeReals)
    reactor3_model = CstrSingleLiqPhase(T1, 101325, 1.3, Out_flow_reactor2,
reaction_set_1, properties_package)
    Out_flow_reactor3 = Flow(reactor3_model.Temperature, reactor3_model.Pressure,
'OutFlow_reactor3')
    for c in Out_flow_reactor3.comp_dict:
        Out_flow_reactor3.comp_dict[c]['mole_flow'] =
global_model.reactor_3.outflow[c]
    global_model.reactor_3.dpn = pyo.Expression(rule=reactor3_model.compute_dpn)
    global_model.reactor_3.Q =
pyo.Expression(rule=reactor3_model.volume_flow_rate_rule)
    global_model.reactor_3.mass_balance = pyo.ConstraintList()
    for eq in reactor3_model.mass_balance(global_model.reactor_3):
        global_model.reactor_3.mass_balance.add(eq == 0.)

# Reactor 4
=======================================================================
    global_model.reactor_4 = pyo.Block()
    global_model.reactor_4.outflow = pyo.Var(component_index, initialize=init_values,
domain=pyo.NonNegativeReals)
    reactor4_model = CstrSingleLiqPhase(T4, 101325, 2, Out_flow_reactor3,
reaction_set_1, properties_package)
    Out_flow_reactor4 = Flow(reactor4_model.Temperature, reactor4_model.Pressure,
'OutFlow_reactor4')
```

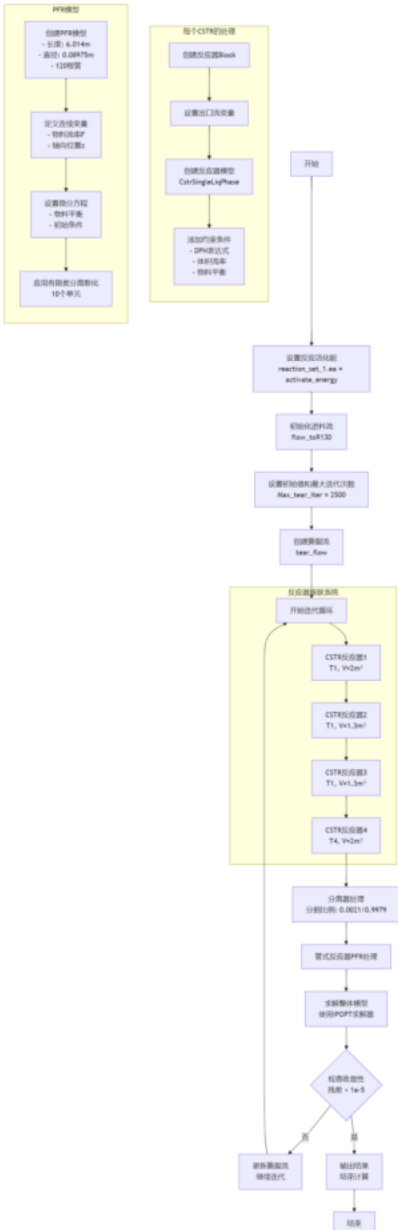
```
    for c in Out_flow_reactor4.comp_dict:
        Out_flow_reactor4.comp_dict[c]['mole_flow'] =
global_model.reactor_4.outflow[c]
    Out_flow_reactor4.Temperature = T4
    global_model.reactor_4.dpn = pyo.Expression(rule=reactor4_model.compute_dpn)
    global_model.reactor_4.Q =
pyo.Expression(rule=reactor4_model.volume_flow_rate_rule)
    global_model.reactor_4.mass_balance = pyo.ConstraintList()
    for eq in reactor4_model.mass_balance(global_model.reactor_4):
        global_model.reactor_4.mass_balance.add(eq == 0.)
    #

    # n_vars = sum(1 for v in global_model.component_data_objects(pyo.Var))
    # n_cons = sum(1 for c in global_model.component_data_objects(pyo.Constraint))
    # print(f"Number of variables: {n_vars}")
    # print(f"Number of constraints: {n_cons}")
    # print(f"Degrees of freedom: {n_vars - n_cons}")

    # 求解器定义 IPOPT
    solver = SolverFactory('ipopt')
    # 定义最大迭代次数
    solver.options['max_iter'] = 5000
    # 求解模型
    solver.solve(global_model)

# Spliter
================================================================
    spliter = Spliter(Out_flow_reactor4, [0.0021, 1 - 0.0021])
    spliter.mass_balance()
    # for strm in spliter.split_Out_flow_list:
    #     strm.Temperature = T4

# PFR1
=================================================================


# 定义 PFR1 模型
 pfr1 = pyo.ConcreteModel()
    # # #
    pfr1_model = PFRSingleliqPhase(t=179, p=101325, l=6.014, D=0.08975,
inflow=spliter.split_Out_flow_list[1],
                                   rx_set=reaction_set_1, prop=properties_package,
U=167.36, dH_rx=0, T_cool_in=172.,
                                   T_cool_out=168., multitubes=120,
                                   diameter=0.08975)
```

```
# 定义 PFR 初始摩尔流量初值函数
    def initialize_F(model, comp, z):
        if z == 0:
            return (pfr1_model.Inflow.comp_dict[comp]['mole_flow'])
        else:
            return pfr1_model.Inflow.comp_dict[comp]['mole_flow'] + 1e-6


    #定义 PFR 温度分布初值
    def initialize_T(model, z):
        if z == 0:
            return pfr1_model.Inflow.Temperature
        else:
            return pfr1_model.Inflow.Temperature

# PFR 离散域定义
    pfr1.z = dae.ContinuousSet(bounds=(0, pfr1_model.Length))
# PFR 各组分摩尔流量变量定义
    pfr1.F = pyo.Var(component_index, pfr1.z,
                     domain=pyo.NonNegativeReals,
                     initialize=initialize_F)
# PFR 温度变量定义
    pfr1.T = pyo.Var(pfr1.z, initialize=initialize_T)

# 温度导数定义
    pfr1.dTdz = dae.DerivativeVar(pfr1.T, wrt=pfr1.z)
# 摩尔流量导数定义
    pfr1.dFdz = dae.DerivativeVar(pfr1.F, wrt=pfr1.z)

# pfr 出口流股定义
    Out_flow_pfr1 = Flow(179, 101, 'OutFlow_pfr1')

# 连接方程约束定义 入口等于分流器出口
    pfr1.initial_flow_constraint = pyo.ConstraintList()
    for c in Out_flow_pfr1.comp_dict:
        Out_flow_pfr1.comp_dict[c]['mole_flow'] = pfr1.F[c, pfr1.z.last()]
        pfr1.initial_flow_constraint.add(
            spliter.split_Out_flow_list[1].comp_dict[c]['mole_flow'] == pfr1.F[c, 0])
        # flow_to_reactor1.comp_dict[c]['mole_flow']== global_model.pfr1.F[c, 0])


    #
global_model.pfr1.initial_flow_constraint.add(spliter.split_Out_flow_list[1].Temper
ature == global_model.pfr1.T[0])
```

```
    # pfr1.initial_flow_constraint.add(flow_to_reactor1.Temperature == pfr1.T[0])
    # 定义 dpn计算表达式
    pfr1.dpn = pyo.Expression(pfr1.z, rule=pfr1_model.compute_dpn)
    # 定义体积流量计算表达式
    pfr1.V_flow = pyo.Expression(pfr1.z, rule=pfr1_model.volume_flow_rate_rule)
    # 定义混合热容计算表达式
    pfr1.cp_mix = pyo.Expression(pfr1.z, rule=pfr1_model.cp_mix_rule)
    # 定义浓度计算表达式
    pfr1.C = pyo.Expression(component_index, pfr1.z,
rule=pfr1_model.concentration_rule)

    # 添加质量守恒约束
    pfr1.mass_balance = pyo.Constraint(component_index, pfr1.z,
rule=pfr1_model.mass_balance)

    # 添加热量守恒表达式
    # global_model.pfr1.heat_balance = pyo.Constraint(global_model.pfr1.z,
rule=pfr1_model.heat_balance)
    # 使用有限差分法对变量域进行离散
    discretizer = pyo.TransformationFactory('dae.finite_difference')
    # 离散单元数 10 二阶中心差分
    discretizer.apply_to(pfr1, nfe=10, scheme='CENTRAL')
    #
    # global_model.tear_constraints = pyo.ConstraintList()
    # for c in component_index:
    #     global_model.tear_constraints.add(
    #         global_model.tear_flow[c] == global_model.pfr1.F[c,
global_model.pfr1.z.last()]
    #     )
    # 求解 PFR 模型 求解器设置
    solver2 = SolverFactory('ipopt', tee=True)
    solver2.options['max_iter'] = 10000
    # 线性求解器使用 ma57
    solver2.options['linear_solver'] = 'ma57'
    solver2.solve(pfr1)

    # 定义撕裂流股
    tear_flow_new = Flow(179, 101325, 'tear_new')
    # 定义撕裂流股残差
    residuals = []
```

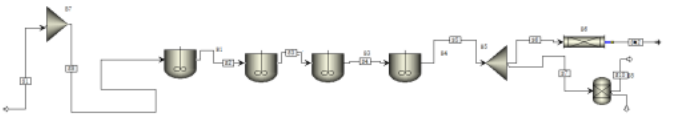
```
    # 计算断裂流股残差
    for c in Out_flow_pfr1.comp_dict:
        tear_flow_new.comp_dict[c]['mole_flow'] =
pyo.value(Out_flow_pfr1.comp_dict[c]['mole_flow'])
        residuals.append((tear_flow_new.comp_dict[c]['mole_flow'] -
tear_flow.comp_dict[c]['mole_flow']) ** 2)

    res_sum = np.sum(residuals)

    # 在到达收敛判据前，持续迭代 容差 1e-5
    if res_sum > 1e-5:
        tear_flow = tear_flow_new
        print(i)
        print(res_sum)
        for c in flow_to_reactor1.comp_dict:
            print(f'{c}: ' + str(pyo.value(global_model.reactor_1.outflow[c])))
    else:
        for c in flow_to_reactor1.comp_dict:
            print(f'{c}: ' + str(pyo.value(global_model.reactor_1.outflow[c])))
        break
```

流程图（见下页）


PFR模型
创建PFR模型
- 长度: 6.014m
- 直径: 0.08975m
- 120根管
定义自续变量
- 物料流速F
- 轴向位置z
设置微分方程
- 物料平衡
- 初始条件
应用有限差分离散化
10个单元
每个CSTR初始化
创建反应器Block
设置出口流变量
创建反应器模型
CstrSingleLiqPhase
添加约束条件
- DPN表达式
- 体积流率
- 物料平衡
开始
设置反应活化能
reaction_set_1.ea = activate_energy
初始化进料流
flow_toR130
设置初始值和最大迭代次数
Max_tear_iter = 2500
创建撕裂流
tear_flow
反应器串联系统
开始迭代循环
CSTR反应器1
T1, V=2m³
CSTR反应器2
T1, V=1.3m³
CSTR反应器3
T1, V=1.3m³
CSTR反应器4
T4, V=2m³
分离器处理
分离比例: 0.0021/0.9979
管式反应器PFR处理
求解整体模型
使用IPOPT求解器
检查收敛性
残差 < 1e-5
否
是
更新撕裂流
继续迭代
输出结果
结束计算
结束

五、基准测试

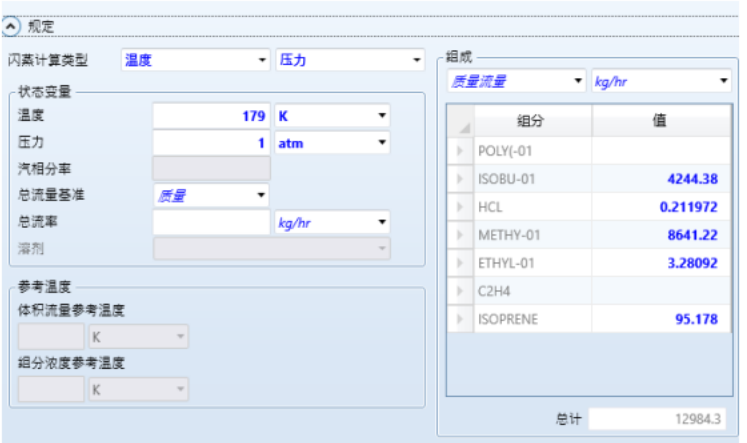
测试流程：



A. 四 CSTR 串级

1、进料及工况条件

工况	数值
异丁烯进料	4244.38
异戊二烯进料	95.178
氯化氢进料	0.211972
EADC 进料	3.28092
氯甲烷进料	8641.22
T1 温度	-94.0547
T2 温度	-93.8633
T3 温度	-93.672
T4 温度	-93.4087



2、反应动力学参数

全局缩放因子=1000

```
kinetic = {'ka(1)': 66 * Reactions_mask['ka(1)'],
           'ka(2)': 66 * Reactions_mask['ka(2)'],
           'ka(3)': 66 * Reactions_mask['ka(3)'],
           'ki_A(1)': 66 * Reactions_mask['ki_A(1)'],
           'ki_A(2)': 66 * Reactions_mask['ki_A(2)'],
           'ki_A(3)': 66 * Reactions_mask['ki_A(3)'],
           'kp_AA(1)': 316.938 * Reactions_mask['kp_AA(1)'],
           'kp_AA(2)': 183.358 * Reactions_mask['kp_AA(2)'],
           'kp_AA(3)': 24.7036 * Reactions_mask['kp_AA(3)'],
           'kp_AB(1)': 55.4641 * Reactions_mask['kp_AB(1)'],
           'kp_AB(2)': 32.0877 * Reactions_mask['kp_AB(2)'],
           'kp_AB(3)': 4.32314 * Reactions_mask['kp_AB(3)'],
           'kp_BA(1)': 63.3876 * Reactions_mask['kp_BA(1)'],
           'kp_BA(2)': 36.6717 * Reactions_mask['kp_BA(2)'],
```

```
            'kp_BA(3)': 4.94073 * Reactions_mask['kp_BA(3)'],
            'ktm_A(1)': 0.035 * Reactions_mask['ktm_A(1)'],
            'ktm_A(2)': 0.105 * Reactions_mask['ktm_A(2)'],
            'ktm_A(3)': 0.113333 * Reactions_mask['ktm_A(3)'],
            'ktm_B(1)': 0.035 * Reactions_mask['ktm_B(1)'],
            'ktm_B(2)': 0.105 * Reactions_mask['ktm_B(2)'],
            'ktm_B(3)': 0.113333 * Reactions_mask['ktm_B(3)'],
            'kd_A(1)': 0.0042 * Reactions_mask['kd_A(1)'],
            'kd_A(2)': 0.0042 * Reactions_mask['kd_A(2)'],
            'kd_A(3)': 0.0042 * Reactions_mask['kd_A(3)'],
            'kd_B(1)': 0.0042 * Reactions_mask['kd_B(1)'],
            'kd_B(2)': 0.0042 * Reactions_mask['kd_B(2)'],
            'kd_B(3)': 0.0042 * Reactions_mask['kd_B(3)']
            }
for c in kinetic:
    kinetic[c] = kinetic[c] / 1000
```

```
activate_energy = {

    'Ea(1)': 35,
    'Ea(2)': 35,
    'Ea(3)': 35,
    'Ei_A(1)': 35,
    'Ei_A(2)': 35,
    'Ei_A(3)': 35,
    'Ep_AA(1)': 45,
    'Ep_AA(2)': 45,
    'Ep_AA(3)': 45,
    'Ep_AB(1)': 45,
    'Ep_AB(2)': 45,
    'Ep_AB(3)': 45,
    'Ep_BA(1)': 45,
    'Ep_BA(2)': 45,
    'Ep_BA(3)': 45,
    'Etm_A(1)': 75,
    'Etm_A(2)': 75,
    'Etm_A(3)': 75,
    'Etm_B(1)': 75,
    'Etm_B(2)': 75,
    'Etm_B(3)': 75,
    'Ed_A(1)': 2,
    'Ed_A(2)': 2,
    'Ed_A(3)': 2,
```

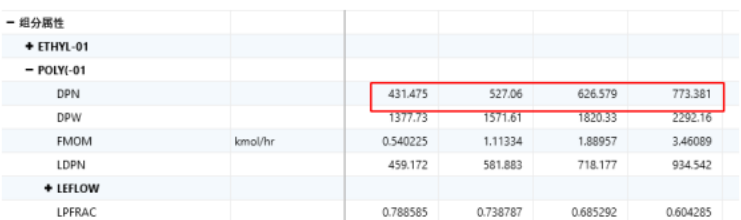
```
    'Ed_B(1)': 2,
    'Ed_B(2)': 2,
    'Ed_B(3)': 2,
}
```

类型	位 1	组分 1	位 2	组分 2	位 3	前因子(f) 1/sec	活化取(f) kJ/mol	前因子(r) 1/hr	活化取(r) cal/mol	参考温度 K	级数	关联数	系数 b	系数 d	标记
ACT-CATALYST	1	ETHYL-01		HCL		0.066	35			179.1					
ACT-CATALYST	2	ETHYL-01		HCL		0.066	35			179.1					
ACT-CATALYST	3	ETHYL-01		HCL		0.066	35			179.1					
CHAIN-INI-1	1	ISOBU-01				0.066	35			179.1					
CHAIN-INI-1	2	ISOBU-01				0.066	35			179.1					
CHAIN-INI-1	3	ISOBU-01				0.066	35			179.1					
PROPAGATION	1	ISOBU-02		ISOBU-01		0.316938	45			179.1					
PROPAGATION	2	ISOBU-02		ISOBU-01		0.183358	45			179.1					
PROPAGATION	3	ISOBU-02		ISOBU-01		0.0247036	45			179.1					
PROPAGATION	1	ISOBU-02		ISOPRENE		0.0554641	45			179.1					
PROPAGATION	2	ISOBU-02		ISOPRENE		0.0320877	45			179.1					
PROPAGATION	3	ISOBU-02		ISOPRENE		0.00432314	45			179.1					
PROPAGATION	1	ISOPR-01		ISOBU-01		0.0633876	45			179.1					
PROPAGATION	2	ISOPR-01		ISOBU-01		0.0366717	45			179.1					
PROPAGATION	3	ISOPR-01		ISOBU-01		0.00494073	45			179.1					
CHAT-MONOMER	1	ISOBU-02		ISOBU-01		3.5e-05	75			179.1					
CHAT-MONOMER	2	ISOBU-02		ISOBU-01		0.000105	75			179.1					
CHAT-MONOMER	3	ISOBU-02		ISOBU-01		0.000113333	75			179.1					
CHAT-MONOMER	1	ISOPR-01		ISOBU-01		3.5e-05	75			179.1					
CHAT-MONOMER	2	ISOPR-01		ISOBU-01		0.000105	75			179.1					
CHAT-MONOMER	3	ISOPR-01		ISOBU-01		0.000113333	75			179.1					
TERM-C-ION	1	ISOBU-02				4.2e-06	2			179.1					
TERM-C-ION	2	ISOBU-02				4.2e-06	2			179.1					
TERM-C-ION	3	ISOBU-02				4.2e-06	2			179.1					
TERM-C-ION	1	ISOPR-01				4.2e-06	2			179.1					
TERM-C-ION	2	ISOPR-01				4.2e-06	2			179.1					
TERM-C-ION	3	ISOPR-01				4.2e-06	2			179.1					

测试结果

（1）数均聚合度

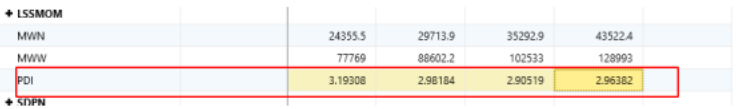
项目	This work	Aspen	相对误差(%)
DPN1	458.184	431.475	6.190196414
DPN2	553.637	527.06	5.042473231
DPN3	651.662	626.579	4.003118247
DPN4	805.064	773.381	4.096732133



− 组分属性					
✚ ETHYL-01					
− POLY(-01					
DPN		431.475	527.06	626.579	773.381
DPW		1377.73	1571.61	1820.33	2292.16
FMOM	kmol/hr	0.540225	1.11334	1.88957	3.46089
LDPN		459.172	581.883	718.177	934.542
✚ LEFLOW					
LPFRAC		0.788585	0.738787	0.685292	0.604285

（2）PDI

项目	This work	Aspen	相对误差(%)
PDI1	3.208	3.193080283	0.455109498
PDI2	2.999	2.981841558	0.5650164
PDI3	2.914	2.905191621	0.302349175
PDI4	2.990	2.963822493	0.888739834



（3）摩尔流量

流股编号	组分（This work）	Aspen
1	IB: 75.03069740493028	75.10886169
2	IB: 74.41189170777429	74.53760781
3	IB: 73.61184023298885	73.76392292
4	IB: 71.8512602448913	72.19783331
1	IP: 1.39524589017139	1.395499393
2	IP: 1.3932278344346574	1.393639108
3	IP: 1.390594577903143	1.39109705
4	IP: 1.3846822922363673	1.385858695
1	HCL: 0.00475734857061601	0.004819342
2	HCL: 0.0041697701035224	0.004244348
3	HCL: 0.0036642139020780793	0.003735701
4	HCL: 0.0029917242373404566	0.003092501
1	EADC: 0.02478801508106211	0.024850072
2	EADC: 0.024200436613968496	0.024275079
3	EADC: 0.023694880412524176	0.023766432
4	EADC: 0.023022390747786555	0.023123231
1	CH3CL: 171.15563258232234	1.71E+02
2	CH3CL: 171.15563258232234	1.71E+02
3	CH3CL: 171.15563258232234	1.71E+02
4	CH3CL: 171.15563258232234	1.71E+02
1	ion-pair[1]: 1.5647091059400189e-06	1.59E-06
2	ion-pair[1]: 1.3575383288689131e-06	1.38E-06
3	ion-pair[1]: 1.1807097121039272e-06	1.20E-06
4	ion-pair[1]: 9.595461392831615e-07	9.91E-07

1	ion-pair[2]: 1.5647091059400189e-06	1.59E-06
2	ion-pair[2]: 1.3575383288689131e-06	1.38E-06
3	ion-pair[2]: 1.1807097121039272e-06	1.20E-06
4	ion-pair[2]: 9.595461392831615e-07	9.91E-07
1	ion-pair[3]: 1.564709105940019e-06	1.59E-06
2	ion-pair[3]: 1.3575383288689133e-06	1.38E-06
3	ion-pair[3]: 1.1807097121039272e-06	1.20E-06
4	ion-pair[3]: 9.595461392831615e-07	9.91E-07

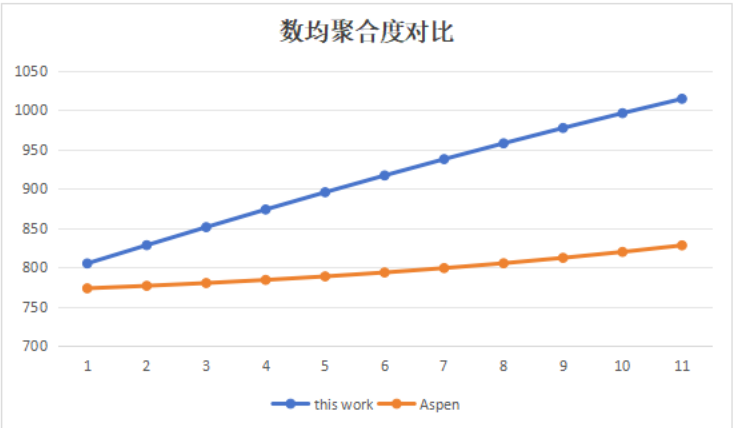
− 摩尔流量	kmol/hr	248.233	248.234	248.235	248.237
POLY(-01	kmol/hr	0.543494	1.11868	1.89693	3.47126
ISOBU-01	kmol/hr	75.1089	74.5376	73.7639	72.1978
HCL	kmol/hr	0.00481934	0.00424435	0.0037357	0.0030925
METHY-01	kmol/hr	171.156	171.156	171.156	171.156
ETHYL-01	kmol/hr	0.0248501	0.0242751	0.0237664	0.0231232
C2H4	kmol/hr	0	0	0	0
ISOPRENE	kmol/hr	1.3955	1.39364	1.3911	1.38586

B、单 PFR

测试结果

（1）数均聚合度

序号	数值	Aspen
1	805.0643	773.381185
2	828.3155	776.564885
3	851.2405	780.1051
4	873.7202	784.088752
5	895.6748	788.556927
6	917.0521	793.552897
7	937.8203	799.122076
8	957.9625	805.310637
9	977.4724	812.165784
10	996.3515	819.735026
11	1014.6072	828.0651


数均聚合度对比
1050
1000
950
900
850
800
750
700
1
2
3
4
5
6
7
8
9
10
11
this work
Aspen