# 模糊控制算法——C语言版本

此工程为C语言版本的模糊控制算法，我将在这里提供模糊控制算法的核心及相关的示例项目。



# 1. 工程框架

## 1.1 数据结构

### 1.1.1 类式链表容器

类式链表容器能够存放各种类型的数据，在实际结构中使用`void* data`指针进行管理。

```
struct list
{
    void *data;
    struct list *next;
};
```

由于`list.data`是`void*`类型，没办法在指针库中对数据进行更多的操作，因此需要使用时传入谓词或事件回调函数等对数据进行操作，也即对象的构造和析构函数等。

### 1.1.2 模糊运算

模糊运算是进行模糊推理的基础，在这里，仅讨论对模糊矩阵的运算。

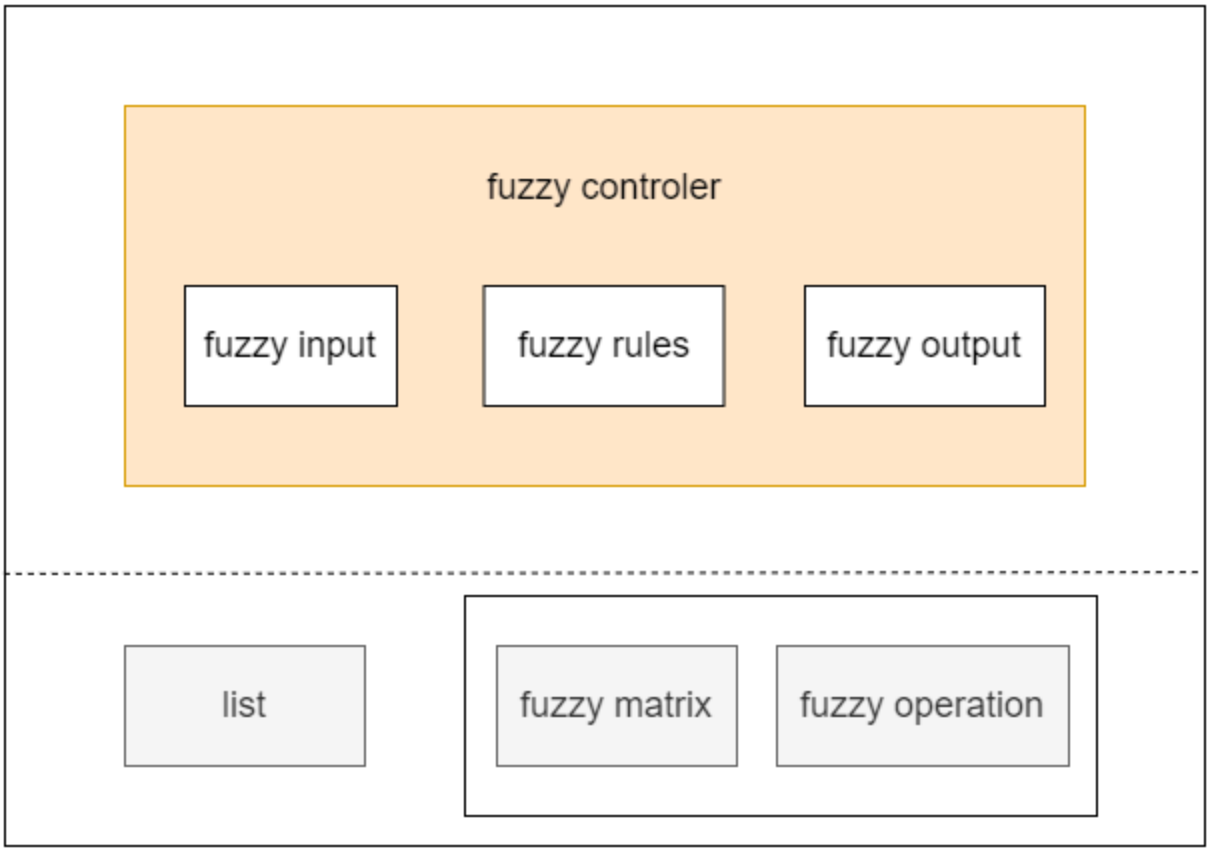
为了实现模糊矩阵运算，首先得有一个表示模糊矩阵的数据类型：

```
/**
* @brief Fuzzy matrix
*
* @memberof mat matrix
* @memberof row matrix rows
* @memberof col matrix columns
*/
struct fuzzy_matrix
{
    fuzzy_number** mat;
    fuzzy_size row;
    fuzzy_size col;
};
```

该数据类型使用二维指针对模糊矩阵进行管理，因此涉及到了动态内存分配，这是很危险的，所以我们需要将危险放在模块中，尽量避免使用者造成内存泄漏。

## 1.2 算法核心框架

算法核心框架图如下：




核心需要使用链表和一些基本的模糊矩阵及其运算，在它们的基础之上，搭建起模糊控制器，其包括输入输出和推理三大功能。

### 1.2.1 模糊输入

模糊输入部件（fc_input）需要记录单个论域的不同模糊子集，我将论域的名称称为`名字`，将模糊子集的名称称为`标签`，它们组成多对“名字—标签”对，用于后续辨别各个模糊子集，另外呢，它还需要能够实现将精确值模糊化的功能，因此我添加了一个模糊化函数，其相关API如下：

| 名称 | 参数 | 返回值 | |
|---|---|---|---|
| fc_input_register | struct fc_input* const,const char* | bool | 注册 |
| fc_input_unregister | struct fc_input* const | bool | 注销 |
| fc_input_add_fuzzy_set | const struct fc_input* const,const struct fuzzy_set* const | bool | 向模 |
| fc_input_clear_fuzzy_set | const struct fc_input* const | bool | 清空 |

| 名称 | 参数 | 返回值 | |
|---|---|---|---|
| fc_input_fuzzing | struct fc_input* const,accurate_number*,fc_size | bool | 进行 |
| fc_input_fuzzing_by_label | struct fc_input* const,accurate_number*,fc_size,fuzzy_number*,const char* | bool | 使用 |
| fc_input_print_data | struct fc_input* const,const char* | bool | 打印 |
| fc_input_print_fuzzy_set | struct fc_input* const,const char* | bool | 打印 |

### 1.2.2 模糊推理

#### 规则库

规则库（fc_rules）的实现是模糊推理的关键一步，我希望有一个方便用户使用、能够较为轻松解析的规则实现，借助于先前编写的 `StreamDeviceAT`，决定使用字符串作为规则条目，并附加一些规则编写规范，那么就能轻松解析了。

现在对具体需求进行分析，首先，需要制定规则编写规范，我希望的规则是下面这样的：

```
"IF A1 AND B2 OR C3 THEN U8"
```

上述规则中存在关键词 `IF`、`AND`、`OR`、`THEN`，这些关键词需要保留，用户不能将其作为输入和输出的名称及标签，不过其中的输入和输出并不好区分哪一部分是名称，那一部分是标签，需要进一步设计：

```
"IF A-1 AND B-2 OR C-3 THEN U-8"
```

现在我们就可以很好地区分名称和标签了，接下来的问题就是怎么设计多输出呢，多输出的各个输出之间并没有关系，无法使用 `AND` 或者 `OR` 对其进行辨别，因此我选择再添加一个关键词 `|`，其使用方法如下：

```
"IF A-1 AND B-2 OR C-3 THEN U-8 | V-9"
```

为了方便后续修改，我将关键词以静态常量数组的方式标记出来：

```
static const fc_rule_keyword __fc_rules_keyword_table[] = {
    "IF", "THEN",            // conditional segmentation keywords
    "AND", "OR",             // conditional relationship keywords
    "?",                     // conditional ignore keywords
    "|",                     // multiple output annotations for results
};
```

至此，规则条目的编写规范便已确定，接下来需要设计规则管理结构体了。

```
/**
 * @brief The rule controller
 *
 * @memberof rule_keyword_table
 * @memberof rule_keyword_num
 * @memberof rules rule library
 */
struct fc_rules
{
    const fc_rule_keyword* rule_keyword_table;
    fc_size rule_keyword_num;

    list_head rules;
};
```

使用指针访问关键词组，节省内存空间、方便管理，规则条目则使用链表进行存储，便于随时添加，链表中存放的规则是一个个字符串。

与之相关的规则对象的注册、注销，添加、清空规则等功能函数简介如下：

| 名称 | 参数 | 返回值 | |
|---|---|---|---|
| fc_rules_register | struct fc_rules* const | bool | 注册一个规则管理对象，注册时会将关键词 |
| fc_rules_unregister | struct fc_rules* const | bool | 注销一个规则管理对象，因为对象的链表中 |
| fc_rules_add_rule | const struct | bool | 向规则管理对象中添加一条规则，添加的规 |

| 名称 | 参数 | 返回值 | |
|---|---|---|---|
| | fc_rules* const,fc_rule_item | | |
| fc_rules_clear_rule | const struct fc_rules* const | bool | 清空规则管理对象中的规则，即清空链表 |
| fc_rules_get_rule_num | const struct fc_rules* const | fc_size | 获取规则管理对象中的规则数目 |
| fc_rules_print_rule | const struct fc_rules* const,const char* | bool | 打印规则管理对象中的所有规则 |

现在我能够使用规则管理对象管理我的规则，但是这些规则还是比较难处理的，我想要得到一个仅包含输入及其关系、输出内容的数据，为此我又设计了一个结构体：

```
struct fc_calculation
{
    list_head condition;
    list_head result;
};
```

其中包含两个链表，分别用来表示输入（条件）和输出（结果），为了方便管理及后续扩展，我将这两个链表中存放的数据的结构设计成同一个：

```
struct __fc_calculation_unit
{
    fc_rule_consition_result cr;
    fc_rules_opera_type opera;
};
```

这个结构中包括了输入/输出子集“名字—标签”对和对应的操作符，对于输入，目前有两个操作符，其分别是 $\vee$ 和 $\wedge$ 用来两者之间取大和取小。

对于这个计算式结构体（`struct fc_calculation`），我为其添加了如下几个操作函数：

| 名称 | 参数 | 返回值 | |
|---|---|---|---|
| fc_rules_create_calculation | struct fc_calculation* const | bool | 创建一个计算式对象，将会为对象创建两个 |
| fc_rules_export_calculation | struct fc_calculation* const,const struct fc_rules* const,const fc_index | bool | 从规则管理库中导出对应索引处的规则的计 |
| fc_rules_delete_calculation | struct fc_calculation* const | bool | 删除计算式 |
| fc_rules_print_calculation | const struct fc_calculation* const,const char* | bool | 打印出计算式 |

经过以上叙述，我们知道了规则库的功能是：

1. 管理规则库
2. 导出规则对应计算式

### 1.2.3 模糊输出

模糊输出部件（fc_output）将激活程度与相应的隶属函数结合，得到`Mamdani`曲线，所有Mamdani曲线取并集得到最终的输出曲线，最后结合实际选择合适的算法得到控制量。

对于部件的使用，有以下几个API。

| 名称 | 参数 | 返回值 | |
|---|---|---|---|
| fc_output_register | struct fc_output* const,const char*,accurate_number,accurate_number,accurate_number | bool | 注册 |

| 名称 | 参数 | 返回值 | |
| --- | --- | --- | --- |
| fc_output_unregister | struct fc_output* const | bool | 注销 |

**接收的推理数据**

该部件需要接收正确对应已有隶属函数的激活程度，并且由于接收的推理数据可能不会立刻被使用，因此采用分组的方式对推理数据进行管理。

因此采用分组接收推理结果的方式进行管理数据，而每一条推理数据包含`标签`和`激活程度`两部分，通过标签对其进行匹配。

```
struct inference_result
{
    const char* name_tag;
    fuzzy_number activate;
};
```

为方便管理这些数据条目，添加了相应的API。

**反模糊化算法**

1. 最大值法
2. 面积中心法

$$val = \frac{\int xf(x)dx}{\int f(x)dx}$$

### 1.2.4 三元归一

结合上述3个部件，便能组成一个模糊控制器。

## 1.3 算法外围框架

算法外围框架图如下：

算法外围框架.png