# 地球系统数值模拟装置项目
# 区域高精度长期气候变化风险模拟分系统
# 气候变化及极端气候事件模拟分析操作流程
## 培训教程

2022年5月

# 极端气候事件模拟分析操作流程

# 培训内容

- 基础公用函数
  - ✓ CSV读写
  - ✓ 数据归一化
  - ✓ 权重计算
  - ✓ 空间插值
- 灾害风险评估与区划实践（以高温为例）
  - ✓ 灾害事件判识
  - ✓ 致灾因子危险性评估
  - ✓ 灾害风险评估与区划

# 基础公用函数

## ➢ CSV文件数据读取

```
Datetime,TEM_Max,Lat,Lon,Alti,Province,City,Cnty,Station_Id_C,Station_levl,S
-999,-999,52.9667,122.5167,433.0,黑龙江,,,50136,,漠河
19510101,-999.0,,,,,,,,
19510102,-999.0,,,,,,,,
19510103,-999.0,,,,,,,,
19510104,-999.0,,,,,,,,
19510105,-999.0,,,,,,,,
19510106,-999.0,,,,,,,,
19510107,-999.0,,,,,,,,
19510108,-999.0,,,,,,,,
19510109,-999.0,,,,,,,,
19510110,-999.0
```

数据格式

```python
def readCsvData(infile=None, col_fields=None):
    """
    读取指定列名的数据
    :param infile: str, csv文件
    :param col_fields: list, 列名list
    :return:
    """
    try:
        data = pd.read_csv(infile, usecols=col_fields, encoding="utf-8")
    except:
        data = pd.read_csv(infile, usecols=col_fields, encoding="gbk")
    return data
```

CSV数据按列名读取函数

```python
infile = r"F:\气象灾害普查项目数据_整理\高温灾害气象数据\高温_50136.csv"
col_fields = ["TEM_Max"]
data = readCsvData(infile=infile, col_fields=col_fields)
```

函数调用

| | TEM_Max |
|---|---|
| 0 | -999.00000 |
| 1 | -999.00000 |
| 2 | -999.00000 |
| 3 | -999.00000 |
| 4 | -999.00000 |
| 5 | -999.00000 |
| 6 | -999.00000 |
| 7 | -999.00000 |
| 8 | -999.00000 |
| 9 | -999.00000 |
| 10 | -999.00000 |
| 11 | -999.00000 |
| 12 | -999.00000 |
| 13 | -999.00000 |
| 14 | -999.00000 |
| 15 | -999.00000 |
| 16 | -999.00000 |
| 17 | -999.00000 |

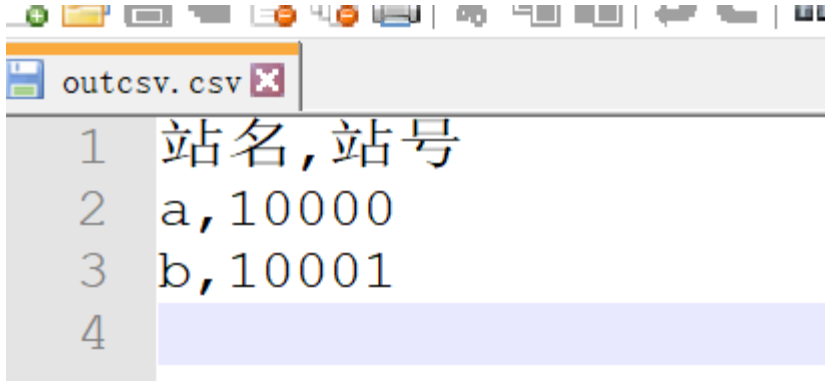
读取结果

# 基础公用函数

## ➢ 数据写入到CSV文件中

```python
def writeCsvFile(indict, csvfile):
    """
    将字典中的数据写进CSV
    :param csvdict: dict,
    :param csvfile: str, 文件路径
    :return:
    """
    try:
        dataframe = pd.DataFrame(indict)
        # 将DataFrame存储为csv,index表示是否显示行名，default=True
        dataframe.to_csv(csvfile, sep=',', index=False, encoding='GBK')
    finally:
        dataframe = None
```

字典数据写入csv函数




写出结果

```python
indict = {"站名":["a", "b"], "站号":["10000", "10001"]}
csvfile = r"D:\HT_Project\lyb_alg_code\train_code\outcsv.csv"
writeCsvFile(indict, csvfile)
```

函数调用

# 基础公用函数

## ➢ 数据归一化

归一化公式： $x' = \frac{x - x_{min}}{x_{max} - x_{min}}$

```python
def dataNormalProcess(data):
    """
    要素归一化处理
    :param data: array,数组, [[a1, b1],[a2, b2],[a3, b3],[a4, b4],[a5, b5],[a6, b6]]
    :return:
    """
    vals_min = np.nanmin(data, axis=0)
    vals_max = np.nanmax(data, axis=0)
    index_arr = (vals_max - vals_min) != 0
    vals_normed = np.zeros_like(data, dtype=np.float32)
    vals_normed[:, index_arr] = (data[:, index_arr] - vals_min[index_arr]) / \
                                (vals_max[index_arr] - vals_min[index_arr])
    return vals_normed
```

数据归一化函数

```python
data = np.array([[1, 1],[2, 2],[3, 3],[4, 4],[5, 5],[6, 6]])
vals_normed = dataNormalProcess(data)
print(vals_normed)
```

函数调用

| | 0 | 1 |
|---|---|---|
| 0 | 1 | 1 |
| 1 | 2 | 2 |
| 2 | 3 | 3 |
| 3 | 4 | 4 |
| 4 | 5 | 5 |
| 5 | 6 | 6 |

归一化前的数据

| | 0 | 1 |
|---|---|---|
| 0 | 0.00000 | 0.00000 |
| 1 | 0.20000 | 0.20000 |
| 2 | 0.40000 | 0.40000 |
| 3 | 0.60000 | 0.60000 |
| 4 | 0.80000 | 0.80000 |
| 5 | 1.00000 | 1.00000 |

归一化后的数据

# 基础公用函数

## ➢ 信息熵赋权法

信息熵表示系统的有序程度。在多指标综合评价中，熵权法可以客观的反映各评价指标的权重。一个系统的有序程度越高，则熵值越大，权重越小；反之，一个系统的无序程度越高，则熵值越小，权重越大。即对于一个评价指标，指标值之间的差距越大，则该指标在综合评价中所起的作用越大；如果某项指标的指标值全部相等，则该指标在综合评价中不起作用。因此，可用信息熵评价所获信息的有序度及其效用，从而尽量避免各评价指标权重的人为因素干扰，使评价结果更符合实际。

信息熵赋权法的主要计算步骤如下：

1）假定评价体系是由$m$个指标$n$个对象构成的系统，首先计算第$i$项指标下第$j$个对象的指标值$r_{ij}$所占指标比重$P_{ij}$；

$$P_{ij}=\frac{r_{ij}}{\sum_{j=1}^{n} r_{ij}}\ （i=1，2\cdots，m；j=1，2\cdots，n）$$

2）由熵权法计算第$i$个指标的熵值 $S_i$ ；

$$S_i=-\frac{1}{\ln}\sum_{j=1}^{n} P_{ij}\,ln\,P_{ij}\ (i=1,2\cdots,m;j=1,2\cdots,n)$$

3）计算第$i$个指标的熵权，确定该指标的客观权重$W_i$；

$$W_i=\frac{1-S_i}{\sum_{i=1}^{m}(1-S_i)}(i=1,2\cdots,m)$$

# 基础公用函数

## ➢ 信息熵赋权法函数

```
def calcWeights(vals_normed):
    """
    信息熵赋权法
    :param vals_normed: array,归一化后的数组（0~1）,[[a1, b1],[a2, b2],[a3, b3],[a4, b4],[a5, b5],[a6, b6]]
    :return:
    """
    vals_normed = np.array(vals_normed)
    n = np.shape(vals_normed)[0]
    if n==1:
        wi = np.full(np.shape(vals_normed)[1], np.nan, dtype=np.float)
    else:
        pij = np.zeros_like(vals_normed, dtype=np.float32)
        sum_array = np.nansum(vals_normed, axis=0)
        pij[:, (sum_array!=0)] = vals_normed[:, (sum_array!=0)] / sum_array[(sum_array!=0)]
        k = -1 / np.log(n)
        pij_log = np.zeros_like(vals_normed, dtype=np.float32)
        pij_log[pij!=0] = np.log(pij[pij!=0])
        si = k*np.nansum(pij_log*pij,axis=0)
        wi = np.zeros_like(si, dtype=np.float32)
        wi[(sum_array!=0)] = (1-si[(sum_array!=0)]) / np.nansum(1-si[(sum_array!=0)])
    return wi
```

信息熵赋权法函数

| | 0 | 1 |
|---|---|---|
| 0 | 0.00000 | 0.00000 |
| 1 | 0.20000 | 0.20000 |
| 2 | 0.40000 | 0.40000 |
| 3 | 0.60000 | 0.60000 |
| 4 | 0.80000 | 0.80000 |
| 5 | 1.00000 | 1.00000 |

输入数据

```
vals_normed = np.array([[0, 0],[0.2, 0.2],[0.4, 0.4],[0.6, 0.6],[0.8, 0.8],[1, 1]])
wi = calcWeights(vals_normed)
print(wi)
```

函数调用

```
[0.5 0.5]
```

输出的权重值

# 基础公用函数

## ➢ 标准差法等级划分

```
def divideRankStd(data, rank_content, rank_field, rank_descript_field,
                  min_value_field, max_value_field, nodata=0):
    """
    标准差方法
    :param data: list或tuple, 要分级的数据
    :param rank_content: dict, 数据划分等级信息{rank_field:[1,2,3,4], rank_descript_
                                                  max_value_field:[5,1,0,-1], min_valu
    :param rank_field: str, 划分等级的字段名, 包含在rank_content中
    :param rank_descript_field: str, 等级描述字段名, 包含在rank_content中
    :param min_value_field: str, 最小值字段, 包含在rank_content中
    :param max_value_field: str,最大值字段, 包含在rank_content中
    :param nodata: float, 无效值
    :return:
    """
    index_data = np.array(data)
    rank_numbers = len(rank_content[rank_field])
    newdata = index_data[index_data != nodata]
    if newdata.size <=1:
        rankdata = np.full(index_data.shape, rank_numbers, dtype=np.int)
        descriptdata = np.array([rank_content[rank_descript_field][rank_numbers - 1]] * (index_data.size),dtype='<U10')
    else:
        # 均值
        mean_h = np.nanmean(newdata)
        # 标准差
        s = np.nanstd(newdata)
        rankdata = np.full(index_data.shape, rank_numbers, dtype=np.int)
        descriptdata = np.array([rank_content[rank_descript_field][rank_numbers - 1]] * (index_data.size),  dtype='<U10')
        for i in range(rank_numbers):
            rank = int(rank_content[rank_field][i])
            rank_descript = rank_content[rank_descript_field][i]
            rank_min = rank_content[min_value_field][i]
            rank_max = rank_content[max_value_field][i]
            if (rank_min is None) & (rank_max is not None):
                rankdata[(index_data < (float(rank_max)*s + mean_h))] = rank
                descriptdata[(index_data < (float(rank_max)*s + mean_h))] = rank_descript
            if (rank_min is not None) & (rank_max is None):
                rankdata[(index_data >= (float(rank_min)*s + mean_h))] = rank
                descriptdata[(index_data >= (float(rank_min)*s + mean_h))] = rank_descript
            if (rank_min is None) & (rank_max is None):
                pass
            if (rank_min is not None) & (rank_max is not None):
                rankdata[(index_data >= (float(rank_min) *s+ mean_h)) & (index_data < (float(rank_max) *s + mean_h))] = rank
                descriptdata[(index_data >= (float(rank_min) *s+ mean_h)) & (index_data < (float(rank_max) *s + mean_h))] = rank_descript
    return rankdata, descriptdata
```

表 1-1 标准差法的分级方式

| 等级 | 分级范围 | 描述 |
|---|---|---|
| 1 级 | $\bar{x}+\sigma \le x$ | 高 |
| 2 级 | $\bar{x}+0.5\sigma \le x < \bar{x}+\sigma$ | 较高 |
| 3 级 | $\bar{x}-0.5\sigma \le x < \bar{x}+0.5\sigma$ | 中 |
| 4 级 | $\bar{x}-\sigma \le x < \bar{x}-0.5\sigma$ | 较低 |
| 5 级 | $x < \bar{x}-\sigma$ | 低 |

注：$x$ 为危险性指数值；$\bar{x}$为危险性指数均值；$\sigma$ 为危险性指数标准差。

# 基础公用函数

## ➢ 标准差法等级划分

```
data = np.array([0.1, 0.123, 0.4, 0.6, 0.7, 0.4, 0.45, 0.76, 0.45, 0.32, 0.32,0.222, 0.45, 0.67])
rank_content = {"rank_field":[1,2,3,4],
                "rank_descript_field":["高", "较高",  "较低", "低"],
                "max_value_field":[5,1,0,-1],
                "min_value_field":[1,0,-1,-5]}
rankdata, descriptdata = divideRankStd(data, rank_content, "rank_field", "rank_descript_field",
                                       "min_value_field", "max_value_field", nodata=0)
print(data)
print(rankdata)
print(descriptdata)
```

**函数调用**

```
[4 4 3 2 1 3 2 1 2 3 3 4 2 1]
['低' '低' '较低' '较高' '高' '较低' '较高' '高' '较高' '较低' '较低' '低' '较高' '高']
```

**返回结果（等级值及等级描述）**

# 基础公用函数

## ➢ 自然断点法等级划分

自然断点法运用了聚类的思维，它的核心思想与聚类一样：使每一组内部的相似性最大，而外部组与组之间的相异性最大。但是与聚类不一样的地方，聚类是不会关注每一类中的要素数量和范围的，而自然断点法在于它还会兼顾每一组之间的要素的范围和个数尽量相近。

自然断点法的主要计算过程如下：

1）定义长度为 n的序列X=[$x_1, x_2, x_3$ ….. $x_n$ ]，划分成$k$类；

2）计算每次分组的SDCM（原始数据的方差）、SDAM（每一类方差的和）、GVF（方差拟合优度）；

$$SDCM = \sum_{j=1}^{k} \frac{1}{N} \sum_{i=1}^{N} (x_{ij} - \bar{x_j})^2$$

$$SDAM = \frac{1}{N} \sum_{i=1}^{N} (x_i - \bar{x})^2$$

$$GVF = 1 - \frac{SDMC}{SDAM}$$

3）一定范围内，GVF越大，分类效果越好。根据每次分组中最大的GVF，确定分组结果。

# 基础公用函数

## ➢ 自然断点法等级划分

```
def divideRankByNaturalBreakpoint(data, rank_content, rank_field, rank_descript_field, nodata=0):
    """
      自然断点等级划分
      :param data: list或tuple, 要分级的数据
      :param rank_content: dict,数据划分等级信息{rank_field:[1,2,3,4], rank_descript_field:["高", "较高",  "较低", "低"]}
      :param rank_field: str, 划分等级的字段名, 包含在rank_content中
      :param rank_descript_field: str, 等级描述字段名, 包含在rank_content中
      :param nodata: float, 无效值
      :return:
    """
    index_data = np.array(data)
    # 划分等级的个数
    rank_numbers = len(rank_content[rank_field])
    # 自然断点法输出分段阈值
    # newdata = index_data[(index_data!=-999)&(index_data!=0)]
    newdata = index_data[index_data != nodata]
    if newdata.size <=1:
        rankdata = np.full(index_data.shape, rank_numbers, dtype=np.int)
        descriptdata = np.array([rank_content[rank_descript_field][rank_numbers - 1]] * (index_data.size), dtype='<U10')
    else:
        if newdata.size<=rank_numbers+2:
            newdata = np.array(list(newdata)+[np.mean(newdata)]*(rank_numbers+2-newdata.size))
        else:
            pass
        thresholds = list(jenkspy.jenks_breaks(newdata, rank_numbers))
        thresholds.sort(reverse=True)
        rankdata = np.full(index_data.shape, rank_numbers, dtype=np.int)
        descriptdata = np.array([rank_content[rank_descript_field][rank_numbers - 1]] * (index_data.size), dtype='<U10')
        for i in range(len(thresholds)-1):
            if i == 0:
                rankdata[(index_data >= thresholds[i + 1])] = rank_content[rank_field][i]
                descriptdata[(index_data >= thresholds[i + 1])] = rank_content[rank_descript_field][i]
            else:
                if i==len(thresholds)-2:
                    rankdata[(index_data < thresholds[i])] = rank_content[rank_field][i]
                    descriptdata[(index_data < thresholds[i])] = rank_content[rank_descript_field][i]
                else:
                    rankdata[(index_data<thresholds[i]) & (index_data>=thresholds[i+1])] = rank_content[rank_field][i]
                    descriptdata[(index_data<thresholds[i]) & (index_data>=thresholds[i+1])] = rank_content[rank_descript_field][i]
    return rankdata, descriptdata
```

# 基础公用函数

## ➢ 自然断点法等级划分

```
data = np.array([0.1, 0.12, 0.4, 0.6, 0.7, 0.4, 0.4, 0.76, 0.45, 0.32, 0.32,0.22, 0.45, 0.6])
rank_content = {"rank_field":[1,2,3,4], "rank_descript_field":["高", "较高",  "较低", "低"]}
rankdata, descriptdata = divideRankByNaturalBreakpoint(data, rank_content, "rank_field",
                                                       "rank_descript_field", nodata=0)
print(data)
print(rankdata)
print(descriptdata)
```

**函数调用**

```
[0.1  0.12 0.4  0.6  0.7  0.4  0.4  0.76 0.45 0.32 0.32 0.22 0.45 0.6 ]
[4 4 3 1 1 3 3 1 2 3 3 3 2 1]
['低' '低' '较低' '高' '高' '较低' '较低' '高' '较高' '较低' '较低' '较低' '较高' '高']
```

**返回结果（等级值及等级描述）**

# 基础公用函数

## ➢ 百分位法等级划分

百分位法又称为百分位数，是数据统计中一种常用的方法。具体定义为把一组统计数据按其数值从小到大顺序排列，并按数据个数100等分。在第ρ个分界点（称为百分位点）上的数值，称为第ρ个百分位数（ρ=1,2,…，99）。在第ρ个分界点到第ρ+1个分界点之间的数据，称为处于第ρ个百分位数。百分位数计算公式如下：

$$P_m = L + \frac{\left(\frac{m}{100}\right) \times N - F_h}{f} \times i$$

$$或 P_m = U + \frac{N\left(1-\frac{m}{100}\right) - F_h}{f} \times i$$

式中，Pm为第m个百分位数，N为总频次，L为Pm所在组的下限，U为Pm所在组的上限，f为Pm所在组的次数，Fh为小于L的累积次数，Fh为大于U的累积次数，i为组距。

# 基础公用函数

- 百分位法等级划分

```
def divideRankByPercent(data, rank_content, rank_field, rank_descript_field,
                        min_value_field, max_value_field, nodata=0):
    """
    百分位划分等级
    :param data: list或tuple， array, 要分级的数据
    :param rank_content: dict,数据划分等级信息 {rank_field:[1,2,3,4], rank_descript_field:["高", "较高",  "较低", "低"],
                                                max_value_field:[100,90,70,30], min_value_field:[90,70,30,0]}
    :param rank_field: str, 划分等级的字段名, 包含在rank_content中
    :param rank_descript_field: str, 等级描述字段名, 包含在rank_content中
    :param min_value_field: str, 最小值字段, 包含在rank_content中
    :param max_value_field: str,最大值字段, 包含在rank_content中
    :param nodata: float, 无效值
    :return:
    """
    index_data = np.array(data)
    # 划分等级的个数
    rank_numbers = len(rank_content[rank_field])
    new_data = index_data[index_data != nodata]
    # new_data = index_data[(index_data != -999) & (index_data != 0)]
    if new_data.size<=1:
        rankdata = np.full(index_data.shape, rank_numbers, dtype=np.int)
        descriptdata = np.array([rank_content[rank_descript_field][rank_numbers - 1]] * (index_data.size), dtype='<U10')
    else:
        if (min_value_field in rank_content) & (max_value_field in rank_content):
            rankdata = np.full(index_data.shape, rank_numbers, dtype=np.int)
            descriptdata = np.array([rank_content[rank_descript_field][rank_numbers - 1]] * (index_data.size), dtype='<U10')
            for i in range(rank_numbers):
                rank = int(rank_content[rank_field][i])
                rank_descript = rank_content[rank_descript_field][i]
                rank_min = rank_content[min_value_field][i]
                rank_max = rank_content[max_value_field][i]
                if (rank_min is None) & (rank_max is not None):
                    max_value = np.nanpercentile(new_data, int(rank_max))
                    rankdata[(index_data < max_value)] = rank
                    descriptdata[(index_data < max_value)] = rank_descript
                if (rank_min is not None) & (rank_max is None):
                    min_value = np.nanpercentile(new_data, int(rank_min))
                    rankdata[(index_data >= min_value)] = rank
                    descriptdata[(index_data >= min_value)] = rank_descript
                if (rank_min is None) & (rank_max is None):
                    pass
                if (rank_min is not None) & (rank_max is not None):
                    max_value = np.nanpercentile(new_data, int(rank_max))
                    min_value = np.nanpercentile(new_data, int(rank_min))
                    if rank_max >= 100:
                        rankdata[(index_data >= min_value) & (index_data <= max_value)] = rank
                        descriptdata[(index_data >= min_value) & (index_data <= max_value)] = rank_descript
                    else:
                        rankdata[(index_data >= min_value) & (index_data < max_value)] = rank
    return rankdata, descriptdata
```

# 基础公用函数

## ➢ 百分位法等级划分

```
data = np.array([0.1, 0.12, 0.4, 0.6, 0.7, 0.4, 0.45, 0.7, 0.4, 0.32, 0.32,0.22, 0.45, 0.67])
rank_content = {"rank_field":[1,2,3,4],
                "rank_descript_field":["高", "较高",  "较低", "低"],
                "max_value_field":[100,90,70,30],
                "min_value_field":[90,70,30,0]}
rankdata, descriptdata = divideRankByPercent(data, rank_content, "rank_field", "rank_descript_field",
                                             "min_value_field", "max_value_field", nodata=0)
print(data)
print(rankdata)
print(descriptdata)
```

**函数调用**

```
[0.1  0.12 0.4  0.6  0.7  0.4  0.45 0.7  0.4  0.32 0.32 0.22 0.45 0.67]
[4 4 3 2 1 3 3 1 3 3 3 4 3 2]
['低' '低' '低' '低' '高' '低' '低' '高' '低' '低' '低' '低' '低' '低']
```

**返回结果（等级值及等级描述）**

# 基础公用函数

## ➢ 克里金插值

克里金插值法是一种基于统计学的插值方法，又称空间局部插值法，原理是利用区域化变量为基础，以变异函数为基本工具，对未知样点进行线性无偏、最优化估计。主要包括计算样本变异函数、根据变异函数对待估计数据建模、利用所建模型进行克里金插值估计和估计方差四大部分。

理论变异函数的基本公式为：$r(x,h)=\frac{1}{2}E\{[Z(x)-Z(x+h)]^2\}$

$Z(x)$和$Z(x+h)$分别是在点x、x+h处的观测值，h为步长；$\{[Z(x)-Z(x+h)]^2\}$为方差。

理论变异函数模型有：线性模型、指数模型、球面模型等。讲过大量实验，系统拟选取球状模型作为理论变异函数：

$$\gamma(h)=\begin{cases}0 & h=0\\ c_0+c\left(\frac{3}{2}\frac{h}{a}-\frac{1}{2}\frac{h^3}{a^3}\right) & 0\le h\le a\\ c_0+c & h>a\end{cases}$$

实验变异函数r(h)反映了区域化变量的空间自相关性：$r(h)=\frac{1}{2N(h)}\sum_{i=1}^{N(h)}[Z(x_i)-Z(x_i+h)]^2$

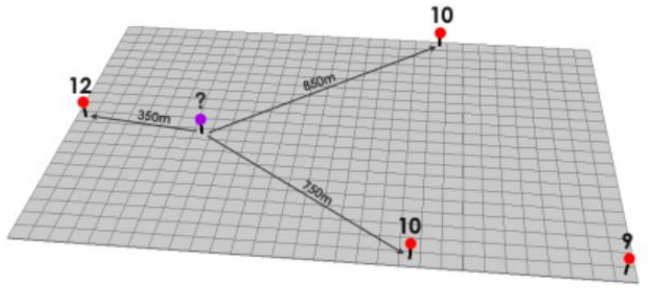
其中，h为监测点之间的空间间隔距离；N(h)为距离等于h的点对数；$Z(x_i)$为处于点$x_i$处变量的观测值；$Z(x_i$ +h)为与点偏离h处变量的实测值。

假设$x_0$为未观测点，$x_i(i=1,2,\dots,N)$为其周围的观测点，Z表示计算的区域化变量，则对$x_0$处某个区域化变量的估计值可以写为：$Z(x_0)=\sum_{i=1}^{n}\lambda_i Z(x_i)$

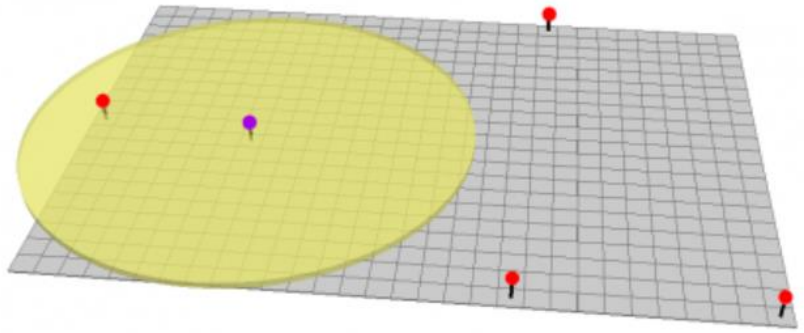
式中$\lambda_i$表示权重，可由理论变异函数求得，为了保证是无偏估计，要求$\sum_{i=1}^{n}\lambda_i=1$。

# 基础公用函数

## ➢ 克里金插值




插值过程中指定了搜索半径和已知点最少个数。若搜索半径内的已知点数大于最少个数，则采用搜索到的已知点；若小于，则采用离插值点最近的设定最少个数已知点



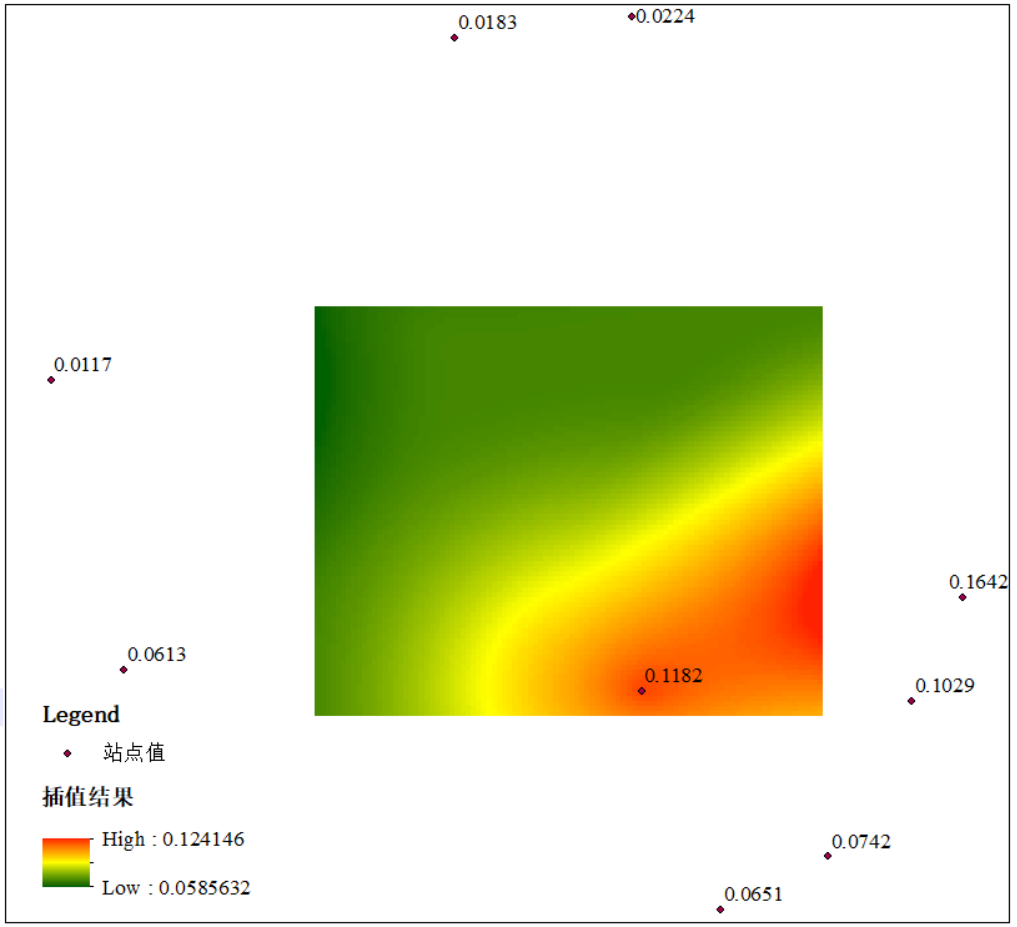
```
def krigeInterpolate(data=None, latdata=None, londata=None,outfile=None,
                     boundary_range=None, dst_epsg=None,dst_rows=None, dst_cols=None,
                     radius_dist=1.0, min_num=5,min_value=None, max_value=None,first_size=200):
    """
    try:
        if min_num > data.size:
        if (dst_cols>first_size) | (dst_rows>first_size):
        else:
        x_min = boundary_range[0]
        y_max = boundary_range[3]
        x_cell = abs(boundary_range[2] - boundary_range[0]) / cols
        y_cell = -1 * abs(boundary_range[3] - boundary_range[1]) / rows
        outdata = np.full((rows, cols), 0, dtype=np.float16)
        driver = gdal.GetDriverByName("MEM")
        out_ds = driver.Create("", cols, rows, 1, gdal.GDT_Float32)
        out_ds.SetGeoTransform((x_min, x_cell, 0, y_max, 0, y_cell))
        out_srs = osr.SpatialReference()
        out_srs.ImportFromEPSG(dst_epsg)
        out_ds.SetProjection(out_srs.ExportToWkt())
        xmaps = np.linspace(x_min + x_cell / 2.0, (x_min + cols * x_cell) - x_cell / 2.0, cols, endpoint=True)
        ymaps = np.linspace(y_max + (y_cell) / 2.0, (y_max + rows * (y_cell)) - (y_cell) / 2.0, rows, endpoint=True)
        dist = radius_dist ** 2  # 检索范围，经纬度
        for i in range(cols):
            lon = xmaps[i]
            dist_x = (londata - lon) ** 2
            for j in range(rows):
                lat = ymaps[j]
                dist_y = (latdata - lat) ** 2
                dist_point = dist_x + dist_y # 两点之间距离平方
                array_index = dist_point < dist
                if np.sum(array_index) < min_num:
                lons = londata[array_index]
                lats = latdata[array_index]
                values = data[array_index]
                if np.sum(values == values[0]) == values.size:
                    outdata[j, i] = values[0]
                else:
                    ok_obj = OrdinaryKriging(lons, lats, values,variogram_model="spherical", variogram_parameters=None,
                                             weight=True,coordinates_type="geographic", nlags=6)
                    z, ss = ok_obj.execute("grid", lon, lat, backend="vectorized")
                    outdata[j, i] = np.array(z)[0][0]
        if min_value is None:
        else:
        if max_value is None:
        else:
        out_band = out_ds.GetRasterBand(1)
        out_band.WriteArray(outdata)
        if outfile is not None:
        else:
        return outds
    finally:
        ds = None
        out_ds = None
```

# 基础公用函数

## ➢ 克里金插值

```
cols = 98
rows = 79
boundary_range = (114.99166666658346, 39.29166666678327,
                  115.80833333281711, 39.95000000029981)
data = np.array([0.1029, 0.0224, 0.0613, 0.0742, 0.0117,
                 0.0651, 0.1182, 0.0183,0.1642])
latdata = np.array([39.3166, 40.4166, 39.3666, 39.0666, 39.8333,
                    38.9811, 39.3333,40.3833, 39.4833])
londata = np.array([115.95, 115.5, 114.6833, 115.8166, 114.5666,
                    115.6433,115.5166, 115.2166, 116.0333])
outfile = r"D:\HT_Project\lyb_alg_code\train_code\krige_test.tif"
krigeInterpolate(data=data, latdata=latdata, londata=londata,
                 outfile=outfile,boundary_range=boundary_range,
                 dst_epsg=4490, dst_rows=rows, dst_cols=cols,
                 radius_dist=1.0, min_num=10,
                 min_value=0, max_value=1)
print(1)
```

函数调用




克里金插值结果

# 基础公用函数

## ➢ 反距离权重插值

反距离权重 (IDW) 插值：彼此距离较近的事物要比彼此距离较远的事物更相似。当为任何未测量的位置预测值时，反距离权重法会采用预测位置周围的测量值。与距离预测位置较远的测量值相比，距离预测位置最近的测量值对预测值的影响更大。反距离权重法假定每个测量点都有一种局部影响，而这种影响会随着距离的增大而减小。由于这种方法为距离预测位置最近的点分配的权重较大，而权重却作为距离的函数而减小，因此称之为反距离权重法。

反距离权重计算过程：

1）计算算未知点到所有点的距离

$$d_i = \sqrt[2]{(x - x_i)^2 + (y - y_i)^2}$$

2）计算权重

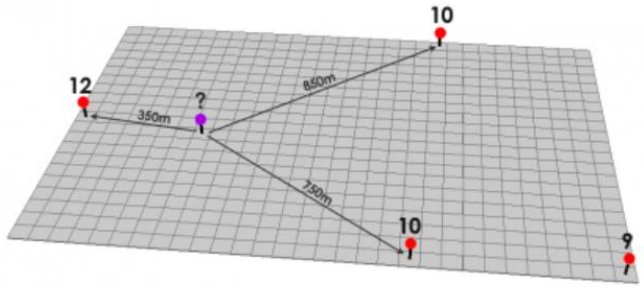
$$w_i = \frac{1/d_i}{\sum_1^n 1/d_i}$$

3）计算插值点的值

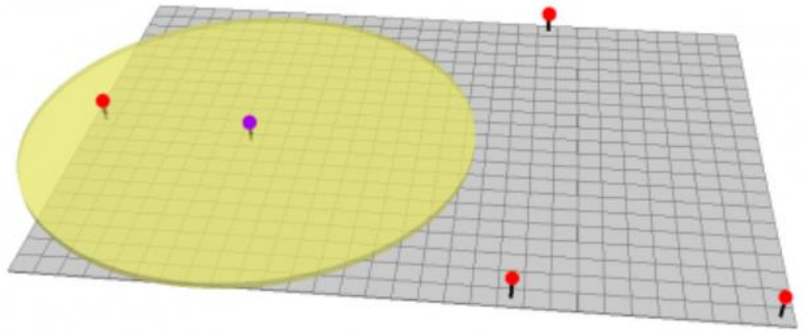
$$Z_0 = \sum_{i=1}^{i=n} w_i * Z(X_i, Y_i)$$

# 基础公用函数

## ➢ 反距离权重插值




插值过程中指定了搜索半径和已知点最少个数。若搜索半径内的已知点数大于最少个数，则采用搜索到的已知点；若小于，则采用离插值点最近的设定最少个数已知点



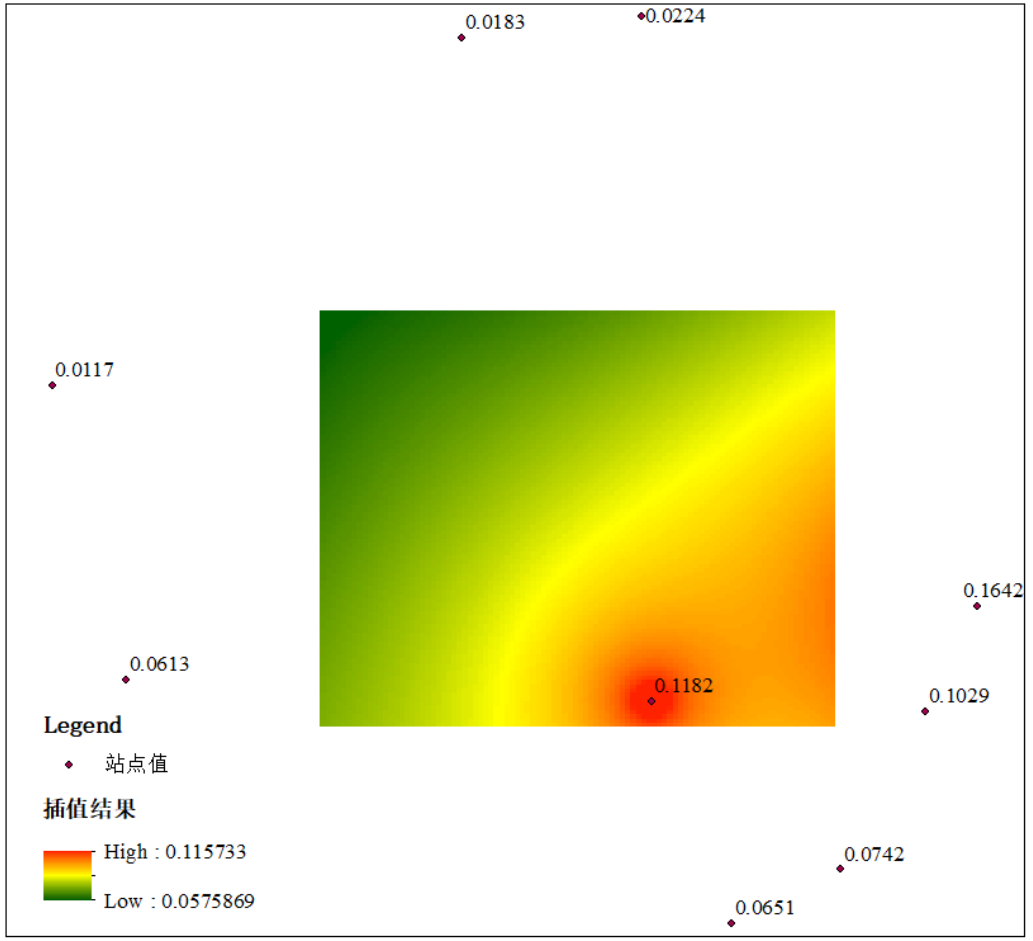
```
def idwInterProcess(data=None, latdata=None, londata=None,outfile=None,
                    boundary_range=None, dst_epsg=None,dst_rows=None, dst_cols=None,
                    radius_dist=1.0, min_num=5,min_value=None, max_value=None):
    """
    try:
        cols = int(dst_cols)
        rows = int(dst_rows)
        if min_num > data.size:
            min_num = data.size
        x_min = boundary_range[0]
        y_max = boundary_range[3]
        x_cell = abs(boundary_range[2] - boundary_range[0]) /cols
        y_cell = -1 * abs(boundary_range[3] - boundary_range[1]) / rows
        outdata = np.full((rows, cols), 0, dtype=np.float32)
        if outfile is None:
        else:
        out_ds.SetGeoTransform((x_min, x_cell, 0, y_max, 0, y_cell))
        out_srs = osr.SpatialReference()
        out_srs.ImportFromEPSG(dst_epsg)
        out_ds.SetProjection(out_srs.ExportToWkt())
        xmaps = np.linspace(x_min + x_cell / 2.0, (x_min + cols * x_cell) - x_cell / 2.0, cols, endpoint=True)
        ymaps = np.linspace(y_max + (y_cell) / 2.0, (y_max + rows * (y_cell)) - (y_cell) / 2.0, rows, endpoint=True)
        dist = radius_dist ** 2  # 检索范围，经纬度
        for i in range(cols):
            lon = xmaps[i]
            dist_x = (londata - lon) ** 2
            for j in range(rows):
                lat = ymaps[j]
                dist_y = (latdata - lat) ** 2
                dist_point = dist_x + dist_y
                array_index = dist_point < dist
                if np.sum(array_index) < min_num:
                    dist_copy = dist_point.copy()
                    dist_copy.sort()
                    array_index = dist_point <= dist_copy[min_num-1]
                lons = londata[array_index]
                lats = latdata[array_index]
                values = data[array_index]
                if np.sum(values==values[0]) == values.size:
                    outdata[j, i] = values[0]
                else:
                    dist_all = 1.0 / np.sqrt((lat - lats) ** 2 + (lon - lons) ** 2)
                    dist_sum = np.sum(dist_all)
                    idw_v = np.sum((dist_all / dist_sum) * values)
                    outdata[j, i] = idw_v
        if min_value is None:
        else:
        if max_value is None:
        else:
        out_band = out_ds.GetRasterBand(1)
        out_band.WriteArray(outdata)
        return out_ds
    finally:
        ds = None
```

# 基础公用函数

## ➢ 反距离权重插值

```
cols = 98
rows = 79
boundary_range = (114.99166666658346, 39.29166666678327,
                  115.80833333281711, 39.95000000029981)
data = np.array([0.1029, 0.0224, 0.0613, 0.0742,
                 0.0117, 0.0651, 0.1182, 0.0183,0.1642])
latdata = np.array([39.3166, 40.4166, 39.3666, 39.0666,
                    39.8333, 38.9811, 39.3333,40.3833, 39.4833])
londata = np.array([115.95   , 115.5    , 114.6833, 115.8166,
                    114.5666, 115.6433,115.5166, 115.2166, 116.0333])
outfile = r"D:\HT_Project\lyb_alg_code\train_code\idw_test.tif"
idwInterProcess(data=data, latdata=latdata, londata=londata,
                    outfile=outfile,boundary_range=boundary_range,
                    dst_epsg=4490, dst_rows=rows, dst_cols=cols,
                    radius_dist=1.0, min_num=10, min_value=0, max_value=1)
print(1)
```

函数调用




反距离插值结果

# 基础公用函数

## ➢ 栅格转矢量函数

```
def polygonizeTheRaster(infile, outfile, dst_nodata=0, dst_fieldname='class'):
    """
    栅格转矢量
    :param infile: str, 栅格文件
    :param outfile: str, 结果文件
    :param dst_nodata: int,数据中的无效值
    :param dst_fieldname: str, 转换字段
    :return:
    """
    try:
        ds = gdal.Open(infile)
        srcband = ds.GetRasterBand(1)
        maskband = srcband.GetMaskBand()
        drv = ogr.GetDriverByName('ESRI Shapefile')
        dst_ds = drv.CreateDataSource(outfile)
        prj = osr.SpatialReference()
        prj.ImportFromWkt(ds.GetProjection())
        dst_layername = os.path.splitext(os.path.basename(infile))[0]
        dst_layer = dst_ds.CreateLayer(dst_layername, srs=prj, geom_type=ogr.wkbPolygon)
        fd = ogr.FieldDefn(dst_fieldname, ogr.OFTInteger)
        dst_layer.CreateField(fd)
        options = []
        # 参数  输入栅格图像波段\掩码图像波段、矢量化后的矢量图层、需要将DN值写入矢量字段的索引、算法选项、进度条回调函数、进度条参数
        gdal.Polygonize(srcband, maskband, dst_layer,  dst_nodata, options)
    finally:
        ds = None
```

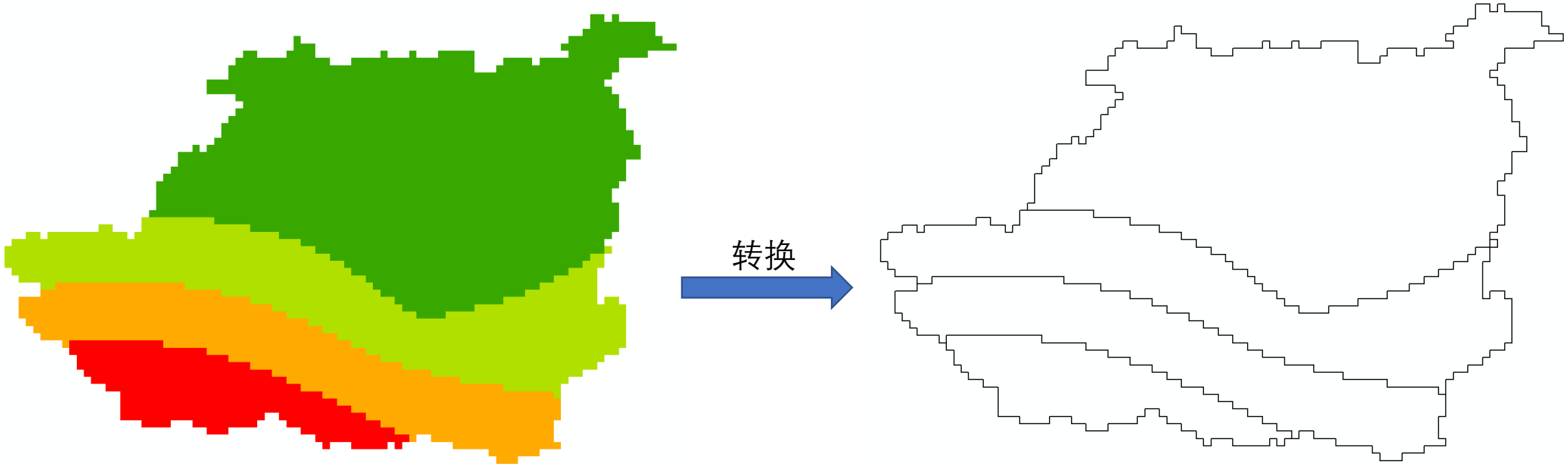
栅格转矢量函数

# 基础公用函数

## ➢ 栅格转矢量函数

```
infile = r"D:\HT_Project\lyb_alg_code\train_code\inputfile.tif"
outfile = r"D:\HT_Project\lyb_alg_code\train_code\outfile.shp"
dst_nodata = 0
dst_fieldname = 'class'
polygonizeTheRaster(infile, outfile, dst_nodata=dst_nodata, dst_fieldname=dst_fieldname)
```

**函数调用**



# 灾害风险评估与区划实践（以高温为例）

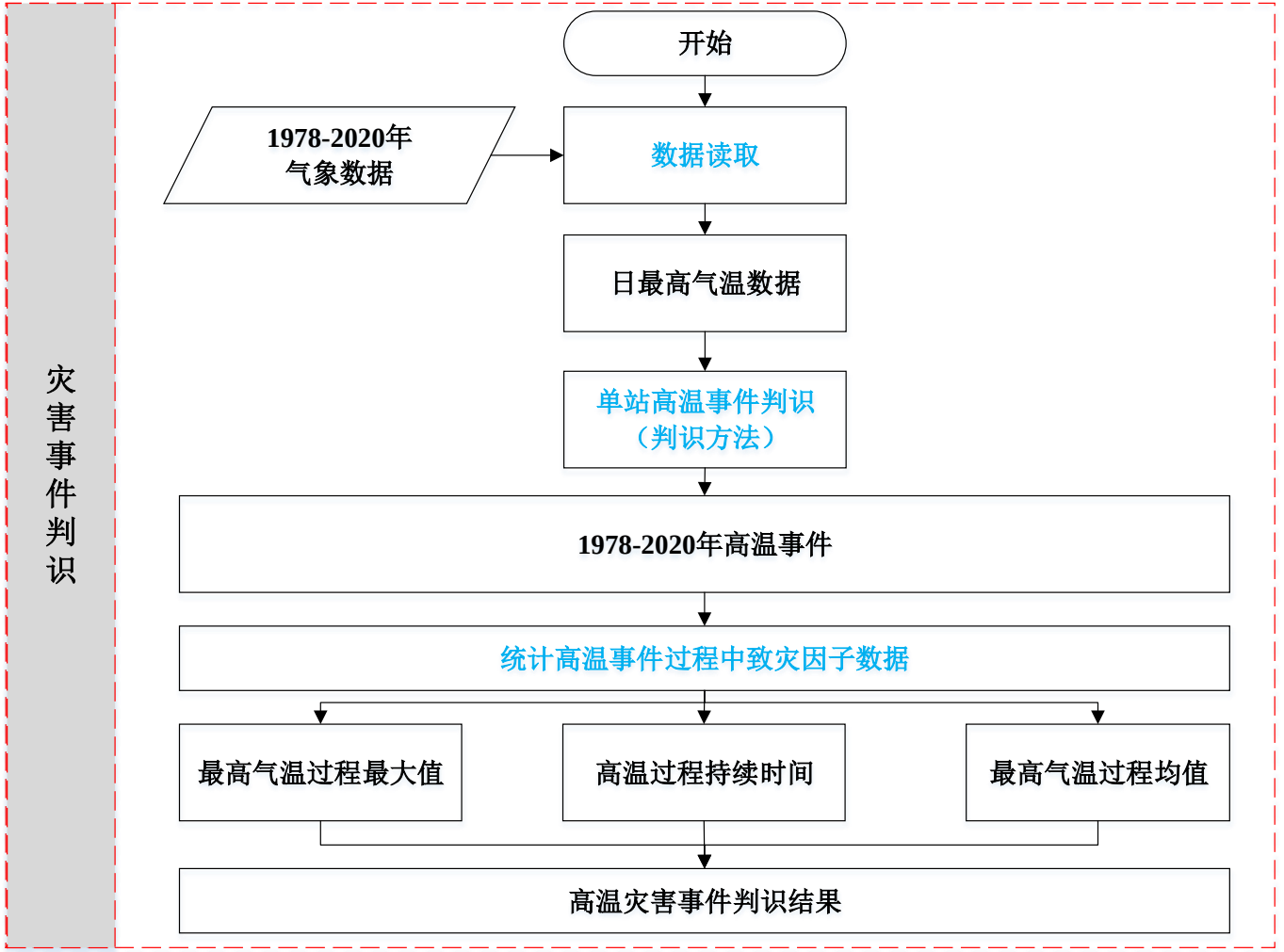
## ➢ 高温灾害事件判识

高温灾害事件判识主要过程

1）数据读取

2）高温事件判识

3）统计致灾因子数据



# 灾害风险评估与区划实践（以高温为例）

## ➢ 高温灾害事件判识

```python
def readData(self, file):
    """
    读取数据
    :param file: str, csv文件路径
    :return:
    """
    try:
        csv_ds = pd.read_csv(file, dtype=np.str, encoding="utf-8")
    except:
        csv_ds = pd.read_csv(file, dtype=np.str, encoding="gbk")
    stn_info = csv_ds.iloc[0:1, 2:]
    csv_ds = csv_ds.iloc[1:, 0:2]
    # 根据时间截取数据
    obj = DataFrameTool()
    csv_ds = obj.rangeSelect(csv_ds, Station.DateTime, self.start_time, self.end_time)
    data = csv_ds[Station.Tem_field].astype(np.float32)
    # 填充值修改成-999
    data = data.replace([999999], -999)
    return data, stn_info
```

**1.数据读取**

```python
def judgeTemProcess(self, data, tem_intensity, length_days):
    """
    判别高温事件的开始和结束日期
    :param data:dataframe,数据
    :param tem_intensity: float,高温阈值
    :param length_days: int, 高温持续天数
    :return:
    """
    # high_temp_events_out = {}
    events_info = {}
    events_start, events_end, events_last = self.highTempEvents(data, tem_intensity, length_days)
    events_info.update({Station.Start_Time: events_start,
                        Station.End_Time: events_end,
                        Station.Time_Length: events_last})
    # high_temp_events_out.update({stn: events_info})
    return events_info
```

**2.高温事件识别**

```python
def highTempInfoOut(self, events_seq_out, data, outpath=None, stn_info=None, outdf=None):
    """
    逐个高温统计事件写入文本，或输出
    :param events_seq_out: dict, 事件过程
    :param data: dataFrame, 数据
    :param outpath: str,输出路径
    :param stn_info: dict, 站点信息字典
    :param outdf: dataframe, 输出
    :return:
    """
    events_start_stn = events_seq_out[Station.Start_Time]
    events_end_stn = events_seq_out[Station.End_Time]
    events_last_stn = events_seq_out[Station.Time_Length]
    bins = list(zip(events_start_stn, events_end_stn))
    events_vals_max = np.zeros_like(events_start_stn, dtype=np.float32)
    events_vals_mean = np.zeros_like(events_start_stn, dtype=np.float32)
    events_date_start = (data.index[events_start_stn]).astype(np.str)
    events_date_end = (data.index[events_end_stn]).astype(np.str)
    if events_start_stn != []:...
    else:...
    stn_info = stn_info.to_dict(orient="list")
    if Station.Province not in stn_info:
        province_name = ""
    else:
        province_name = list(stn_info[Station.Province])[0]
```

**3.高温事件致灾因子统计与结果输出**

# 灾害风险评估与区划实践（以高温为例）

## ➢ 高温危险性评估与区划

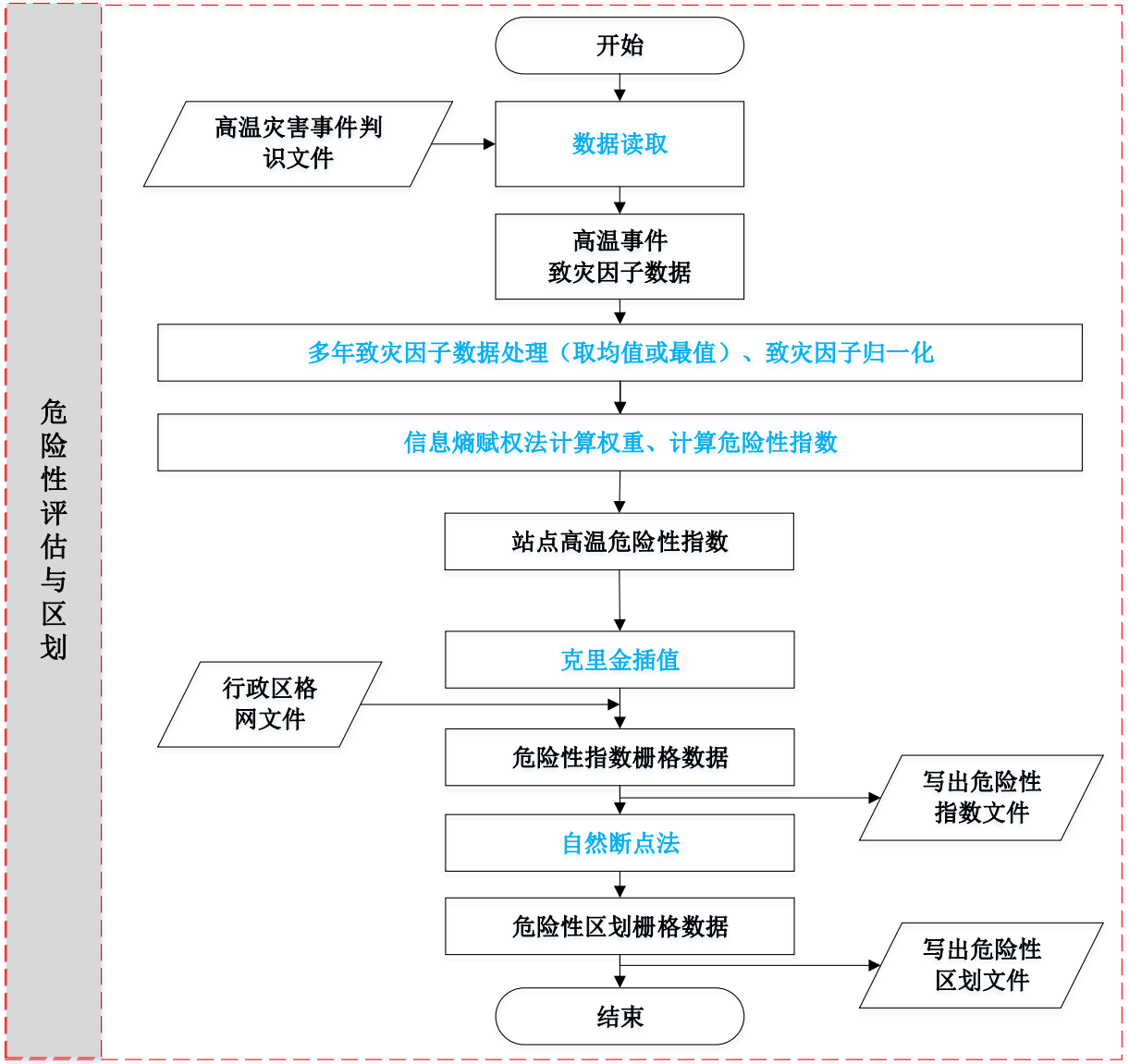
高温危险性评估与区划主要过程

1）读取致灾因子数据和处理

2）高温危险性指数计算

3）站点高温危险性指数克里金插值

4）自然断点法得到危险性区划



# 灾害风险评估与区划实践（以高温为例）

## ➢ 高温危险性评估与区划

1 数据读取和致灾因子处理

```
def calculateHazardsData(filelist=None, start_time=None, end_time=None):
    """..."""
    year_length = int(end_time[0:4]) - int(start_time[0:4])
    outdf = pd.DataFrame([])
    for file in filelist:
        df_ = pd.DataFrame([])
        try:
            df = pd.read_csv(file, dtype=np.str, encoding="utf-8")
        except:
            df = pd.read_csv(file, dtype=np.str, encoding="gbk")
        df_[Station.Station_Name] = [df[Station.Station_Name].values[0]]
        df_[Station.Province] = [df[Station.Province].values[0]]
        df_[Station.Cnty] = [df[Station.Cnty].values[0]]
        df_[Station.Station_Id_d] = [df[Station.Station_Id_d].values[0]]
        df_[Station.Lat] = [float(df[Station.Lat].values[0])]
        df_[Station.Lon] = [float(df[Station.Lon].values[0])]
        df_[Station.Alti] = [float(df[Station.Alti].values[0])]
        obj = DataFrameTool()
```

2 高温危险性计算

```
def calculateDangerIndex(data_dict=None, w_method=None, w_list=None, fillvalue=-999):
    """..."""
    data = np.array([data_dict[Station.Time_Length], data_dict[Station.Max_Tem],data_
    # 数据归一化
    index_array = np.isnan(data)
    if np.sum(index_array) >= data.size-1:
        # 当没有事件的时候，指数为0
        data_dict[Station.Danger_Index] = np.full(data.shape[1], 0, dtype=np.int16)
    else:
        data_normed = ArrayTool.dataNormalProcess(data)
        # 获取权重
        if w_method == Station.OWM:
            wi = w_list
        else:
            wi = ArrayTool.calcWeights(data_normed)
        # 计算危险性数据
        d_data = np.dot(data_normed, wi)
        d_data[np.isnan(d_data)] = fillvalue
        data_dict[Station.Danger_Index] = np.around(d_data, 4)
    return data_dict
```

```
outds = obj.krigeInterpolate(data=np.array(data_dict[Station.Danger_Index]),
                             latdata=np.array(data_dict[Station.Lat]),
                             londata=np.array(data_dict[Station.Lon]),
                             outfile=None,
                             boundary_range=boundary_range,
                             dst_epsg=dst_epsg,
                             dst_rows=rows, dst_cols=cols,
                             radius_dist=1.0, min_num=10,
                             min_value=0, max_value=1, first_size=200)
```

**3 站点高温危险性指数克里金插值**

```
# 危险性栅格数据等级划分
obj.rasterDivideRank(infile=dangerfile, outfile=rankfile, method=rank_method,
                     rank_field=Station.rank_field,
                     rank_content=rank_content, min_value_field=Station.MinValue,
                     max_value_field=Station.MaxValue, nodata=-999, dst_nodata=0)
obj.polygonizeTheRaster(rankfile, outshpfile, dst_nodata=0, dst_fieldname='class')
```

**4自然断点法得到危险性区划**

# 灾害风险评估与区划实践（以高温为例）

## ➢ 高温风险评估与区划

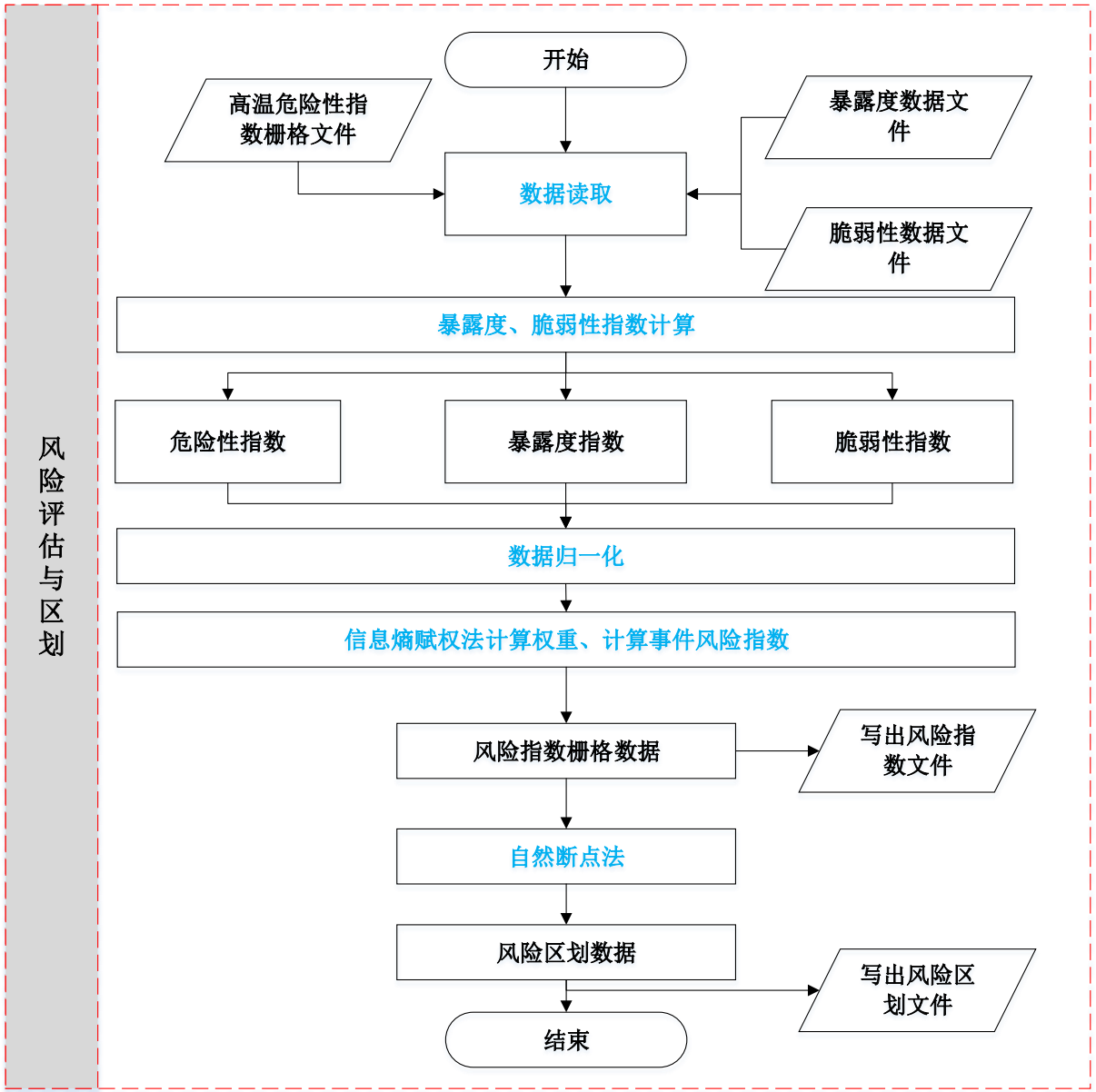
高温风险评估与区划主要过程

1）数据读取及数据处理

2）数据归一化

3）计算风险指数

4）自然断点法得到风险区划



# 灾害风险评估与区划实践（以高温为例）

## ➢ 高温风险评估与区划

**1 数据读取和处理**

```
# 1)获取承灾体数据
filelist = getFiles(body_dir, start_time=start_time, end_time=end_time)
#2)处理承灾体数据，得到年均值
body_ds = getYearAverage(filelist, danger_file, None)
# 数据归一化（数据归一化到0.5-1）
#3)获取脆弱性数据
filelist = getFiles(vul_dir, start_time=start_time, end_time=end_time
#4)处理脆弱性数据，得到年均值
if len(filelist)<=0:
    vul_ds = None
else:
    vul_ds = getYearAverage(filelist, danger_file, None)
```

**2 数据归一化**

```
obj = RasterTool()
body_ds = obj.dataNormal(infile=body_ds, outfile=None, nodata=-999)
vul_ds = obj.dataNormal(infile=vul_ds, outfile=None, nodata=-999)
# 数据归一化（数据归一化到0.5-1）
danger_ds = obj.dataNormal(infile=danger_file, outfile=None, nodata=-999)
```

**3 计算风险指数**

```
# 计算风险指数
caculateRiskIndex(danger_ds, body_ds, vul_ds,
                  outfile=raster_path_index, nodata=-999,
                  w_method=w_method, w_list=w_list)
# 等级划分
```

**4 自然断点法得到风险区划**

```
# 等级划分
obj = RasterTool()
obj.rasterDivideRank(infile=raster_path_index, outfile=raster_path,
                     method=rank_method, rank_field=Station.rank_field,
                     rank_content=rank_content, min_value_field=Station.MinValue,
                     max_value_field=Station.MaxValue, nodata=-999, dst_nodata=0)
obj.polygonizeTheRaster(raster_path, poly_shp, dst_nodata=0, dst_fieldname='class')
```

# 命令行操作流程

# 极端气候事件风险模拟分析子系统

目录结构

脚本文件

参数设置

命令提交

运行过程

结果查看

分系统目录： /data/luoyong/H03

子系统目录：

算法脚本目录：/data/luoyong/H03/algorithm/JDQH

输入数据目录：/data/luoyong/H03/inputdata/JDQH

输出结果目录：/data/luoyong/H03/product/JDQH

模块目录：

算法脚本目录：/data/luoyong/H03/algorithm/JDQH/${模块缩写}

输入数据目录：/data/luoyong/H03/inputdata/JDQH/${模块缩写}

输出结果目录：/data/luoyong/H03/product/JDQH/${模块缩写}

| 模块 | 缩写 |
| --- | --- |
| 极端高温事件风险分析模块 | JDHW |
| 极端低温事件风险分析模块 | JDDW |
| 气象干旱风险分析模块 | QXGH |
| 强降水事件风险分析模块 | JDJS |
| 台风风险分析模块 | TCFX |

# 极端气候事件风险模拟分析子系统

目录结构

**脚本文件**

参数设置

命令提交

运行过程

结果查看

各模块运行脚本：

提交脚本：run_${模块缩写}.sh
主程序脚本：${模块缩写}_Danger.sh　app.py　src/main_gw_stn_danger.py
制图脚本：${模块缩写}_plot.sh

```
#!/usr/bin/bash
source ~/.bashrc
#conda activate hazard
# 判断是否传递用户配置参数文件,不存在则终止程序
if [ -z $1 ];then
    echo "未输入参数配置文件userconfig.json"
    exit 1
fi
# 程序记录开始时间
starttime=$(date '+%Y-%m-%d %H:%M:%S')
MODULE="JDGW"
MODULENAME="极端高温事件风险分析"
#接收日志文件输出路径
log_file=$(cat $1 | jq .algorithmLogFile -r)
json_file=$(cat $1 | jq .resultJsonFile -r)
flow_file=$(cat $1 | jq .algorithmFlowFile -r)
userconfig=$1
#检查日志和json文件是否已经存在，存在则删除
if [ -f $log_file ];then
    rm $log_file
fi
if [ -f $json_file ];then
    rm $json_file
fi
if [ -f $flow_file ];then
        rm $flow_file
fi
```

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

"""
调用高温危险性代码
@Version<1.0.0> 2020-07-15 Created by LYB
@Version<1.0.1> 2021-01-10 Created by LYB
"""

version = "1.0.1"
import traceback
import time
import os
import sys
from arspy import glo
from arspy.id import product, period, fmt
# PROJ_LIB_Path = os.path.join(os.path.dirname(sys.executable), "Lib\site
roj")
# os.environ["PROJ_LIB"] = PROJ_LIB_Path
# rootPath = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
# sys.path.append(rootPath)
from src.common_tool.json_tool import JsonTool
from src.common_tool.logging_tool import Logger
from src.gw_stn_danger import dangerEvaluate
from src.gw_stn_danger import dangerEvaluate2
from src.common_tool.file_tool import FileTool
```

# 极端气候事件风险模拟分析子系统

参数文件：useconfig.json

目录结构

脚本文件

**参数设置**

命令提交

运行过程

结果查看

```
{
"analyseStrtYear": 19780101,
"analyseEndYear": 20201231,
"regionName": "中国",
"regionCode": "000000",
"inputDataDir": "/data/luoyong/H03/inputdata/JDQH/JDGW",
"productDir": "/data/luoyong/H03/product/JDQH/JDGW",
"resultJsonFile": "/data/luoyong/H03/product/JDQH/JDGW/output.json",
"algorithmLogFile": "/data/luoyong/H03/product/JDQH/JDGW/output.log",
"algorithmFlowFile": "/data/luoyong/H03/product/JDQH/JDGW/outputFlow.json"
```

| 参数名称 | 参数描述 |
| --- | --- |
| analyseStrtYear | 分析起始年 |
| analyseEndYear | 分析终止年 |
| regionName | 分析区域 |
| regionCode | 分析区域行政代码 |
| inputDataDir | 输入数据目录 |
| productDir | 输出目录 |
| resultJsonFile | 输出结果json |
| algorithmLogFile | 输出日志json |
| algorithmFlowFile | 输出流程json |

# 极端气候事件风险模拟分析子系统

目录结构

脚本文件

参数设置

**命令提交**

运行过程

结果查看

分析模块运行提交：

bash run_${模块缩写}.sh useconfig.json

# 极端气候事件风险模拟分析子系统

目录结构

脚本文件

参数设置

命令提交

运行过程

结果查看

分析模块运行:

屏幕实时打印结果

```
(bdfx) [luoyong@login01 JDGW]$ bash run_JDGW.sh /data/luoyong/H03/algorithm/JDQH/JDGW/userconfi
g.json
2022-05-25 15:13:23 : 主程序初始化
2022-05-25 15:13:23 : 完成参数解析
2022-05-25 15:13:23 : 执行主程序算法 GWFX_Danger.sh
2022-05-25 15:15:53 : 主程序算法 GWFX_Danger.sh 执行完毕,耗时2m30.296s
2022-05-25 15:15:53 : 执行主程序绘图算法 GWFX_plot.sh
2022-05-25 15:16:09 : 主程序算法 GWFX_plot.sh 执行完毕,耗时0m15.720s
2022-05-25 15:16:09 : 输出产品路径
2022-05-25 15:16:09 : 输出json路径 /data/luoyong/H03/product/JDQH/JDGW/output.json
2022-05-25 15:16:09 : 输出日志路径 /data/luoyong/H03/product/JDQH/JDGW/output.log
2022-05-25 15:16:09 : 输出算法执行流程路径 /data/luoyong/H03/product/JDQH/JDGW/outputFlow.json
```

```
(bdfx) [luoyong@login01 JDGW]$ bash run_JDGW.sh /data/luoyong/H03/algorithm/JDQH/JDGW/userconfi
g.json
2022-05-25 15:11:25 : 主程序初始化
2022-05-25 15:11:25 : 完成参数解析
2022-05-25 15:11:25 : 执行主程序算法 GWFX_Danger.sh
2022-05-25 15:11:33 : 主程序算法 GWFX_Danger.sh 执行失败,错误详见/data/luoyong/H03/algorithm/JD
QH/JDGW/GW/GWFX_Danger.log
```

# 极端气候事件风险模拟分析子系统

目录结构

脚本文件

参数设置

命令提交

运行过程

**结果查看**

分析模块运行结果:

专题图直接查看