|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | | **文档编号**  **File No.** |  |
| **文档名称**  **File name** | **中文**  **Local** | C#编码规范 | **版本**  **Version** | **V 1.0** |
| **英文**  **English** | Code Specification for c# | **密级**  **Level** | **内部（INSIDE）** |
| **存放位置**  **Locate** |  | | **作者**  **Author** | **谭明珉** |

**C#编码规范**

**Code Specification for c#**

**(仅供内部使用 only for inside of bpnet)**

作者：谭明珉

日期：2019年12月09日

目录

**1规范目的·······································3**

**2适用范围·······································3**

**3代码注释·······································3**

**3.1代码注释约定······························3**

**3.2模块头部注释规范··························3**

**3.3方法注释规范······························4**

**3.4代码行注释规范····························6**

**3.5变量注释规范······························7**

**4命名规则·······································8**

**4.1命名的基本约定····························8**

**4.2各种标识符类型的命名约定··················9**

**5其他规范·······································10**

**5.1编程风格··································10**

**5.2资源释放··································13**

**5.3错误处理··································13**

**5.4其他······································14**

**1     规范目的**

1. 一个软件的生命周期中，80%的花费在于维护；
2. 几乎没有任何一个软件，在其整个生命周期中，均由最初的开发人员来维护；
3. 编码规范可以改善软件的可读性，可以让程序员尽快而彻底地理解新的代码。为了执行规范，每个软件开发人员必须一致遵守编码规范；
4. 使用统一编码规范的主要原因，是使应用程序的结构和编码风格标准化，以便于阅读和理解这段代码；
5. 好的编码约定可使源代码严谨、可读性强且意义清楚，与其它语言约定相一致，并且尽可能的直观。

**2     适用范围**

1. 本规范主要以C#为开发语言的规范；
2. 由于本规范是为撰写程序而设计，所以适用于一切有关程序撰写的工作事项。对于具体的每个项目，可能需要对之进行裁剪和补存。
3. 适用人员：.net方向开发；
4. 适用产品：以C#编写的程序。

**3     代码注释**

**3.1    代码注释约定**

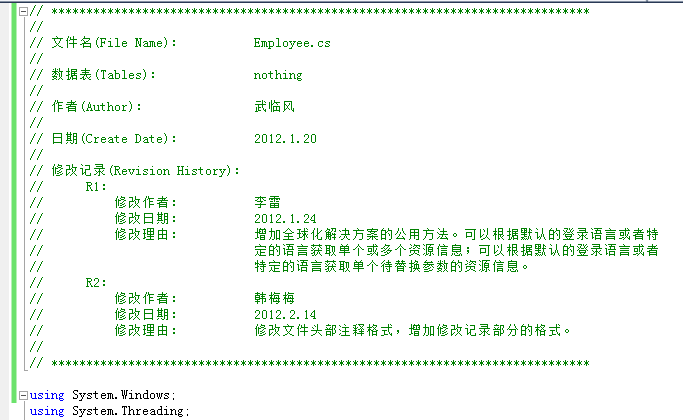
1. 所有的方法和函数都应该以描述这段代码的功能的一段简明注释开始（方法是干什么）。这种描述不应该包括执行过程细节（它是怎么做的），因为这常常是随时间而变的，而且这种描述会导致不必要的注释维护工作，甚至更糟—成为错误的注释。代码本身和必要的嵌入注释将描述实现方法。
2. 当参数的功能不明显且当过程希望参数在一个特定的范围内时，也应描述传递给过程的参数。被过程改变的函数返回值和全局变量，特别是通过引用参数的那些，也必须在每个过程的起始处描述它们。

**3.2     模块头部注释规范**

**以一个物理文件为单元的都需要有模块头部注释规范，例如：C#中的.cs文件**用于每个模块开头的说明，主要包括（**粗体字**为常用部分，其余为可选部分）

1. **文件名称(File Name)**： 此文件的名称
2. **功能描述(Description)**：   此模块的功能描述与大概流程说明
3. **数据表(Tables)**：             所用到的数据表，视图，存储过程的说明，如关系比较复杂，则应说明哪些是可擦写的，哪些表为只读的。
4. **作者(Author)**：
5. **日期(Create Date)**：
6. 参考文档(Reference)(可选)：          该档所对应的分析文档，设计文檔。
7. 引用(Using) (可选)﹕           开发的系统中引用其它系统的Dll、对象时，要列出其对应的出处，是否与系统有关﹙不清楚的可以不写﹚，以方便制作安装档。
8. **修改记录(Revision History)**：若档案的所有者改变，则需要有修改人员的名字、修改日期及修改理由。
9. **分割符：**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* (前后都要)

示例如下：



**3.3     方法注释规范**

　　1>  C# 提供一种机制，使程序员可以使用含有XML 文本的特殊注释语法为他们的代码编写文档。在源代码文件中，具有某种格式的注释可用于指导某个工具根据这些注释和它们后面的源代码元素生成XML。具体应用当中，类、接口、属性、方法必须有<Summary>节，另外方法如果有参数及返回值，则必须有<Param>及<Returns>节。示例如下：

　　　　/// <summary>

　　　　/// …

　　　　/// </summary>

　　　　/// <param name=””></param>

　　　　/// <returns></returns>

　　2>  事件不需要头注解，但包含复杂处理时（如：循环/数据库操作/复杂逻辑等），应分割成单一处理函数，事件再调用函数。

　　3>  所有的方法必须在其定义前增加方法注释。

　　4>  方法注释采用 /// 形式自动产生XML标签格式的注释。

|  |  |  |
| --- | --- | --- |
| **标记** | **说明** | **备注** |
| <c> | 提供了一种将说明中的文本标记为代码的方法 |  |
| <code> | 提供了一种将多行指示为代码的方法 |  |
| <example> | 可以指定使用方法或其他库成员的示例。一般情况下，这将涉及到 <code> 标记的使用。 |  |
| <exception> | 对可从当前编译环境中获取的异常的引用。 |  |
| <include> | 得以引用描述源代码中类型和成员的另一文件中的注释。 |  |
| <list> | 用于定义表或定义列表中的标题行。 |  |
| <para> | 用于诸如<summary>、<remarks> 或 <returns> 等标记内，使您得以将结构添加到文本中。 |  |
| <param> | 应当用于方法声明的注释中，以描述方法的一个参数。 |  |
| <paramref> | 提供了一种指示词为参数的方法。 |  |
| <permission> | 得以将成员的访问记入文档。 |  |
| <remarks> | 用于添加有关某个类型的信息，从而补充由 <summary> 所指定的信息。 |  |
| <returns> | 应当用于方法声明的注释，以描述返回值。 |  |
| <see> | 得以从文本内指定链接。 |  |
| <seealso> | 对可以通过当前编译环境进行调用的成员或字段的引用。 |  |
| <summary> | 应当用于描述类型或类型成员。 |  |
| <value> | 得以描述属性。 |  |

　　　　示例图如下：



　　5>  在公用类库中的公用方法需要在一般方法的注释后添加作者、日期及修改记录信息，统一采用XML标签的格式加注，标签如下：

　　　　<Author></Author> 作者

　　　　<CreateDate></CreateDate> 建立日期

　　　　<RevisionHistory> 修改记录

　　　　<ModifyBy></ModifyBy>    修改作者

　　　　<ModifyDate></ModifyDate>   修改日期

　　　 <ModifyReason></ModifyReason>   修改理由

　　　　</RevisionHistory>

　　　　<LastModifyDate></LastModifyDate> 最后修改日期

　　6>  一个代码文件如果是由一人编写，则此代码文件中的方法无需作者信息，非代码文件作者在此文件中添加方法时必须要添加作者、日期等注释。

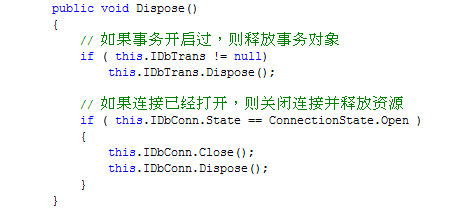
　　7>  修改任何方法，必须要添加修改记录的注释。

**3.4     代码行注释规范**

　　1>  如果处理某一个功能需要很多行代码实现，并且有很多逻辑结构块，类似此种代码应该在代码开始前添加注释，说明此块代码的处理思路及注意事项等

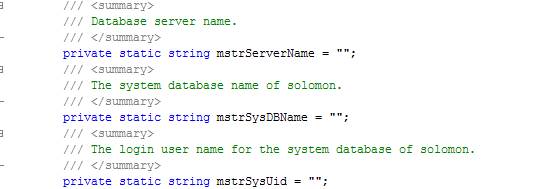
　　2>  注释从新行增加，与代码开始处左对齐

　　3>  双斜线与注释之间以空格分开，示例图如下所示：

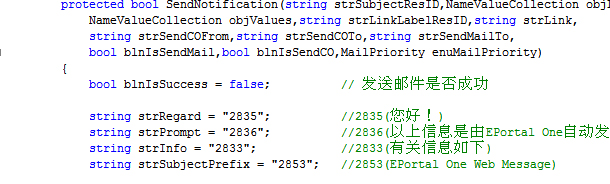


**3.5     变量注释规范**

　　1>  定义变量时需添加变量注释，用以说明变量的用途。

　　2>  Class级变量应以采用 /// 形式自动产生XML标签格式的注释，示例图如下所示：　　                                                                                                

　　3>  方法级的变量注释可以放在变量声明语句的后面，与前后行变量声明的注释左对齐，注释与代码间以Tab隔开。



**4    命名规则**

**4.1     命名的基本约定**

　　1>  要使用可以准确说明变量/字段/类的完整的英文描述符，如firstName。对一些作用显而易见的变量可以采用简单的命名，如在循环里的递增（减）变量就可以被命名为 “i”。

　　2>  要尽量采用项目所涉及领域的术语。

　　3>  要采用大小写混合，提高名字的可读性。为区分一个标识符中的多个单词，把标识符中的每个单词的首字母大写。不采用下划线作分隔字符的写法。

　　　　有两种适合的书写方法，适应于不同类型的标识符：

**PasalCasing：**标识符的第一个单词的字母大写；

**camelCasing：**标识符的第一个单词的字母小写。

　　4>  下表描述了不同类型标识符的大小写规则：

|  |  |  |
| --- | --- | --- |
| **标识符** | **大小写** | **示例** |
| 命名空间 | Pascal | namespace Com.Techstar.ProductionCenter |
| 类型 | Pascal | public class DevsList |
| 接口 | Pascal | public interface ITableModel |
| 方法 | Pascal | public void UpdateData() |
| 属性 | Pascal | Public int Length{…} |
| 事件 | Pascal | public event EventHandler Changed; |
| 私有字段 | Camel | private string fieldName; |
| 非私有字段 | Pascal | public string FieldName； |
| 枚举值 | Pascal | FileMode{Append} |
| 参数 | Camel | public void UpdateData(string fieldName) |
| 局部变量 | Camel | string fieldName; |

　　5>  避免使用缩写，如果一定要使用，就谨慎使用。同时，应该保留一个标准缩写的列表，并且在使用时保持一致。

　　6>  对常见缩略词，两个字母的缩写要采用统一大小写的方式（示例：ioStream，getIOStream）；多字母缩写采用首字母大写，其他字母小写的方式（示例： getHtmlTag）；

　　7>  避免使用长名字（最好不超过 15 个字母）。

　　8>  避免使用相似或者仅在大小写上有区别的名字。

**4.2     各种标示符类型的命名约定**

**1>  程序集和命名空间命名**

　 项目名称 + 模块名称，一般情况下建议命名空间和目录结构相同。例如：

项目数据访问：项目名称.Models；

**2> 类和接口命名**

　　l  类的名字要用名词；

　　l  避免使用单词的缩写，除非它的缩写已经广为人知，如HTTP。

　　l  接口的名字要以字母I开头。保证对接口的标准实现名字只相差一个“I”前缀，例如对IComponent接口的标准实现为Component；

　　l  泛型类型参数的命名：命名要为T或者以T开头的描述性名字，例如：

　　　　public class List<T>

　　　　public class MyClass<Tsession>

　　l  对同一项目的不同命名空间中的类，命名避免重复。避免引用时的冲突和混淆；

**3> 方法命名**

　　l  第一个单词一般是动词；

　　l  如果方法返回一个成员变量的值，方法名一般为Get+成员变量名，如若返回的值是bool变量，一般以Is作为前缀。另外，如果必要，考虑用属性来替代方法；

　　l  如果方法修改一个成员变量的值，方法名一般为：Set + 成员变量名。同上，考虑 用属性来替代方法。

**4> 变量命名**

　　l  按照使用范围来分，我们代码中的变量的基本上有以下几种类型，类的公有变量；类的私有变量（受保护同公有）；方法的参数变量；方法内部使用的局部变量。这些变量的命名规则基本相同，见标识符大小写对照表。

　　l 　　类的变量按通常的方式命名，无特殊要求；

　　l 　　方法的参数变量采用camalString，例如workerName；

　　l  方法内部的局部变量采用camalString，例如workerName。

　　l  不要用\_或&作为第一个字母；

　　l  尽量要使用短而且具有意义的单词；

　　l  单字符的变量名一般只用于生命期非常短暂的变量：i,j,k,m,n一般用于integer；c,d,e 一般用于characters；s用于string

　　l  如果变量是集合，则变量名要用复数。例如表格的行数，命名应为：RowsCount；

**5    其它规范**

**5.1      编程风格**

**1>  变量声明：**

　　为了保持更好的阅读习惯，请不要把多个变量声明写在一行中，即一行只声明一个变量。

　　例如：

　　String strTest1, strTest2;

　　应写成：

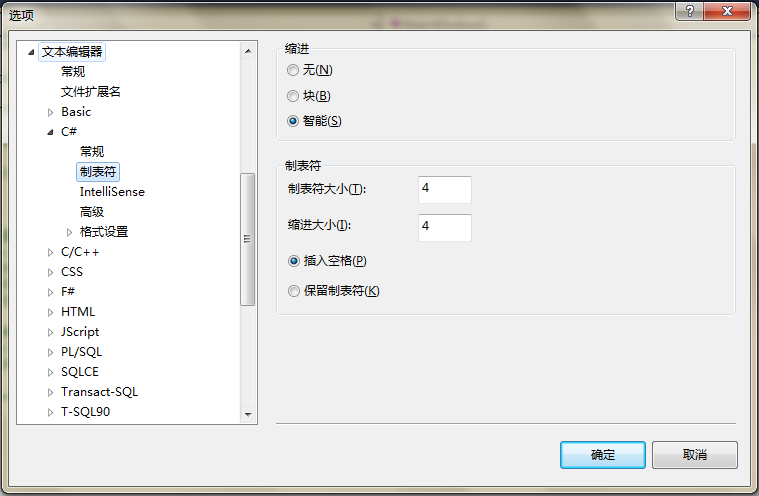
　　String strTest1;

　　String strTest2;

**2>   代码缩进：**

　　l  一致的代码缩进风格，有利于代码的结构层次的表达，使代码更容易阅读和传阅；

　　l  代码缩进使用Tab键实现，最好不要使用空格，为保证在不同机器上使代码缩进保持一致，特此规定C#的Tab键宽度为4个字符，设定界面如下(工具–选项)：



　　l  避免方法中有超过5个参数的情况，一般以2,3个为宜。如果超过了，则应使用对象来传递多个参数。

　　l  为了更容易阅读，代码行请不要太长，最好的宽度是屏幕宽度（根据不同的显示分辩率其可见宽度也不同）。请不要超过您正在使用的屏幕宽度。（每行代码不要超过80个字符。）

　　l  所有块的{}号分别放置一行，并嵌套对齐，不要放在同一行上

**3>    空白：**

　　l  空行将逻辑相关的代码段分隔开，以提高可读性。

　　l  下列情况应该总是使用两个空行：

　　　　a)  一个源文件的两个片段(section)之间。

　　　　b)  类声明和接口声明之间。

　　l  下列情况应该总是使用一个空行：

　　　　a)  两个方法之间。

　　　　b)  方法内的局部变量和方法的第一条语句之间。

　　　　d)  一个方法内的两个逻辑段之间，用以提高可读性。

　　l  下列情况应该总是使用空格：

　　　　a)  空白应该位于参数列表中逗号的后面，如：

　　　　　　void UpdateData(int a, int b)

　　　　b)  所有的二元运算符，除了"."，应该使用空格将之与操作数分开。一元操作符和操作数之间不因该加空格，比如：负号("-")、自增("++")和自减("--")。例如：

　　　　　　　　a += c + d;

　　　　　　　　d++;

　　　　c)  for 语句中的表达式应该被空格分开，例如：

　　　　　　　　for (expr1; expr2; expr3)

　　　　d)  强制转型后应该跟一个空格，例如：

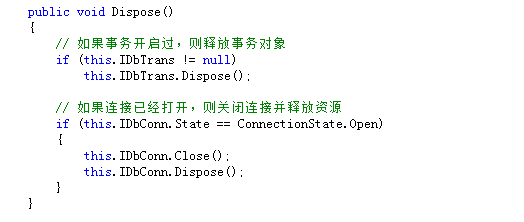
　　　　　　　　char c;

　　　　　　　　int a = 1;

　　　　　　　　c = (char) a;

**5.2     资源释放**

　　所有外部资源都必须显式释放。例如：数据库连接对象、IO对象等。



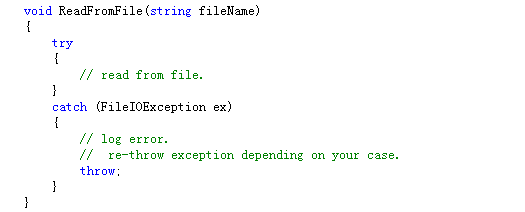
**5.3     错误处理**

　　1>  不要“捕捉了异常却什么也不做“。如果隐藏了一个异常，你将永远不知道异常到底发生了没有。

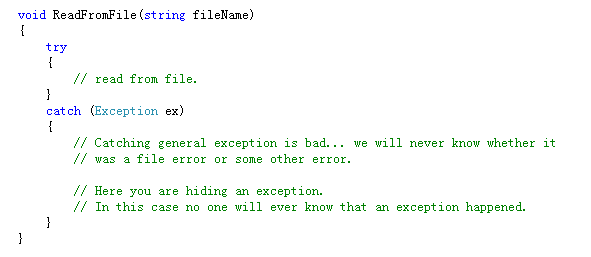
　　2>  发生异常时，给出友好的消息给用户，但要精确记录错误的所有可能细节，包括发生的时间，和相关方法，类名等。

　　3>  只捕捉特定的异常，而不是一般的异常。

**正确做法：**



**错误做法：**



**5.4     其它**

　　1>  一个方法只完成一个任务。不要把多个任务组合到一个方法中，即使那些任务非常小。

　　2>  使用C#的特有类型，而不是System命名空间中定义的别名类型。

　　3>  别在程序中使用固定数值，用常量代替。

　　4>  避免使用很多成员变量。声明局部变量，并传递给方法。不要在方法间共享成员变量。如果在几个方法间共享一个成员变量，那就很难知道是哪个方法在什么　　　　   时候修改了它的值。

　　5>  别把成员变量声明为 public 或 protected。都声明为 private 而使用 public/protected 的属性

　　6>  不在代码中使用具体的路径和驱动器名。 使用相对路径，并使路径可编程。

　　7>  应用程序启动时作些“自检”并确保所需文件和附件在指定的位置。必要时检查数据库连接。出现任何问题给用户一个友好的提示。

　　8>  如果需要的配置文件找不到，应用程序需能自己创建使用默认值的一份。

　　9>  如果在配置文件中发现错误值，应用程序要抛出错误，给出提示消息告诉用户正确值。

　　10>  DataColumn取其列时要用字段名，不要用索引号。  
　　　　例： 正确DataColumn[“Name”]  
    　　　　  不好 DataColumn[0]

　　11>  在一个类中，字段定义全部统一放在class的头部、所有方法或属性的前面。

　　12>  在一个类

private void button1\_Click(object sender, EventArgs e)

{

string[] sd = new string[dataGridView1.RowCount-5]; //新建一个数组存放观测角度的

原始值

double[] sdr = new double[sd.Length]; //新建一个数组存放观测角度的弧度值

double[] cr = new double[sd.Length ]; //新建一个数组存放计算的坐标方位角

double sum = 0;

cr[0] = dmstorad(Convert.ToString(dataGridView1.Rows[0].Cells[4].Value));

//获取第一个坐标方位角，并将其转换成弧度，放入cr[]数组第一个元素中

double acd = dmstorad(Convert.ToString

(dataGridView1.Rows[dataGridView1.RowCount-6].Cells[4].Value));

//获取终边坐标方位角，并将其转换成弧度，放入放入acd中用于计算和检核

第四章 导线平差计算

数据

第四章 导线平差计算

一、角度调整与计算-2

for (int i = 1; i < sd.Length; i++) //从第二行开始循环，将观测角度的原始值放入

sd[]数组中,并转换成弧度值存放在sdr数组中

{

sd[i] = Convert.ToString(dataGridView1.Rows[i].Cells[1].Value);

sdr[i] = dmstorad(sd[i]);

}

sum = fangweijiao(sdr, cr); //计算改正前坐标方位角和观测角度总和，分别存储在

cr数组和sum中

dataGridView1.Rows[dataGridView1.RowCount - 4].Cells[1].Value = radtodms(sum);

//将观测角度总和放入表格中

double fd, fdx;

fd = cr[cr.Length - 1] - acd;//计算角度闭合差，单位弧度

fdx = 60 \* Math.Sqrt(sd.Length - 1);//计算角度闭合差限差，单位秒

dataGridView1.Rows[dataGridView1.RowCount - 3].Cells[1].Value =

Convert.ToString(Math.Round(fd \* 180 / Math.PI \* 3600, 2))+"″";

//将角度闭合差存入表格中

dataGridView1.Rows[dataGridView1.RowCount - 2].Cells[1].Value =

Convert.ToString(Math.Round(fdx, 2))+"″";//将角度闭合差限差存入表格中

一、角度调整与计算-3

if (Math.Abs(fd \* 180 / Math.PI \* 3600) > fdx)//检查角度闭合差是否满足要求

MessageBox.Show("角度闭合差超限！");

else

{

double vd = -fd / (sd.Length - 1);//分配角度闭合差（观测左角）

double sumvd = 0;

for (int i = 1; i < sdr.Length; i++)

{

sdr[i] += vd;//计算改正后的观测角度，并存入sdr数组中

sumvd += vd;

dataGridView1.Rows[i].Cells[2].Value =

Convert.ToString(Math.Round(vd \* 180 / Math.PI \* 3600, 2))+"″";

//将角度改正数存入表格中

dataGridView1.Rows[i].Cells[3].Value = radtodms(sdr[i]);

}

if (Math.Round(sumvd, 8) != Math.Round(-fd , 8)) //秒保留2位对应弧度是8位

MessageBox.Show("角度改正数分配有误！");

else

dataGridView1.Rows[dataGridView1.RowCount - 4].Cells[2].Value =

Convert.ToString(Math.Round(sumvd \* 180 / Math.PI \* 3600, 2)) + "″";

//将角度改正数总和存入表格中

sum = fangweijiao(sdr, cr);//推算改正后的坐标方位角

if (Math.Round(cr[cr.Length - 1], 8) != Math.Round(acd, 8))

MessageBox.Show("坐标方位角推算有误！");

else

{

dataGridView1.Rows[dataGridView1.RowCount - 4].Cells[3].Value =

radtodms(sum); //将改正后观测角度总和放入表格中

for (int i = 1; i < cr.Length-1; i++)//将改正后坐标方位角存入表格

dataGridView1.Rows[i].Cells[4].Value = radtodms(cr[i]);

} }

//至此角度调整和计算完毕

第四章 导线平差计算

计算结果

double x2, y2, x3, y3; //存放已知两个点的x，y坐标

x2 = Convert.ToDouble(dataGridView1.Rows[1].Cells[12].Value);

y2 = Convert.ToDouble(dataGridView1.Rows[1].Cells[13].Value);

x3 = Convert.ToDouble(dataGridView1.Rows[sd.Length - 1].Cells[12].Value);

y3 = Convert.ToDouble(dataGridView1.Rows[sd.Length - 1].Cells[13].Value);

double[] sl = new double[sd.Length - 1]; //存放观测距离

double[] dx = new double[sl.Length]; //存放坐标增量

double[] dy = new double[sl.Length];

double suml = 0, sumdx = 0, sumdy = 0;

for (int i=1;i<sl.Length;i++)

{

sl[i] = Convert.ToDouble(dataGridView1.Rows[i].Cells[5].Value);

//将观测距离放到sl数组中

suml += sl[i]; //计算距离总和

dx[i] = sl[i] \* Math .Cos (cr[i]); //利用距离和坐标方位角计算坐标增量

dy[i] = sl[i] \* Math .Sin (cr[i]);

sumdx += dx[i]; //计算坐标增量总和

sumdy += dy[i];

}

第四章 导线平差计算

二、坐标增量调整与计算-1 （先计算、后显示再表格中）

C#中函数

第四章 导线平差计算

数据

double fx, fy, fxy, k1;

fx = sumdx - (x3 - x2); //计算坐标增量闭合差

fy = sumdy - (y3 - y2);

fxy = Math.Sqrt(fx \* fx + fy \* fy); //计算导线全长闭合差

k1 = suml / fxy; //计算导线全长相对闭合差分母

double[] vx = new double[sl.Length]; //定义数组用于存放坐标增量的改正数及总和

double[] vy = new double[sl.Length];

double sumvx = 0, sumvy = 0;

double[] cx = new double[sl.Length]; //定义数组用于存放改正后的坐标增量及总和

double[] cy = new double[sl.Length];

double sumcx = 0, sumcy = 0;

double[] x = new double[sl.Length+1]; //定义数组用于存放x，y坐标

double[] y = new double[sl.Length+1];

x[1] = x2;

y[1] = y2;

if (k1 < 2000) //判断导线全长相对闭合差是否超限

MessageBox.Show("导线全长相对闭合差超限！");

else

{

for(int i=1; i <vx.Length; i ++)

{

vx[i] = -fx \* sl[i] / suml; //计算坐标增量改正数

vy[i] = -fy \* sl[i] / suml;

sumvx += vx[i]; //计算坐标增量改正数总和

sumvy += vy[i];

}

if (Math.Round(sumvx, 4) != Math.Round(-fx, 4)

|| Math.Round(sumvy, 4) != Math.Round(-fy, 4))

MessageBox.Show("坐标增量分配有误！");

else

{

for (int i = 1; i < vx.Length; i ++)

{

cx[i] = dx[i] + vx[i]; //计算改正后坐标增量

cy[i] = dy[i] + vy[i];

sumcx += cx[i]; //计算改正后坐标增量总和

sumcy += cy[i];

}

C#中1/2000=0两个整数相除，可以写成1.0/2000

4位小数

if (Math.Round(sumcx, 4) != Math.Round(x3 - x2, 4)

|| Math.Round(sumcy, 4) != Math.Round(y3 - y2, 4))

MessageBox.Show(“改正后的坐标增量计算有误！");

else

{

for (int i = 2; i < x.Length; i++)

{

x[i] = x[i-1] + cx[i-1]; //计算x,y坐标

y[i] = y[i-1] + cy[i-1];

}

if (Math.Round(x[x.Length - 1], 4) != Math.Round(x3, 4)

|| Math.Round(y[y.Length - 1], 4) != Math.Round(y3, 4))

MessageBox.Show("坐标计算有误！");

else

{

for(int i=1;i<sl.Length;i++)

{

dataGridView1.Rows[i].Cells[6].Value = Convert.ToString(Math.Round(dx[i],4));

//将坐标增量放入表格

dataGridView1.Rows[i].Cells[7].Value = Convert.ToString(Math.Round(dy[i],4));

dataGridView1.Rows[i].Cells[8].Value = Convert.ToString(Math.Round(vx[i],4));

//将坐标增量改正数放入表格

dataGridView1.Rows[i].Cells[9].Value = Convert.ToString(Math.Round(vy[i],4));

二、坐标增量调整与计算-5

dataGridView1.Rows[i].Cells[10].Value = Convert.ToString(Math.Round(cx[i], 4));

dataGridView1.Rows[i].Cells[11].Value = Convert.ToString(Math.Round(cy[i], 4));

//将改正后坐标增量放入表格

dataGridView1.Rows[i].Cells[12].Value = Convert.ToString(Math.Round(x[i], 3));

dataGridView1.Rows[i].Cells[13].Value = Convert.ToString(Math.Round(y[i], 3));

//将x,y坐标放入表格

}

dataGridView1.Rows[dataGridView1.RowCount - 4].Cells[5].Value =

Convert.ToString(Math.Round(suml,4));

dataGridView1.Rows[dataGridView1.RowCount - 4].Cells[6].Value =

Convert.ToString(Math.Round(sumdx,4));

dataGridView1.Rows[dataGridView1.RowCount - 4].Cells[7].Value =

Convert.ToString(Math.Round(sumdy,4));

//将距离总和、坐标增量总和放入表格中

dataGridView1.Rows[dataGridView1.RowCount - 4].Cells[8].Value =

Convert.ToString(Math.Round(sumvx,4));

dataGridView1.Rows[dataGridView1.RowCount - 4].Cells[9].Value =

Convert.ToString(Math.Round(sumvy,4));

//将坐标增量改正数总和、改正后坐标增量总和放入表格中

二、坐标增量调整与计算-6

dataGridView1.Rows[dataGridView1.RowCount - 4].Cells[10].Value =

Convert.ToString(Math.Round(sumcx,4));

dataGridView1.Rows[dataGridView1.RowCount - 4].Cells[11].Value =

Convert.ToString(Math.Round(sumcy,4));

dataGridView1.Rows[dataGridView1.RowCount - 3].Cells[7].Value =

Convert.ToString(Math.Round(fx,4));

dataGridView1.Rows[dataGridView1.RowCount - 2].Cells[7].Value =

Convert.ToString(Math.Round(fy,4));

dataGridView1.Rows[dataGridView1.RowCount - 3].Cells[10].Value =

Convert.ToString(Math.Round(fxy,4));

dataGridView1.Rows[dataGridView1.RowCount - 2].Cells[11].Value =

Convert.ToString((int)k1); //导线全长相对闭合差分母取整

}}}}