室内定位算法仿真程序设计与说明

# 版本：

|  |  |  |
| --- | --- | --- |
| 日期 | 版本 | 改动说明 |
| 2018．1．3 | Alpha 1.0 | 基本功能 |
| 2018．1．26 | Beta 3.0 | Weighted Taylor  Both on CPU and ARM. |
|  |  |  |
|  |  |  |

# 主要功能

## 文件读取

* 读取“.ini”配置文件
* 读取记录距离信息的“.log”文件

可以处理两种格式的log文件

* 可以加载机器的位图

## 算法功能

* 位置算法
  + 全质心
  + 相减最小二乘
  + 纯二维求中心
  + TaylorSeries
  + WeightedTaylor
  + KalmanCoupled
  + KalmanGauss
  + KalmanWeight
  + KalmanSmooth
  + LMedS
  + Bilateration
* 标签追踪

## 结果显示

* 显示距离
* 显示定位解算结果
* 显示计算中间过程

# 主要结构

分为**主程序（locAlgrithom）**和**测试程序（test）**两部分。locAlgrithom实现软件功能，test对locAlgrithom重要模块进行单元测试。

locAlgrithom实现流程为：输入🡪定位解算🡪显示。

## 输入

分为配置文件读取与测距文件读取。

### 配置文件

在文件dataSensorIni.h和dataSensorIni.cpp中实现。

class uiMainWindow中的class dataSensorIni用于读取ini配置文件，并将数据存储到configData中。

struct configData存储传感器的位置，oper、warning、stop区域的矢量坐标信息。程序中configData通过dataSensorIni::get\_q()读取。

### 测距文件

在文件dataDistanceLog.h和dataDistanceLog.cpp中实现。

class uiMainWindow中的class dataDistanceLog用于读取log测距文件，并将数据存储到distanceData中。

struct distanceData {

QMap<int, oneTag> tagsData; // 保存不同tag的测距信息

bool isInitialized{false}; // data ready

};

struct oneTag {

oneTag(int tagId) : tagId{tagId} {}

oneTag() {}

int tagId;

QVector<oneLogData> distData; //测距信息

};

struct oneLogData { // 用于存储一次完整的测距信息

QDateTime time; // kalman相关算法使用

QVector<int> distance;

locationCoor p\_t\_1;

locationCoor p\_t;

QString toString(); // 方便显示

};

oneTag::tagId与distanceData::tagsData的索引是一致的。程序中distanceData通过dataDistanceLog::get\_q()读取。

## 定位解算

定位算法主要分为：算法调用、算法执行、结果存储。

### 算法调用

不同算法通过主界面上按钮调用。解算算法主要分定位与追踪两个子模块。

#### 定位算法

枚举CALC\_POS\_TYPE对应界面传递的不同定位算法调用。不同的算法调用后通过统一的uiMainWindow::posCalcPROCESS(CALC\_POS\_TYPE type)函数调用。

void uiMainWindow::posCalcPROCESS(CALC\_POS\_TYPE type) {

// 1. 界面更新

// 2. 记录type

calcPos.calcPosType = type;

// 3. 计时

// 4. 循环不同tag

foreach (storeTagInfo \*info, store.tags) {

// 5. 清空显示信息

// 6. 算法调用接口

calcPos.calcPosVector(info);

// 7. 清空追踪数据

}

// 8. 记录显示时间

// 9. 更新界面显示

handleModelDataUpdate(false);

}

#### 追踪算法

与定位算法类似，枚举TRACK\_METHOD对应界面传递的不同追踪算法调用。不同的算法调用后通过统一的uiMainWindow::trackCalcPROCESS(TRACK\_METHOD type)函数调用。

void uiMainWindow::trackCalcPROCESS(TRACK\_METHOD type) {

// 1. 界面更新

// 2. 记录type

calcTrack.calcTrackMethod = type;

// 3. 计时

// 4. 循环不同tag

foreach (storeTagInfo \*info, store.tags) {

// 5. 清空显示信息

// 6. 算法调用接口，根据定位数据计算追踪数据

calcTrack.calcTrackVector(info->methodInfo[MEASUR\_STR], info->methodInfo[TRACKx\_STR]);

}

// 7. 记录显示时间

// 8. 更新界面显示

handleModelDataUpdate(false);

}

### 算法执行

#### 定位算法

void calcTagPos::calcPosVector (storeTagInfo \*tagInfo) {

// 清空原有数据

for (int i = 0; i < oneTagData.distData.count(); i++) {

tmpX = calcOnePosition(tmpDist.distance, mse, T, tmpLastPos, kalmanData, usedSensor, tmpTrace, tmpWeight, x\_hat);

// 检测NLOS，距离滤波

tmpX = calcOnePosition(tmpDist.distance, mse, T, tmpLastPos, kalmanData, usedSensor, tmpTrace, tmpWeight, x\_hat);

// 结果及中间结果记入class showTagModel

}

}

locationCoor calcTagPos::calcOnePosition(const int \*dist, dType &MSE, dType T,locationCoor lastPos, oneKalmanData &kalmanData, bool \*usedSensor, QVector<QPointF> &iterTrace, QVector<dType> &weight, QPointF &x\_hat)

calcTagPos::calcOnePosition为定位算法的接口函数，通过不同calcPosType调用不同的静态定位算法，并将结果与最终结果传出。calcPosVector将传出的结果记录于class showTagModel中。

#### 追踪算法

与定位算法类似，被调用接口函数如下。

void calcTagTrack::calcTrackVector(storeMethodInfo &tagMeasInfo, storeMethodInfo &tagKalmanInfo) {

// 清空原有数据

for(int i = 0; i < tagMeasInfo.Ans.count(); i++) {

calcOneTrack(z\_x\_meas, T, Rx, tagTrackParam, recParam);

// 结果及中间结果记入class showTagModel

}

}

void calcTagTrack::calcOneTrack(const locationCoor &z\_x\_meas,

dType T, dType Rx,

trackParams &trackParam,

tagTrackRecord &recParam)

为追踪算法的接口函数，通过不同calcTrackMethod调用不同的静态追踪算法，并将结果与最终结果传出。calcTrackVector将传出的结果记录于class showTagModel中。

### 结果存储

计算结果及中间变量存放在class uiMainWindow的class showTagModel中以便后面显示时调取。

class showTagModel {

... ...

... ...

QMap<int, storeTagInfo \*> tags;

};

struct storeTagInfo {

int tagId{-1};

bool isTagPosInitialed{false};

QMap<QString, storeMethodInfo> methodInfo;

// data store for one tag

... ...

... ...

// 清空相关函数

};

struct storeMethodInfo {

// data store for one tag method

... ...

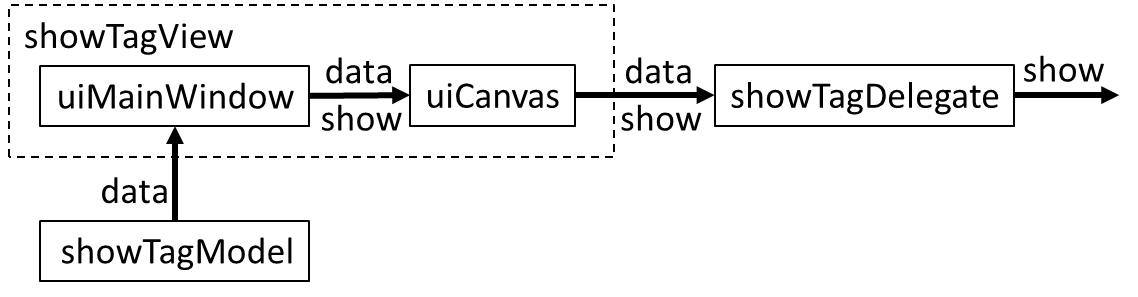
... ...

// 清空相关函数

};

storeTagInfo的索引为tagId。storeMethodInfo保存一种运算方法所算得的结果。这里暂时用于区分定位与追踪的结果。不同计算方法所需要记录的公共部分存放在storeTagInfo的数据部分。

## 显示



class showTagDelegate用于存放tag对应的显示方法及本时刻所需显示的内容，其中本时刻所需显示的内容从class showTagModel中取得。

class uiCanvas是用于显示的画布，showTagDelegate属于uiCanvas，用于记录即将显示的数据及数据显示方法。

class showTagDelegate

{

// add data method

private:

QMap<QString, showTagOneMethod> oneTagMethod;

int tagId{-1};

// data

... ...

// tagViewData中打到对就tagId并存储于tagView中

oneTagView tagView;

private:

// 配色方案与tagId相绑定

static QMap<int, oneTagView> tagViewData;

// tagsViewDataBase中为所有可选的配色方案

static tagsView tagsViewDataBase;

};

struct tagsView { // 记录所有的配色方案

tagsView();

static QList<oneTagView> viewDatabase;

static int count;

};

struct oneTagView { // 一种配色方案

QColor color[2];

SHOW\_SHAPE type[2];

int nColorStyle;

bool isUsed;

};

显示不同tag的解算结果是，先要给tag分配显示配色方案，若配色方案已分配完，tag没有配色方案，则不显示对应tag的解算结果。等其它tag不用显示释放配色方案时，才可以显示其它tag的解算结果。

class uiUsrFrame控制tag是否显示并完成与showTagDelegate::tagsViewDataBase的协调。class uiUsrFrame提供了ui接口，方便打开或关闭显示tag解算结果。class uiUsrFrame中对不同tag的控制通过class uiUsrInfoBtn实现，class uiUsrInfoBtn由QToolButton类派生，为Frame提供了ui接口。

class uiUsrFrame : public QFrame

{

... ...

... ...

private:

QList<uiUsrInfoBtn \*> usrBtns;

... ...

... ...

};

每个tag对应一个uiUsrInfoBtn。uiUsrInfoBtn🡪uiUsrFrame🡪uiMainWindow🡪uiCanvas。uiCanvas通过syncWithUiFrame函数同步uiUsrFrame（数据）与uiCanvas（显示）两者的信息。uiMainWindow同时拥有uiUsrFrame与uiCanvas，并将两者撮合在一起。

显示内容的开关是通过界面的响应直接传递到uiCanvas中，显示时刻uiMainWindow通过函数handleModelDataUpdate给uiCanvas喂数据。