GTSAM库用于SFM（以数据集排版）

1. SFM简介

2. Examples

2.1 SFMdata.h

包含两个函数：

```
// 构建路标点
std::vector<gtsam::Point3> createPoints() {

  // Create the set of ground-truth landmarks
  std::vector<gtsam::Point3> points;
  points.push_back(gtsam::Point3(10.0,10.0,10.0));
  points.push_back(gtsam::Point3(-10.0,10.0,10.0));
  points.push_back(gtsam::Point3(-10.0,-10.0,10.0));
  points.push_back(gtsam::Point3(10.0,-10.0,10.0));
  points.push_back(gtsam::Point3(10.0,10.0,-10.0));
  points.push_back(gtsam::Point3(-10.0,10.0,-10.0));
  points.push_back(gtsam::Point3(-10.0,-10.0,-10.0));
  points.push_back(gtsam::Point3(10.0,-10.0,-10.0));

  return points;
}
```

```
// 构建相机位姿
std::vector<gtsam::Pose3> createPoses(
            const gtsam::Pose3& init = gtsam::Pose3(gtsam::Rot3::Ypr(M_PI/2,0,-M
            const gtsam::Pose3& delta = gtsam::Pose3(gtsam::Rot3::Ypr(0,-M_PI/4,
            int steps = 8) {

  // Create the set of ground-truth poses
  // Default values give a circular trajectory, radius 30 at pi/4 intervals, alw
  std::vector<gtsam::Pose3> poses;
  int i = 1;
  poses.push_back(init);
  for(; i < steps; ++i) {
```

```
    poses.push_back(poses[i-1].compose(delta));
  }

  return poses;
}
```

2.2 data.h数据集

	GenericProjectionFactor	ExpressionFactor	SmartFactor
SFMxample.cpp	✔		
SFMExampleExpression.cpp		✔	
SFMExample_SmartFactor.cpp			✔
SFMExample_SmartFactorPCG.cpp			✔

1. **GenericProjectionFactor**

```
class GenericProjectionFactor: public NoiseModelFactor2<POSE, LANDMARK> {
protected:

  // Keep a copy of measurement and calibration for I/O
  Point2 measured_;                    ///< 2D measurement
  boost::shared_ptr<CALIBRATION> K_;  ///< shared pointer to calibration objec
  boost::optional<POSE> body_P_sensor_; ///< The pose of the sensor in the boc

  // verbosity handling for Cheirality Exceptions
  bool throwCheirality_; ///< If true, rethrows Cheirality exceptions (default
  bool verboseCheirality_; ///< If true, prints text for Cheirality exceptions

public:

  /// shorthand for base class type
  typedef NoiseModelFactor2<POSE, LANDMARK> Base;

  /// shorthand for this class
  typedef GenericProjectionFactor<POSE, LANDMARK, CALIBRATION> This;

  /// shorthand for a smart pointer to a factor
  typedef boost::shared_ptr<This> shared_ptr;

  /// Default constructor
```

```
GenericProjectionFactor() :
    measured_(0, 0), throwCheirality_(false), verboseCheirality_(false) {
}
```

A. 构造函数：两种，后者比前者多了对正景深约束的异常处理

```
/**
     * Constructor
     * TODO: Mark argument order standard (keys, measurement, parameters)
     * @param measured is the 2 dimensional location of point in image (the meas
     * @param model is the standard deviation
     * @param poseKey is the index of the camera
     * @param pointKey is the index of the landmark
     * @param K shared pointer to the constant calibration
     * @param body_P_sensor is the transform from body to sensor frame (default
     */
    GenericProjectionFactor(const Point2& measured, const SharedNoiseModel& mode
        Key poseKey, Key pointKey, const boost::shared_ptr<CALIBRATION>& K,
        boost::optional<POSE> body_P_sensor = boost::none) :
          Base(model, poseKey, pointKey), measured_(measured), K_(K), body_P_sen
          throwCheirality_(false), verboseCheirality_(false) {}

    /**
     * Constructor with exception-handling flags
     * TODO: Mark argument order standard (keys, measurement, parameters)
     * @param measured is the 2 dimensional location of point in image (the meas
     * @param model is the standard deviation
     * @param poseKey is the index of the camera
     * @param pointKey is the index of the landmark
     * @param K shared pointer to the constant calibration
     * @param throwCheirality determines whether Cheirality exceptions are rethr
     * @param verboseCheirality determines whether exceptions are printed for Ch
     * @param body_P_sensor is the transform from body to sensor frame  (default
     */
    GenericProjectionFactor(const Point2& measured, const SharedNoiseModel& mode
        Key poseKey, Key pointKey, const boost::shared_ptr<CALIBRATION>& K,
        bool throwCheirality, bool verboseCheirality,
        boost::optional<POSE> body_P_sensor = boost::none) :
          Base(model, poseKey, pointKey), measured_(measured), K_(K), body_P_sen
          throwCheirality_(throwCheirality), verboseCheirality_(verboseCheiralit

```

B. 优化结果处理：存储函数，打印函数

```
/// Evaluate error h(x)-z and optionally derivatives
    Vector evaluateError(const Pose3& pose, const Point3& point,
        boost::optional<Matrix&> H1 = boost::none, boost::optional<Matrix&> H2 =
      try {
        if(body_P_sensor_) {
          if(H1) {
            gtsam::Matrix H0;
            PinholeCamera<CALIBRATION> camera(pose.compose(*body_P_sensor_, H0),
            Point2 reprojectionError(camera.project(point, H1, H2, boost::none)
            *H1 = *H1 * H0;
            return reprojectionError;
          } else {
            PinholeCamera<CALIBRATION> camera(pose.compose(*body_P_sensor_), *K_
            return camera.project(point, H1, H2, boost::none) - measured_;
          }
        } else {
          PinholeCamera<CALIBRATION> camera(pose, *K_);
          return camera.project(point, H1, H2, boost::none) - measured_;
        }
      } catch( CheiralityException& e) {
        if (H1) *H1 = Matrix::Zero(2,6);
        if (H2) *H2 = Matrix::Zero(2,3);
        if (verboseCheirality_)
          std::cout << e.what() << ": Landmark "<< DefaultKeyFormatter(this->key
              " moved behind camera " << DefaultKeyFormatter(this->key1()) << st
        if (throwCheirality_)
          throw CheiralityException(this->key2());
      }
      return Vector2::Constant(2.0 * K_->fx());
    }

/**
     * print
     * @param s optional string naming the factor
     * @param keyFormatter optional formatter useful for printing Symbols
     */
    void print(const std::string& s = "", const KeyFormatter& keyFormatter = Def
      std::cout << s << "GenericProjectionFactor, z = ";
      traits<Point2>::Print(measured_);
      if(this->body_P_sensor_)
        this->body_P_sensor_->print("  sensor pose in body frame: ");
      Base::print("", keyFormatter);
    }
```

C. 两个案例中暂时未出现的虚函数

```
/// @return a deep copy of this factor
// 克隆方法
    virtual gtsam::NonlinearFactor::shared_ptr clone() const {
      return boost::static_pointer_cast<gtsam::NonlinearFactor>(
          gtsam::NonlinearFactor::shared_ptr(new This(*this))); }

/// equals
// equals方法
    virtual bool equals(const NonlinearFactor& p, double tol = 1e-9) const {
      const This *e = dynamic_cast<const This*>(&p);
      return e
          && Base::equals(p, tol)
          && traits<Point2>::Equals(this->measured_, e->measured_, tol)
          && this->K_->equals(*e->K_, tol)
          && ((!body_P_sensor_ && !e->body_P_sensor_) || (body_P_sensor_ && e->b
    }
```

2.3 dubrovnik-3-7-pre.txt数据集

大数据量的优化问题，最大迭代次数设为100。

	GeneralSFMFactor	ExpressionFactor	Optimizer	time（ms）
SFMExample_bal.cpp	✔		LM	81.5201
SFMExample_bal_METIS.cpp	✔		LM	80.195
SFMExample_bal_COLAMD.cpp	✔		LM	68.628
SFMExampleExpressions_bal.cpp		✔	LM	51.84

矩阵排序算法：

METIS算法（Multilevel Partitioning of Irregular Networks）通过将图形分解为小的连通子图来工作，然后在子图中应用递归二分划分。 METIS算法的主要思想是将矩阵分解为多个子矩阵，这些子矩阵可以使用较少的存储器和更快的算法进行处理。

COLAMD算法（Column Approximate Minimum Degree）则使用一种基于对称因式分解的技术，通过对矩阵的列进行排序，使其具有更好的稀疏性。 COLAMD算法的主要目的是减少高斯消元算法的计算时间和内存使用，从而提高矩阵求解的效率。

3. 总结与分析