IMU因子图原理及代码介绍

基本语法

1.IMU预积分模块

GTSAM的一种IMU预积分在**PreintegratedImuMeasurements**类中实现(ImuFactor.h文件中)：

```
class GTSAM_EXPORT PreintegratedImuMeasurements: public PreintegrationType {

  friend class ImuFactor;
  friend class ImuFactor2;

protected:

  Matrix9 preintMeasCov_; ///< COVARIANCE OF: [PreintROTATION PreintPOSITION Pre
  ///< (first-order propagation from *measurementCovariance*).

public:

  /// Default constructor for serialization and wrappers
  PreintegratedImuMeasurements() {
    preintMeasCov_.setZero();
  }

  /**
    *  Constructor, initializes the class with no measurements
    *  @param p       Parameters, typically fixed in a single application
    *  @param biasHat Current estimate of acceleration and rotation rate biases
    */
  PreintegratedImuMeasurements(const std::shared_ptr<PreintegrationParams>& p,
      const imuBias::ConstantBias& biasHat = imuBias::ConstantBias()) :
      PreintegrationType(p, biasHat) {
    preintMeasCov_.setZero();
  }
```

可以看到该类是继承自**PreintegrationType**类，并且和**ImuFactor**是友元。构造函数中可以传入IMU预积分的参数（在PreintegrationParams中）。保护成员中有preintMeasCov_，构造函数中主要是把该矩阵令为零。

构造函数也可以直接从基类构造：

```
/**
  *  Construct preintegrated directly from members: base class and preintMeasCov
  *  @param base               PreintegrationType instance
  *  @param preintMeasCov      Covariance matrix used in noise model.
  */
  PreintegratedImuMeasurements(const PreintegrationType& base, const Matrix9& pr
     : PreintegrationType(base),
       preintMeasCov_(preintMeasCov) {
  }
```

成员函数中，比较常用的是IMU预积分的函数**integrateMeasurement**：

```
/**
   * Add a single IMU measurement to the preintegration.
   * @param measuredAcc Measured acceleration (in body frame, as given by the se
   * @param measuredOmega Measured angular velocity (as given by the sensor)
   * @param dt Time interval between this and the last IMU measurement
   */
  void integrateMeasurement(const Vector3& measuredAcc,
      const Vector3& measuredOmega, const double dt) override;
```

具体实现如下：

```
//------------------------------------------------------------------------------
void PreintegratedImuMeasurements::integrateMeasurement(
    const Vector3& measuredAcc, const Vector3& measuredOmega, double dt) {
  if (dt <= 0) {
    throw std::runtime_error(
        "PreintegratedImuMeasurements::integrateMeasurement: dt <=0");
  }

  // Update preintegrated measurements (also get Jacobian)
  Matrix9 A;  // overall Jacobian wrt preintegrated measurements (df/dx)
  Matrix93 B, C;  // Jacobian of state wrpt accel bias and omega bias respective
  PreintegrationType::update(measuredAcc, measuredOmega, dt, &A, &B, &C);

  // first order covariance propagation:
  // as in [2] we consider a first order propagation that can be seen as a
  // prediction phase in EKF

  // propagate uncertainty
  // TODO(frank): use noiseModel routine so we can have arbitrary noise models.
```

```
  const Matrix3& aCov = p().accelerometerCovariance;
  const Matrix3& wCov = p().gyroscopeCovariance;
  const Matrix3& iCov = p().integrationCovariance;

  // (1/dt) allows to pass from continuous time noise to discrete time noise
  // Update the uncertainty on the state (matrix A in [4]).
  preintMeasCov_ = A * preintMeasCov_ * A.transpose();
  // These 2 updates account for uncertainty on the IMU measurement (matrix B in
  preintMeasCov_.noalias() += B * (aCov / dt) * B.transpose();
  preintMeasCov_.noalias() += C * (wCov / dt) * C.transpose();

  // NOTE(frank): (Gi*dt)*(C/dt)*(Gi'*dt), with Gi << Z_3x3, I_3x3, Z_3x3 (9x3 m
  preintMeasCov_.block<3, 3>(3, 3).noalias() += iCov * dt;
}
```

其中的**update**函数是更新预积分量的主要函数。该函数是基类**PreintegrationType**的成员函数：

```
//---------------------------------------------------------------------------------
void TangentPreintegration::update(const Vector3& measuredAcc,
    const Vector3& measuredOmega, const double dt, Matrix9* A, Matrix93* B,
    Matrix93* C) {
  // Correct for bias in the sensor frame
  // 加速度/角速度 测量值 - BIAS
  Vector3 acc = biasHat_.correctAccelerometer(measuredAcc);
  Vector3 omega = biasHat_.correctGyroscope(measuredOmega);

  // Possibly correct for sensor pose by converting to body frame
  // 一般没有使用该参数
  //默认情况下imu坐标系和body坐标系是一个坐标系，也可以设置不是一个坐标系。
  //通过在初始化时设置p->body_P_sensor的值确定imu坐标系到body坐标系的转换矩阵
  Matrix3 D_correctedAcc_acc, D_correctedAcc_omega, D_correctedOmega_omega;
  if (p().body_P_sensor)
    std::tie(acc, omega) = correctMeasurementsBySensorPose(acc, omega,
        D_correctedAcc_acc, D_correctedAcc_omega, D_correctedOmega_omega);

  // Do update
  deltaTij_ += dt;
  // preintegrated_ 类型为 Eigen::Vector9，初始化为 0  内容为欧拉角、位置、速度
  preintegrated_ = UpdatePreintegrated(acc, omega, dt, preintegrated_, A, B, C);

  if (p().body_P_sensor) {
    // More complicated derivatives in case of non-trivial sensor pose
    *C *= D_correctedOmega_omega;
```

```
    if (!p().body_P_sensor->translation().isZero())
      *C += *B * D_correctedAcc_omega;
    *B *= D_correctedAcc_acc; // NOTE(frank): needs to be last
  }

  // new_H_biasAcc = new_H_old * old_H_biasAcc + new_H_acc * acc_H_biasAcc
  // where acc_H_biasAcc = -I_3x3, hence
  // new_H_biasAcc = new_H_old * old_H_biasAcc - new_H_acc
  // 求解k+1对BIAS的偏导数
  preintegrated_H_biasAcc_ = (*A) * preintegrated_H_biasAcc_ - (*B);

  // new_H_biasOmega = new_H_old * old_H_biasOmega + new_H_omega *
omega_H_biasOmega
  // where omega_H_biasOmega = -I_3x3, hence
  // new_H_biasOmega = new_H_old * old_H_biasOmega - new_H_omega
  preintegrated_H_biasOmega_ = (*A) * preintegrated_H_biasOmega_ - (*C);
}
```

其中在**UpdatePreintegrated**函数进行积分：

```
Vector9 TangentPreintegration::UpdatePreintegrated(const Vector3& a_body,
    const Vector3& w_body, double dt, const Vector9& preintegrated,
    OptionalJacobian<9, 9> A, OptionalJacobian<9, 3> B,
    OptionalJacobian<9, 3> C) {
  const auto theta = preintegrated.segment<3>(0);
  const auto position = preintegrated.segment<3>(3);
  const auto velocity = preintegrated.segment<3>(6);

  // This functor allows for saving computation when exponential map and its
  // derivatives are needed at the same location in so<3>
  // 指数映射
  so3::DexpFunctor local(theta);

  // Calculate exact mean propagation
  Matrix3 w_tangent_H_theta, invH;
  // 相当于 applyInvDexp(w_body, &w_tangent_H_theta, &invH)
  // angular velocity mapped back to tangent space
  // 转换到李代数
  const Vector3 w_tangent = local.applyInvDexp(w_body, A ? &w_tangent_H_theta :
  const Rot3 R(local.expmap());  // nRb: rotation of body in nav frame
  const Vector3 a_nav = R * a_body;    // local 为世界到body的旋转角
  const double dt22 = 0.5 * dt * dt;

  // 简单的积分
  Vector9 preintegratedPlus;
```

```
  preintegratedPlus <<                          // new preintegrated vector:
      theta + w_tangent * dt,                   // theta
      position + velocity * dt + a_nav * dt22,  // position   注意这里的加速度是已
      velocity + a_nav * dt;                    // velocity

  // 计算噪声协方差需要用到的矩阵 A、B、C
  if (A) {
    // Exact derivative of R*a with respect to theta:
    const Matrix3 a_nav_H_theta = R.matrix() * skewSymmetric(-a_body) * local.de

    A->setIdentity();
    A->block<3, 3>(0, 0).noalias() += w_tangent_H_theta * dt;  // theta
    A->block<3, 3>(3, 0) = a_nav_H_theta * dt22;  // position wrpt theta...
    A->block<3, 3>(3, 6) = I_3x3 * dt;            // .. and velocity
    A->block<3, 3>(6, 0) = a_nav_H_theta * dt;    // velocity wrpt theta
  }
  if (B) {
    B->block<3, 3>(0, 0) = Z_3x3;
    B->block<3, 3>(3, 0) = R.matrix() * dt22;
    B->block<3, 3>(6, 0) = R.matrix() * dt;
  }
  if (C) {
    C->block<3, 3>(0, 0) = invH * dt;
    C->block<3, 3>(3, 0) = Z_3x3;
    C->block<3, 3>(6, 0) = Z_3x3;
  }

  return preintegratedPlus;
}
```

在doc文件夹中，GTSAM提供了关于IMU积分的官方文档（**ImuFactor.pdf**），其包含了IMU预积分的公式推导。这里列出与上述代码相关的部分。

首先是IMU的积分：

$$\theta_{k+1} = \theta_k + H\left(\theta_k\right)^{-1} \omega_k^b \Delta_t$$
$$p_{k+1} = p_k + v_k \Delta_t + R_k a_k^b \frac{\Delta_t^2}{2}$$
$$v_{k+1} = v_k + R_k a_k^b \Delta_t$$

分别是角度、位置和速度的递推公式。其中 $H\left(\theta_k\right)^{-1}$ 是BCH近似公式中的雅可比。这个公式对应代码：

```
Vector9 preintegratedPlus;
  preintegratedPlus <<                          // new preintegrated vector:
      theta + w_tangent * dt,                   // theta
```

```
        position + velocity * dt + a_nav * dt22,  // position   注意这里的加速度是已:
        velocity + a_nav * dt;                    // velocity
```

接下来是噪声量的传播。定义 $\zeta_k = [\theta_k, p_k, v_k]$ ，则：

$$\zeta_{k+1} = f\left(\zeta_k, a_k^b, \omega_k^b\right)$$

$$\Sigma_{k+1} = A_k \Sigma_k A_k^T + B_k \Sigma_\eta^{ad} B_k^T + C_k \Sigma_\eta^{gd} C_k^T$$

其中 A_k, B_k, C_k 分别是 ζ_{k+1} 对 $\zeta_k, a_k^b, \omega_k^b$ 的导数，他们分别是9×9，9×3，9×3的矩阵。

最后更新了 ζ_{k+1} 对BIAS的导数：

$$\frac{\partial \zeta_{k+1}}{\partial b_a} = \frac{\partial f}{\partial \zeta_k}\frac{\partial \zeta_k}{\partial b_a} + \frac{\partial f}{\partial a}\frac{\partial a}{\partial b_a} = A\frac{\partial \zeta_k}{\partial b_a} - I * B$$

$$\frac{\partial \zeta_{k+1}}{\partial b_w} = \frac{\partial f}{\partial \zeta_k}\frac{\partial \zeta_k}{\partial b_w} + \frac{\partial f}{\partial w}\frac{\partial w}{\partial b_a} = A\frac{\partial \zeta_k}{\partial b_a} - I * C$$

总结：

- IMU预积分主要用 **PreintegratedImuMeasurements** 类来定义，且它和 **ImuFactor** 是友元
- 在该类中，主要通过 **integrateMeasurement** 函数来进行预积分，预积分公式推导参考 https://zhuanlan.zhihu.com/p/443860992
- 预积分的同时会计算新的噪声协方差矩阵 **preintMeasCov_** ，它将在 **ImuFactor** 中使用

2.IMU预积分参数

IMU预积分类在初始化时，需要提供预积分的参数。常用的预积分参数的定义通过 **PreintegrationParams**：

```
struct PreintegrationParams: PreintegratedRotationParams {
  Matrix3 accelerometerCovariance; ///< continuous-time "Covariance" of accelerc
  Matrix3 integrationCovariance; ///< continuous-time "Covariance" describing ir
  bool use2ndOrderCoriolis; ///< Whether to use second order Coriolis integratic
  Vector3 n_gravity; ///< Gravity vector in nav frame

  /// The Params constructor insists on getting the navigation frame gravity vec
  /// For convenience, two commonly used conventions are provided by named const
  PreintegrationParams(const Vector3& n_gravity)
      : accelerometerCovariance(I_3x3),
        integrationCovariance(I_3x3),
        use2ndOrderCoriolis(false),
```

```
      n_gravity(n_gravity) {}

  // Default Params for a Z-down navigation frame, such as NED: gravity points a
  static boost::shared_ptr<PreintegrationParams> MakeSharedD(double g = 9.81) {
    return boost::make_shared<PreintegrationParams>(Vector3(0, 0, g));
  }

  // Default Params for a Z-up navigation frame, such as ENU: gravity points alc
  static boost::shared_ptr<PreintegrationParams> MakeSharedU(double g = 9.81) {
    return boost::make_shared<PreintegrationParams>(Vector3(0, 0, -g));
  }

  void print(const std::string& s) const;
  bool equals(const PreintegratedRotation::Params& other, double tol) const;

  void setAccelerometerCovariance(const Matrix3& cov) { accelerometerCovariance
  void setIntegrationCovariance(const Matrix3& cov)   { integrationCovariance =
  void setUse2ndOrderCoriolis(bool flag)              { use2ndOrderCoriolis = fl
  ......
  }
```

它继承自**PreintegratedRotationParams**。在实际使用中，通常需要初始化4个参数：重力分量（**MakeSharedD**或**MakeSharedU**中初始化）、加速度协方差**accelerometerCovariance**、角速度协方差**gyroscopeCovariance**和积分协方差**integrationCovariance**。

3.IMU因子

GTSAM中IMU因子的定义在**ImuFactor**类中实现(ImuFactor.h文件中)：

```
class GTSAM_EXPORT ImuFactor: public NoiseModelFactorN<Pose3, Vector3, Pose3, Ve
    imuBias::ConstantBias> {
private:

  typedef ImuFactor This;
  typedef NoiseModelFactorN<Pose3, Vector3, Pose3, Vector3,
      imuBias::ConstantBias> Base;

  PreintegratedImuMeasurements _PIM_;

public:

  // Provide access to the Matrix& version of evaluateError:
```

```
  using Base::evaluateError;

  /** Shorthand for a smart pointer to a factor */
#if !defined(_MSC_VER) && __GNUC__ == 4 && __GNUC_MINOR__ > 5
  typedef typename std::shared_ptr<ImuFactor> shared_ptr;
#else
  typedef std::shared_ptr<ImuFactor> shared_ptr;
#endif

  /** Default constructor - only use for serialization */
  ImuFactor() {}

  /**
   * Constructor
   * @param pose_i Previous pose key
   * @param vel_i  Previous velocity key
   * @param pose_j Current pose key
   * @param vel_j  Current velocity key
   * @param bias   Previous bias key
   * @param preintegratedMeasurements The preintegreated measurements since the
   * last pose.
   */
  ImuFactor(Key pose_i, Key vel_i, Key pose_j, Key vel_j, Key bias,
      const PreintegratedImuMeasurements& preintegratedMeasurements);

  ~ImuFactor() override {
  }

  /// @return a deep copy of this factor
  gtsam::NonlinearFactor::shared_ptr clone() const override;

  /// @name Testable
  /// @{
  GTSAM_EXPORT friend std::ostream& operator<<(std::ostream& os, const ImuFactor
  void print(const std::string& s = "", const KeyFormatter& keyFormatter =
                                          DefaultKeyFormatter) const override;
  bool equals(const NonlinearFactor& expected, double tol = 1e-9) const override
  /// @}

  /** Access the preintegrated measurements. */

  const PreintegratedImuMeasurements& preintegratedMeasurements() const {
    return _PIM_;
  }
...
};
```

其中构造函数：

```
ImuFactor::ImuFactor(Key pose_i, Key vel_i, Key pose_j, Key vel_j, Key bias,
    const PreintegratedImuMeasurements& pim) :
    Base(noiseModel::Gaussian::Covariance(pim.preintMeasCov_), pose_i, vel_i,
        pose_j, vel_j, bias), _PIM_(pim) {
}
```

初始化IMU因子时，需要提供两帧的Key，两帧的速度以及预积分量。然后将该因子add到因子图中即可。

实例分析

以**ImuFactorsExample2.cpp**为例，分析IMU因子的实际使用方式。

首先，调用了PreintegrationParams类，给定IMU预积分的参数：

```
auto params = PreintegrationParams::MakeSharedU(kGravity);   //设置重力分量(0,0,-
  params->setAccelerometerCovariance(I_3x3 * 0.1);     //加速度计方差
  params->setGyroscopeCovariance(I_3x3 * 0.1);         //陀螺仪方差
  params->setIntegrationCovariance(I_3x3 * 0.1);       //积分方差

  params->setUse2ndOrderCoriolis(false);               //是否使用二阶科氏
  params->setOmegaCoriolis(Vector3(0, 0, 0));          //角速度科氏
```

接下来定义了一个误差量（用GTSAM中的Pose3定义）：

```
Pose3 delta(Rot3::Rodrigues(-0.1, 0.2, 0.25), Point3(0.05, -0.10, 0.20));
```

接下来定义了该问题的场景，一个绕原点旋转的相机。这里主要是为了方便后续生成一系列的IMU数据：

```
// Start with a camera on x-axis looking at origin
  double radius = 30;
  const Point3 up(0, 0, 1), target(0, 0, 0);
  const Point3 position(radius, 0, 0);
  const auto camera = PinholeCamera<Cal3_S2>::Lookat(position, target, up);
  const auto pose_0 = camera.pose();
```

```cpp

// Now, create a constant-twist scenario that makes the camera orbit the origi
//创建一个恒定的旋转场景，使相机绕着原点旋转
double angular_velocity = M_PI,  // rad/sec
    delta_t = 1.0 / 18;          // makes for 10 degrees per step
Vector3 angular_velocity_vector(0, -angular_velocity, 0);
Vector3 linear_velocity_vector(radius * angular_velocity, 0, 0);
auto scenario = ConstantTwistScenario(angular_velocity_vector,
                                      linear_velocity_vector, pose_0);
```

创建因子图，ISAM2求解器以及初始值：

```cpp
// Create a factor graph  创建因子图
NonlinearFactorGraph newgraph;
// Create (incremental) ISAM2 solver  创建ISAM2求解器
ISAM2 isam;
// Create the initial estimate to the solution  创建初始估计值
// Intentionally initialize the variables off from the ground truth  故意使初始
Values initialEstimate, result;
```

对初始位置X(0)和速度V(0)添加先验：

```cpp
auto noise = noiseModel::Diagonal::Sigmas(
    (Vector(6) << Vector3::Constant(0.1), Vector3::Constant(0.3)).finished());
newgraph.addPrior(X(0), pose_0, noise);   //对X0添加一个先验值  即初始位置

auto velnoise = noiseModel::Diagonal::Sigmas(Vector3(0.1, 0.1, 0.1));
Vector n_velocity(3);
n_velocity << 0, angular_velocity * radius, 0;
newgraph.addPrior(V(0), n_velocity, velnoise);  //对V0添加一个先验值  初始速度
initialEstimate.insert(V(0), n_velocity);
```

添加BIAS的先验：

```cpp
// Add imu priors  添加IMU bias的先验
Key biasKey = Symbol('b', 0);
auto biasnoise = noiseModel::Diagonal::Sigmas(Vector6::Constant(0.1));
newgraph.addPrior(biasKey, imuBias::ConstantBias(), biasnoise);
initialEstimate.insert(biasKey, imuBias::ConstantBias());     //添加b0到初始估讠
```

创建IMU预积分类：

```
// IMU preintegrator
PreintegratedImuMeasurements accum(params);
```

开始，当i=0时，添加X(0)和X(1)的先验；当i>=2时，添加更多的先验。其中，这些先验位姿是在scenario场景中提取的：

```
// Simulate poses and imu measurements, adding them to the factor graph
  for (size_t i = 0; i < 4; ++i) {
    double t = i * delta_t;
    if (i == 0) {  // First time add two poses   当i==0时  添加X0和X1的先验
      auto pose_1 = scenario.pose(delta_t);
      initialEstimate.insert(X(0), pose_0.compose(delta));   //这里的先验都加上了d
      initialEstimate.insert(X(1), pose_1.compose(delta));
      // GTSAM_PRINT(initialEstimate);

    } else if (i >= 2) {  // Add more poses as necessary    当i>=2时  添加更多的先
      auto pose_i = scenario.pose(t);
      initialEstimate.insert(X(i), pose_i.compose(delta));
    }
```

周期性的添加bias：

```
if (i > 0) {
      // Add Bias variables periodically   周期性的添加bias
      if (i % 5 == 0) {
        biasKey++;
        Symbol b1 = biasKey - 1;
        Symbol b2 = biasKey;
        Vector6 covvec;
        covvec << 0.1, 0.1, 0.1, 0.1, 0.1, 0.1;
        auto cov = noiseModel::Diagonal::Variances(covvec);
        auto f = std::make_shared<BetweenFactor<imuBias::ConstantBias> >(
            b1, b2, imuBias::ConstantBias(), cov);
        newgraph.add(f);
        initialEstimate.insert(biasKey, imuBias::ConstantBias());
      }
```

每添加一帧，进行一次预积分，并把预积分量清空：

```
// Predict acceleration and gyro measurements in (actual) body frame
      // 把加速度转到body系下（减去重力分量
      Vector3 measuredAcc = scenario.acceleration_b(t) -
                            scenario.rotation(t).transpose() * params->n_gravity
      Vector3 measuredOmega = scenario.omega_b(t);
      // 预积分量
      accum.integrateMeasurement(measuredAcc, measuredOmega, delta_t);

      // Add Imu Factor  添加IMU因子
      ImuFactor imufac(X(i - 1), V(i - 1), X(i), V(i), biasKey, accum);
      newgraph.add(imufac);

      // insert new velocity, which is wrong  加入新的速度值（错误的）
      initialEstimate.insert(V(i), n_velocity);
      // initialEstimate.insert(V(i), scenario.velocity_b(t));

      accum.resetIntegration();   //清空预积分量
```

打印输出：

```
// Incremental solution
    isam.update(newgraph, initialEstimate);
    result = isam.calculateEstimate();
    newgraph = NonlinearFactorGraph();
    initialEstimate.clear();
  }
  GTSAM_PRINT(result);
```