Homework 3

[Question 1]

Prove:

$$\frac{\partial r_{\Delta p_{ij}}}{\partial \phi_i} = \left(\mathbf{R}_i^\top \left(\mathbf{p}_j - \mathbf{p}_i - \mathbf{v}_i \Delta t_{ij} - \frac{1}{2}\mathbf{g}\Delta t_{ij}^2\right)\right)^\wedge .$$

Proof

$r_{\Delta p_{ij}} := R_i^\top \left(p_j - p_i - v_i \Delta t_{ij} - \frac{1}{2} g \Delta t_{ij}^2\right) - \Delta \tilde{p}_{ij}.$

$\Rightarrow$ Applying right perturbation on R_i

$r_{\Delta p_{ij}}(R_i^\top Exp(\phi)) := (R_i Exp(\phi))^\top \left(p_j - p_i - v_i \Delta t_{ij} - \frac{1}{2} g \Delta t_{ij}^2\right) - \Delta \tilde{p}_{ij}.$

$\approx (I - \delta\phi^\wedge) R_i^\top \left(p_j - p_i - v_i \Delta t_{ij} - \frac{1}{2} g \Delta t_{ij}^2\right) - \Delta \tilde{p}_{ij}.$

$= (R_i^\top \left(p_j - p_i - v_i \Delta t_{ij} - \frac{1}{2} g \Delta t_{ij}^2\right) - \Delta \tilde{p}_{ij}) - \delta\phi^\wedge R_i^\top \left(p_j - p_i - v_i \Delta t_{ij} - \frac{1}{2} g \Delta t_{ij}^2\right)$

$= r_{\Delta p_{ij}} + [R_i^\top \left(p_j - p_i - v_i \Delta t_{ij} - \frac{1}{2} g \Delta t_{ij}^2\right)]^\wedge \delta\phi$

$\Rightarrow$

$$\frac{\partial r_{\Delta p_{ij}}}{\partial \phi_i} = \left(\mathbf{R}_i^\top \left(\mathbf{p}_j - \mathbf{p}_i - \mathbf{v}_i \Delta t_{ij} - \frac{1}{2}\mathbf{g}\Delta t_{ij}^2\right)\right)^\wedge .$$

[Question 2] - Implement Pre-Integration Graph Optimization Triggered by Odom based on G2O

Program Architecture

Upon receiving a new GNSS or odom update:

1. Create a new frame object.
2. Get an estimate of the current frame by $last frame \oplus imupreintegration$
3. Optimize
    - Add regular edges, like Prior edges, and pre-integration edges. Some examples include:
        - V: IMU Pre-integration Edge, between 2 GNSS frames `r_{\Delta v_{i,j}} = R_i^T (v_j - v_i - g \Delta t_{i,j}) - \Delta \tilde{v}_{ij}       - For Update, the Jacobian of the residual w.r.t velocity_estimate is $-R_i^T$ and $R_i^T$ for Frame A and Frame B respectively`
        - R:

        * IMU Pre-integration Edge() `const SO3 dR = preint_->GetDeltaRotation(bg); eR = dR^{-1} * R_i^T * R_j       J_R`
        * EdgePriorPoseNavState `const Vec3d er = SO3(state_.R_.matrix().transpose() * vp->estimate().so3().matrix()).log();       // Jacobian:       const Vec3d er = SO3(state_.R_.matrix().transpose() * vp->estimate().so3().matrix()).log();       _jacobianOplus[0].block<3, 3>(0, 0) = SO3::jr_inv(er);    // dr/dr`
    - If this frame is a GNSS update: add an GNSS edge
    - If this frame is an Odom update: add an odom edge (`current_frame_velocity_estimate - v_linear_odom)``
4. Update the current frame

Here is a more comprehensive list of edges and vertices associated

Here is the final result

Sorry, you have to paste it in your browser to view

```
<p align="center">
   <figure>
        <img src="https://github.com/user-attachments/assets/7e83481b-dd78-4b0d-93b4-80c0f0f
   </figure>
</p>
```

Here is code that changed

- Odom processing function so they are fired properly: `run_gins_pre_integ.cc`

```
.SetOdomProcessFunc([&](const sad::Odom& odom) {
    imu_init.AddOdom(odom);

    if (imu_inited && gnss_inited) {
        gins.AddOdom(odom);
    }
    auto state = gins.GetState();
    save_result(fout, state);
    if (ui) {
        ui->UpdateNavState(state);
        usleep(1e3);
    }
})
```

- To get ready, we need to add some state methods and attributes in `gins_pre_integ.h`:

```
Vec3d odom_vel_world(const sad::Odom& odom, std::shared_ptr<NavStated> frame) const;
Odom last_odom_;  //   odom
Odom this_odom_;  //   odom
bool last_odom_set_ = false;
```

```
bool gnss_opt = false;  //?
bool odom_opt = false;  //?
```

- The integration part is the main source of changes

```
void GinsPreInteg::AddGnss(const GNSS& gnss) {
    // this frame is created here!!
    this_frame_ = std::make_shared<NavStated>(current_time_);
    this_gnss_ = gnss;

    if (!first_gnss_received_) {
        if (!gnss.heading_valid_) {
            //   GNSS
            return;
        }
        //  gnss   pose  gnss
        this_frame_->timestamp_ = gnss.unix_time_;
        this_frame_->p_ = gnss.utm_pose_.translation();
        this_frame_->R_ = gnss.utm_pose_.so3();
        this_frame_->v_.setZero();
        this_frame_->bg_ = options_.preinteg_options_.init_bg_;
        this_frame_->ba_ = options_.preinteg_options_.init_ba_;

        pre_integ_ = std::make_shared<IMUPreintegration>(options_.preinteg_options_);

        last_frame_ = this_frame_;
        last_gnss_ = this_gnss_;
        first_gnss_received_ = true;
        current_time_ = gnss.unix_time_;
        return;
    }
    //   GNSS
    pre_integ_->Integrate(last_imu_, gnss.unix_time_ - current_time_);
    current_time_ = gnss.unix_time_;
    *this_frame_ = pre_integ_->Predict(*last_frame_, options_.gravity_);
    gnss_opt = true;
    Optimize();
    last_frame_ = this_frame_;
    last_gnss_ = this_gnss_;
}

Vec3d GinsPreInteg::odom_vel_world(const sad::Odom& odom, std::shared_ptr<NavStated> frame)
    double velo_l = options_.wheel_radius_ * odom.left_pulse_ / options_.circle_pulse_ * 2 *
    double velo_r =
        options_.wheel_radius_ * odom.right_pulse_ / options_.circle_pulse_ * 2 * M_PI / opt
    double average_vel = 0.5 * (velo_l + velo_r);
```

```
    Vec3d odom_world = frame -> R_ * Vec3d(average_vel, 0, 0);
    return odom_world;
};

void GinsPreInteg::AddOdom(const sad::Odom& odom) {
    // We wait for the first GNSS to come. Not the best, but easy
    if (!first_gnss_received_) return;

    // creating a current frame from the last frame
    // Setting odom, flags for optimization
    // this_frame_ = std::make_shared<NavStated>(current_time_);
    this_frame_ = std::make_shared<NavStated>(*last_frame_);
    this_frame_->timestamp_ = current_time_;
    this_odom_ = odom;

    if (!first_odom_received_ && first_gnss_received_){
        //  gnss   pose  gnss
        //TODO
        std::cout<<"3"<<std::endl;
        this_frame_->timestamp_ = odom.timestamp_;
        this_frame_->v_ = odom_vel_world(odom, this_frame_);

        pre_integ_ = std::make_shared<IMUPreintegration>(options_.preinteg_options_);
        last_frame_ = this_frame_;
        last_odom_ = this_odom_;
        first_odom_received_ = true;
        current_time_ = odom.timestamp_;
        return;
    }

    pre_integ_->Integrate(last_imu_, odom.timestamp_ - current_time_);

    current_time_ = odom.timestamp_;
    *this_frame_ = pre_integ_->Predict(*last_frame_, options_.gravity_);

    odom_opt = true;
    Optimize();

    last_odom_ = odom;
    last_odom_set_ = true;
    last_frame_ = this_frame_;
}

// Just optimize two frames. In GNSS triggered system, last_frame is updated with GNSS
void GinsPreInteg::Optimize() {
    if (pre_integ_->dt_ < 1e-2) {
```

```
    //    . Increased time intervals so gnss and odom are not too close to each other
    return;
}

using BlockSolverType = g2o::BlockSolverX;
using LinearSolverType = g2o::LinearSolverEigen<BlockSolverType::PoseMatrixType>;

auto* solver = new g2o::OptimizationAlgorithmLevenberg(
    g2o::make_unique<BlockSolverType>(g2o::make_unique<LinearSolverType>()));
g2o::SparseOptimizer optimizer;
optimizer.setAlgorithm(solver);

//     pose, v, bg, ba
auto v0_pose = new VertexPose();
v0_pose->setId(0);
v0_pose->setEstimate(last_frame_->GetSE3());
optimizer.addVertex(v0_pose);

auto v0_vel = new VertexVelocity();
v0_vel->setId(1);
v0_vel->setEstimate(last_frame_->v_);
optimizer.addVertex(v0_vel);

auto v0_bg = new VertexGyroBias();
v0_bg->setId(2);
v0_bg->setEstimate(last_frame_->bg_);
optimizer.addVertex(v0_bg);

auto v0_ba = new VertexAccBias();
v0_ba->setId(3);
v0_ba->setEstimate(last_frame_->ba_);
optimizer.addVertex(v0_ba);

//    pose, v, bg, ba
auto v1_pose = new VertexPose();
v1_pose->setId(4);
v1_pose->setEstimate(this_frame_->GetSE3());
optimizer.addVertex(v1_pose);

auto v1_bg = new VertexGyroBias();
v1_bg->setId(6);
v1_bg->setEstimate(this_frame_->bg_);
optimizer.addVertex(v1_bg);

auto v1_ba = new VertexAccBias();
v1_ba->setId(7);
```

```
v1_ba->setEstimate(this_frame_->ba_);
optimizer.addVertex(v1_ba);

auto v1_vel = new VertexVelocity();
v1_vel->setId(5);
v1_vel->setEstimate(this_frame_->v_);
optimizer.addVertex(v1_vel);

//
// Given the current bg, and ba, and we know the start and end time, calculate r_{delta
auto edge_inertial = new EdgeInertial(pre_integ_, options_.gravity_);
edge_inertial->setVertex(0, v0_pose);
edge_inertial->setVertex(1, v0_vel);
edge_inertial->setVertex(2, v0_bg);
edge_inertial->setVertex(3, v0_ba);
edge_inertial->setVertex(4, v1_pose);
edge_inertial->setVertex(5, v1_vel);
auto* rk = new g2o::RobustKernelHuber();
rk->setDelta(200.0);
edge_inertial->setRobustKernel(rk);
optimizer.addEdge(edge_inertial);

//
auto* edge_gyro_rw = new EdgeGyroRW();
edge_gyro_rw->setVertex(0, v0_bg);
edge_gyro_rw->setVertex(1, v1_bg);
edge_gyro_rw->setInformation(options_.bg_rw_info_);
optimizer.addEdge(edge_gyro_rw);

auto* edge_acc_rw = new EdgeAccRW();
edge_acc_rw->setVertex(0, v0_ba);
edge_acc_rw->setVertex(1, v1_ba);
edge_acc_rw->setInformation(options_.ba_rw_info_);
optimizer.addEdge(edge_acc_rw);

//    . it calculates err_r, err_p, etc. err_r = (last_pose.rotation^T * v0_pose).log()
// TODO: should this be new EdgePriorPoseNavState(*this_frame_, prior_info_)
// Oh no, it just measures the difference of the current and previous estimates of the
auto* edge_prior = new EdgePriorPoseNavState(*last_frame_, prior_info_);
edge_prior->setVertex(0, v0_pose);
edge_prior->setVertex(1, v0_vel);
edge_prior->setVertex(2, v0_bg);
edge_prior->setVertex(3, v0_ba);
optimizer.addEdge(edge_prior);

// Rico: Here, you either add an GNSS edge, or an Odom edge. There is no
```

```
// need to add both for a single frame
if (gnss_opt){

    // GNSS
    auto edge_gnss0 = new EdgeGNSS(v0_pose, last_gnss_.utm_pose_);
    edge_gnss0->setInformation(options_.gnss_info_);
    optimizer.addEdge(edge_gnss0);

    auto edge_gnss1 = new EdgeGNSS(v1_pose, this_gnss_.utm_pose_);
    edge_gnss1->setInformation(options_.gnss_info_);
    optimizer.addEdge(edge_gnss1);
    gnss_opt = false;
} else if (odom_opt){
    // Odom
    Vec3d vel_world0 = odom_vel_world(last_odom_, last_frame_);
    Vec3d vel_world1 = odom_vel_world(this_odom_, this_frame_);

    auto v0_vel = new VertexVelocity();
    v0_vel->setId(5);
    v0_vel->setEstimate(last_frame_->v_);
    optimizer.addVertex(v0_vel);

    EdgeEncoder3D* edge_odom0 = nullptr;
    edge_odom0 = new EdgeEncoder3D(v0_vel, vel_world0);
    edge_odom0->setInformation(options_.odom_info_);
    optimizer.addEdge(edge_odom0);
    EdgeEncoder3D* edge_odom1 = nullptr;
    edge_odom1 = new EdgeEncoder3D(v1_vel, vel_world1);
    edge_odom1->setInformation(options_.odom_info_);
    optimizer.addEdge(edge_odom1);

    odom_opt = false;
}


optimizer.setVerbose(options_.verbose_);
optimizer.initializeOptimization();
optimizer.optimize(20);

last_frame_->R_ = v0_pose->estimate().so3();
last_frame_->p_ = v0_pose->estimate().translation();
last_frame_->v_ = v0_vel->estimate();
last_frame_->bg_ = v0_bg->estimate();
last_frame_->ba_ = v0_ba->estimate();

this_frame_->R_ = v1_pose->estimate().so3();
```

```
    this_frame_->p_ = v1_pose->estimate().translation();
    this_frame_->v_ = v1_vel->estimate();
    this_frame_->bg_ = v1_bg->estimate();
    this_frame_->ba_ = v1_ba->estimate();

    //  integ
    options_.preinteg_options_.init_bg_ = this_frame_->bg_;
    options_.preinteg_options_.init_ba_ = this_frame_->ba_;
    pre_integ_ = std::make_shared<IMUPreintegration>(options_.preinteg_options_);
}
```

[Question 3] - Show The Correctness of Jacobians of Residuals Calculating Using The Auto Differentiation Functionality in G2O

So the Jacobians in question, are the ones defined here:

$$
\begin{aligned}
\frac{\partial r_{\Delta p_{ij}}}{\partial p_i} &= -R_i^\top, \\
\frac{\partial r_{\Delta p_{ij}}}{\partial p_j} &= R_i^\top, \\
\frac{\partial r_{\Delta p_{ij}}}{\partial v_i} &= -R_i^\top \Delta t_{ij}, \\
\frac{\partial r_{\Delta p_{ij}}}{\partial \phi_i} &= \left(R_i^\top \left(p_j - p_i - v_i \Delta t_{ij} - \frac{1}{2} g \Delta t_{ij}^2\right)\right)^\wedge. \\
\frac{\partial r_{\Delta v_{i,j}}}{\partial b_{a,i}} &= \sum_{k=i}^{j-1} \Delta \tilde{R}_{ik} \Delta t, \\
\frac{\partial r_{\Delta v_{i,j}}}{\partial b_{g,i}} &= \sum_{k=i}^{j-1} \Delta \tilde{R}_{ik} \left(\tilde{a}_k - b_{a,i}\right)^\wedge \frac{\partial \Delta \tilde{R}_{ik}}{\partial b_{g,i}} \Delta t. \\
\frac{\partial r_{\Delta p_{ij}}}{\partial p_i} &= -R_i^\top, \\
\frac{\partial r_{\Delta p_{ij}}}{\partial p_j} &= R_i^\top, \\
\frac{\partial r_{\Delta p_{ij}}}{\partial v_i} &= -R_i^\top \Delta t_{ij}, \\
\frac{\partial r_{\Delta p_{ij}}}{\partial \phi_i} &= \left(R_i^\top \left(p_j - p_i - v_i \Delta t_{ij} - \frac{1}{2} g \Delta t_{ij}^2\right)\right)^\wedge. \\
&\dots
\end{aligned}
$$

To test to see if they are correct, we just need to comment out the overriding function `void EdgeInertial::linearizeOplus()`. This will enable automatic differentiation.

```
// gins_pre_integ.cc
void GinsPreInteg::Optimize() {
    ...
    // At the end of the function
    edge_inertial->checkJacobians();  //
}

// g2o_types.cc
void EdgeInertial::checkJacobians() {
    linearizeOplus();

    auto* p1 = dynamic_cast<const VertexPose*>(_vertices[0]);
    auto* v1 = dynamic_cast<const VertexVelocity*>(_vertices[1]);
    auto* bg1 = dynamic_cast<const VertexGyroBias*>(_vertices[2]);
    auto* ba1 = dynamic_cast<const VertexAccBias*>(_vertices[3]);
    auto* p2 = dynamic_cast<const VertexPose*>(_vertices[4]);
    auto* v2 = dynamic_cast<const VertexVelocity*>(_vertices[5]);

    Vec3d bg = bg1->estimate();
    Vec3d ba = ba1->estimate();
    Vec3d dbg = bg - preint_->bg_;

    //
    const SO3 R1 = p1->estimate().so3();
    const SO3 R1T = R1.inverse();
    const SO3 R2 = p2->estimate().so3();

    auto dR_dbg = preint_->dR_dbg_;
    auto dv_dbg = preint_->dV_dbg_;
    auto dp_dbg = preint_->dP_dbg_;
    auto dv_dba = preint_->dV_dba_;
    auto dp_dba = preint_->dP_dba_;

    //
    Vec3d vi = v1->estimate();
    Vec3d vj = v2->estimate();
    Vec3d pi = p1->estimate().translation();
    Vec3d pj = p2->estimate().translation();

    const SO3 dR = preint_->GetDeltaRotation(bg);
    const SO3 eR = SO3(dR).inverse() * R1T * R2;
    const Vec3d er = eR.log();
```

```
    const Mat3d invJr = SO3::jr_inv(eR); //

    Eigen::Matrix3d dR_dR1 = -invJr * (R2.inverse() * R1).matrix();
    Eigen::Matrix3d dv_dR1 = SO3::hat(R1T * (vj - vi - grav_ * dt_));
    Eigen::Matrix3d dp_dR1 = SO3::hat(R1T * (pj - pi - v1->estimate() * dt_ - 0.5 * grav_ *
    Eigen::Matrix3d dp_dp1 = -R1T.matrix();
    Eigen::Matrix3d dv_dv1 = -R1T.matrix();
    Eigen::Matrix3d dp_dv1 = -R1T.matrix() * dt_;
    Eigen::Matrix3d dR_dbg1 = -invJr * eR.inverse().matrix() * SO3::jr((dR_dbg * dbg).eval()
    Eigen::Matrix3d dv_dbg1 = -dv_dbg;
    Eigen::Matrix3d dp_dbg1 = -dp_dbg;
    Eigen::Matrix3d dv_dba1 = -dv_dba;
    Eigen::Matrix3d dp_dba1 = -dp_dba;
    Eigen::Matrix3d dr_dr2 = invJr;
    Eigen::Matrix3d dp_dp2 = R1T.matrix();
    Eigen::Matrix3d dv_dv2 = R1T.matrix();

    LOG(INFO) << "dR_dR1 diff: " << (_jacobianOplus[0].block<3, 3>(0, 0) - dR_dR1).lpNorm<Ei
    LOG(INFO) << "dv_dR1 diff: " << (_jacobianOplus[0].block<3, 3>(3, 0) - dv_dR1).lpNorm<Ei
    LOG(INFO) << "dp_dR1 diff: " << (_jacobianOplus[0].block<3, 3>(6, 0) - dp_dR1).lpNorm<Ei
    LOG(INFO) << "dp_dp1 diff: " << (_jacobianOplus[0].block<3, 3>(6, 3) - dp_dp1).lpNorm<Ei
    LOG(INFO) << "dv_dv1 diff: " << (_jacobianOplus[1].block<3, 3>(3, 0) - dv_dv1).lpNorm<Ei
    LOG(INFO) << "dp_dv1 diff: " << (_jacobianOplus[1].block<3, 3>(6, 0) - dp_dv1).lpNorm<Ei
    LOG(INFO) << "dR_dbg1 diff: " << (_jacobianOplus[2].block<3, 3>(0, 0) - dR_dbg1).lpNorm<
    LOG(INFO) << "dv_dbg1 diff: " << (_jacobianOplus[2].block<3, 3>(3, 0) - dv_dbg1).lpNorm<
    LOG(INFO) << "dp_dbg1 diff: " << (_jacobianOplus[2].block<3, 3>(6, 0) - dp_dbg1).lpNorm<
    LOG(INFO) << "dv_dba1 diff: " << (_jacobianOplus[3].block<3, 3>(3, 0) - dv_dba1).lpNorm<
    LOG(INFO) << "dp_dba1 diff: " << (_jacobianOplus[3].block<3, 3>(6, 0) - dp_dba1).lpNorm<
    LOG(INFO) << "dr_dr2 diff: " << (_jacobianOplus[4].block<3, 3>(0, 0) - dr_dr2).lpNorm<Ei
    LOG(INFO) << "dp_dp2 diff: " << (_jacobianOplus[4].block<3, 3>(6, 3) - dp_dp2).lpNorm<Ei
    LOG(INFO) << "dv_dv2 diff: " << (_jacobianOplus[5].block<3, 3>(3, 0) - dv_dv2).lpNorm<Ei
};
```

We can see that the final differences are in the order of 10^{-7}!