Exercises

Find $\frac{\partial R^{-1}p}{\partial R}$ using left and right perturbations

Right Perturbation:

$$\begin{aligned}
\frac{\partial R^{-1}p}{\partial R} &= \lim_{\phi\to 0}\frac{(Rexp(\phi^\wedge))^{-1}p - R^{-1}p}{\phi} \\
&= \lim_{\phi\to 0}\frac{exp(\phi^\wedge)^{-1}R^{-1}p - R^{-1}p}{\phi} \\
&= \lim_{\phi\to 0}\frac{exp(-\phi^\wedge)R^{-1}p - R^{-1}p}{\phi} \\
&\approx \lim_{\phi\to 0}\frac{(I-\phi^\wedge)R^{-1}p - R^{-1}p}{\phi} \\
&= \lim_{\phi\to 0}\frac{-\phi^\wedge R^{-1}p}{\phi} \\
&= \lim_{\phi\to 0}\frac{(R^{-1}p)^\wedge\phi}{\phi} \\
&= (R^{-1}p)^\wedge
\end{aligned}$$

Left Perturbation:

$$\begin{aligned}
\frac{\partial R^{-1}p}{\partial R} &= \lim_{\phi\to 0}\frac{(exp(\phi^\wedge)R)^{-1}p - R^{-1}p}{\phi} \\
\frac{\partial R^{-1}p}{\partial R} &= \lim_{\phi\to 0}\frac{R^{-1}exp(\phi^\wedge)^{-1}p - R^{-1}p}{\phi} \\
\frac{\partial R^{-1}p}{\partial R} &= \lim_{\phi\to 0}\frac{R^{-1}exp(-\phi^\wedge)p - R^{-1}p}{\phi} \\
\frac{\partial R^{-1}p}{\partial R} &\approx \lim_{\phi\to 0}\frac{R^{-1}(I-\phi^\wedge)p - R^{-1}p}{\phi} \\
\frac{\partial R^{-1}p}{\partial R} &= \lim_{\phi\to 0}\frac{-R^{-1}\phi^\wedge p}{\phi} \\
\frac{\partial R^{-1}p}{\partial R} &= \lim_{\phi\to 0}\frac{R^{-1}p^\wedge\phi}{\phi} \\
&= R^{-1}p^\wedge
\end{aligned}$$

Find $\frac{\partial R_1 R_2^{-1}}{\partial R_2}$ using left and right perturbations

I'm not sure about $\lim_{\phi\to 0}\frac{-\phi^\wedge}{\phi}$

- Right Perturbation:

$$
\begin{aligned}
\frac{\partial R_1 R_2^{-1}}{\partial R_2} &= \lim_{\phi \to 0} \frac{R_1 (R_2 exp(\phi^\wedge))^{-1} - R_1 R_2^{-1}}{\phi} \\
&= \lim_{\phi \to 0} \frac{R_1 exp(-\phi^\wedge) R_2^{-1} - R_1 R_2^{-1}}{\phi} \\
&= \lim_{\phi \to 0} \frac{R_1 R_2^T R_2 exp(-\phi^\wedge) R_2^T - R_1 R_2^T}{\phi} \\
&= \lim_{\phi \to 0} \frac{R_1 R_2^T exp(-R_2 \phi^\wedge) - R_1 R_2^T}{\phi} \\
&\approx \lim_{\phi \to 0} \frac{R_1 R_2^T (I - R_2 \phi^\wedge) - R_1 R_2^T}{\phi} \\
&= \lim_{\phi \to 0} \frac{-R_1 \phi^\wedge}{\phi} \\
&= -R_1
\end{aligned}
$$

Left Perturbation:

$$
\begin{aligned}
\frac{\partial R_1 R_2^{-1}}{\partial R_2} &= \lim_{\phi \to 0} \frac{R_1 (exp(\phi^\wedge) R_2)^{-1} - R_1 R_2^{-1}}{\phi} \\
&= \lim_{\phi \to 0} \frac{R_1 R_2^{-1} exp(-\phi^\wedge) - R_1 R_2^{-1}}{\phi} \\
&= \lim_{\phi \to 0} \frac{R_1 R_2^{-1} (I - \phi^\wedge) - R_1 R_2^{-1}}{\phi} \\
&= \lim_{\phi \to 0} \frac{-R_1 R_2^{-1} \phi^\wedge}{\phi} \\
&= -R_1 R_2^{-1}
\end{aligned}
$$

Programming Exercise

```
//
// Created by xiang on 22-12-29. Modified by Rico 2024-12-19
//

#include <gflags/gflags.h>
#include <glog/logging.h>

#include "common/eigen_types.h"
#include "common/math_utils.h"
#include "tools/ui/pangolin_window.h"

///
///        flags
```

```
DEFINE_double(angular_velocity, 10.0, "    ");
DEFINE_double(linear_velocity, 5.0, "     m/s");
DEFINE_bool(use_quaternion, false, "     ");

int main(int argc, char** argv) {
    google::InitGoogleLogging(argv[0]);
    FLAGS_stderrthreshold = google::INFO;
    FLAGS_colorlogtostderr = true;
    google::ParseCommandLineFlags(&argc, &argv, true);

    ///
    sad::ui::PangolinWindow ui;
    if (ui.Init() == false) {
        return -1;
    }

    double angular_velocity_rad = FLAGS_angular_velocity * sad::math::kDEG2RAD;  //
    double z_acc = -0.1;
    SE3 pose;                                                                     // TWB
    Vec3d omega(0, 0, angular_velocity_rad);                                      //
    Vec3d v_body(FLAGS_linear_velocity, 0, 0);                                    //
    const double dt = 0.05;                                                       //

    while (ui.ShouldQuit() == false) {
        //
        Vec3d v_world = pose.so3() * v_body;
        pose.translation() += v_world * dt;

        //
        if (FLAGS_use_quaternion) {
            // theta is halved in the quaternion world
            Quatd q = pose.unit_quaternion() * Quatd(1, 0.5 * omega[0] * dt, 0.5 * omega[1]
            // Quatd q = pose.unit_quaternion() * Quatd(std::cos(0.5 * angular_velocity_rad
            q.normalize();
            // auto& quat = q;
            // std::cout << "=========Quaternion coefficients: "
            //    << "w = " << quat.w() << ", "
            //    << "x = " << quat.x() << ", "
            //    << "y = " << quat.y() << ", "
            //    << "z = " << quat.z() << std::endl;
            pose.so3() = SO3(q);
        } else {
            pose.so3() = pose.so3() * SO3::exp(omega * dt);
        }
        v_body += Vec3d(0, 0, z_acc * dt);
```

```
        LOG(INFO) << "pose: " << pose.translation().transpose();
        ui.UpdateNavState(sad::NavStated(0, pose, v_world));

        usleep(dt * 1e6);
    }

    ui.Quit();
    return 0;
}
```

Below is from my blogpost

Gauss-Newton Optimization

In Gauss Newton, we specifically look at minimizing a least squares problem. Assume we have a:

- scalar-valued cost function $c(x)$,
- vector-valued function: $f(x)$, `[m, 1]`
- Jacobian J_0 at x_0 is consequently `[m, n]`
- Hessian H is $D^2c(x)$. It's approximated as J^TJ

$$
\begin{aligned}
&c(x) = |f(x)^2| \\
&x* = argmin(|f(x)^2|)
\end{aligned}
$$

First order Taylor Expansion:

$$
\begin{aligned}
&argmin_{\Delta x}(|f(x+\Delta x)^2|) \\
&= argmin_{\Delta x}[(f(x_0) + J_0\Delta x)^T(f(x_0) + J_0\Delta x)] \\
&= argmin_{\Delta x}[f(x_0)^Tf(x_0) + f(x_0)^TJ_0\Delta x + (J_0\Delta x)^Tf(x_0) + (J_0\Delta x)^T(J_0\Delta x)] \\
&= argmin_{\Delta x}[f(x_0)^Tf(x_0) + 2f(x_0)^TJ_0\Delta x + (J_0\Delta x)^T(J_0\Delta x)]
\end{aligned}
$$

Take the derivative of the above and set it to 0, we get

$$
\begin{aligned}
&\frac{\partial f(x+\Delta x)^2}{\partial \Delta x} = 2J_0^Tf(x_0) + [(J_0^TJ_0) + (J_0^TJ_0)^T]\Delta x \\
&= 2J_0^Tf(x_0) + 2(J_0^TJ_0)\Delta x \\
&= 0
\end{aligned}
$$

So we can solve for Δx with $H = J_0^TJ_0$, $b = -J_0^Tf(x_0)$:

$$
\begin{aligned}
&(J_0^TJ_0)\Delta x = -J_0^Tf(x_0) \\
&\rightarrow H\Delta x = g
\end{aligned}
$$

- Note: because J_0 may not have an inverse, here we cannot multiply J_0^{-1} to eliminate J_0^T
- In fact, to Δx is available if and only if H is **positive definite**.
- In least square, $f(x)$ is a.k.a residuals. Usually, it represents the **error between a data point and from its ground truth**.

In SLAM, we always frame this least squares problem with `e = [observered_landmark - predicted_landmark]` at each landmark. So all together, we want to **minimize the total least squares of the difference between observations and predictions.** In the meantime, at each landmark, there is an error covariance, so all together, there's an error matrix Σ. Here in cost calculation, we take Σ^{-1} so the **larger the error covariance, the lower the weight the corresponding difference gets.**

With $e(x + \Delta x) \approx e(x) + J\Delta x$,

$$
\begin{aligned}
&x* = argmin(|e^T\Sigma^{-1}e|) \\
&\rightarrow argmin_{\Delta x}(|e(x+\Delta x)^T\Sigma^{-1}e(x+\Delta x)|) \\
&\text{similar steps as above ...} \\
&\rightarrow (J_0^T\Sigma^{-1}J_0)\Delta x = -J_0^T\Sigma^{-1}f(x_0)
\end{aligned}
$$

Using Cholesky Decomposition, one can get $\Sigma^{-1} = A^TA$. Then we can write the above as

$$
((AJ_0)^T(AJ_0))\Delta x = -(AJ_0)^TAf(x_0)
$$

For a more detailed derivation, please see here

Levenberg-Marquardt (LM) Optimization

Again, **Taylor expansion** works better when Δx is small, so the function can be better estimated by it. So, similar to regularization techniques on step sizes in deep learning, like L1, L2 regularization, we can regularize the step size, Δx

$$
(H + \mu I_H)\Delta x = -J_0^Tf(x_0) = g
$$

Intuitively,

- as μ grows, the diagonal identity matrix μI_H grows, so $H + \mu I_H \rightarrow \mu I_H$. So, $\Delta x \approx (H + \mu I_H)^{-1}g = \frac{g}{\mu}$, which means Δx grows smaller. In the meantime, Δx will be similar to that in gradient descent.
- as μ becomes smaller, Δx will become more like Gauss-Newton. However, due to μI_H, $(H + \mu I_H)$ is positive semi-definite, which provides more stability for solving for Δx.