ДОДАТОК А
ІНФРАСТРУКТУРНА ТА ІНТЕЛЕКТУАЛЬНА СКЛАДОВА ДОДАТКУ

```
import sys
import pygame  # pylint: disable=import-error
import cv2  # pylint: disable=import-error

class Window:
  def __init__(self, w, h, caption="window", double=False,
halve=False):
    self.w = w
    self.h = h
    pygame.display.init()
    pygame.display.set_caption(caption)
    self.double = double
    self.halve = halve
    if self.double:
      self.rw, self.rh = w*2, h*2
    elif self.halve:
      self.rw, self.rh = w//2, h//2
    else:
      self.rw, self.rh = w, h
    self.screen = pygame.display.set_mode((self.rw,
self.rh))
    pygame.display.flip()

    # hack for xmonad, it shrinks the window by 6 pixels
after the display.flip
    if self.screen.get_width() != self.rw:
      self.screen =
pygame.display.set_mode((self.rw+(self.rw-
self.screen.get_width()), self.rh+(self.rh-
self.screen.get_height())))
      pygame.display.flip()
```

```
    def draw(self, out):
      pygame.event.pump()
      if self.double:
        out2 = cv2.resize(out, (self.w*2, self.h*2))
        pygame.surfarray.blit_array(self.screen,
out2.swapaxes(0, 1))
      elif self.halve:
        out2 = cv2.resize(out, (self.w//2, self.h//2))
        pygame.surfarray.blit_array(self.screen,
out2.swapaxes(0, 1))
      else:
        pygame.surfarray.blit_array(self.screen,
out.swapaxes(0, 1))
      pygame.display.flip()

    def getkey(self):
      while 1:
        event = pygame.event.wait()
        if event.type == pygame.QUIT:
          pygame.quit()
          sys.exit()
        if event.type == pygame.KEYDOWN:
          return event.key

    def getclick(self):
      for event in pygame.event.get():
        if event.type == pygame.MOUSEBUTTONDOWN:
          mx, my = pygame.mouse.get_pos()
          return mx, my

  if __name__ == "__main__":
    import numpy as np
    win = Window(200, 200, double=True)
    img: np.ndarray = np.zeros((200, 200, 3), np.uint8)
    while 1:
```

```
        print("draw")
        img += 1
        win.draw(img)
    #include <sys/time.h>
    #include <sys/resource.h>

    #include <cmath>

    #include "locationd.h"

    using namespace EKFS;
    using namespace Eigen;

    ExitHandler do_exit;
    const double ACCEL_SANITY_CHECK = 100.0;  // m/s^2
    const double ROTATION_SANITY_CHECK = 10.0;  // rad/s
    const double TRANS_SANITY_CHECK = 200.0;  // m/s
    const double CALIB_RPY_SANITY_CHECK = 0.5; // rad (+- 30
deg)
    const double ALTITUDE_SANITY_CHECK = 10000; // m
    const double MIN_STD_SANITY_CHECK = 1e-5; // m or rad
    const double VALID_TIME_SINCE_RESET = 1.0; // s
    const double VALID_POS_STD = 50.0; // m
    const double MAX_RESET_TRACKER = 5.0;
    const double SANE_GPS_UNCERTAINTY = 1500.0; // m

    static VectorXd floatlist2vector(const capnp::List<float,
capnp::Kind::PRIMITIVE>::Reader& floatlist) {
      VectorXd res(floatlist.size());
      for (int i = 0; i < floatlist.size(); i++) {
        res[i] = floatlist[i];
      }
      return res;
    }
```

```
    static Vector4d quat2vector(const Quaterniond& quat) {
      return Vector4d(quat.w(), quat.x(), quat.y(), quat.z());
    }

    static Quaterniond vector2quat(const VectorXd& vec) {
      return Quaterniond(vec(0), vec(1), vec(2), vec(3));
    }

    static void
init_measurement(cereal::LiveLocationKalman::Measurement::Buil
der meas, const VectorXd& val, const VectorXd& std, bool
valid) {
      meas.setValue(kj::arrayPtr(val.data(), val.size()));
      meas.setStd(kj::arrayPtr(std.data(), std.size()));
      meas.setValid(valid);
    }


    static MatrixXdr rotate_cov(const MatrixXdr& rot_matrix,
const MatrixXdr& cov_in) {
      // To rotate a covariance matrix, the cov matrix needs
to multiplied left and right by the transform matrix
      return ((rot_matrix *  cov_in) *
rot_matrix.transpose());
    }

    static VectorXd rotate_std(const MatrixXdr& rot_matrix,
const VectorXd& std_in) {
      // Stds cannot be rotated like values, only covariances
can be rotated
      return rotate_cov(rot_matrix,
std_in.array().square().matrix().asDiagonal()).diagonal().arra
y().sqrt();
    }
```

```
    Localizer::Localizer() {
      this->kf = std::make_unique<LiveKalman>();
      this->reset_kalman();

      this->calib = Vector3d(0.0, 0.0, 0.0);
      this->device_from_calib = MatrixXdr::Identity(3, 3);
      this->calib_from_device = MatrixXdr::Identity(3, 3);

      for (int i = 0; i < POSENET_STD_HIST_HALF * 2; i++) {
        this->posenet_stds.push_back(10.0);
      }

      VectorXd ecef_pos = this->kf-
>get_x().segment<STATE_ECEF_POS_LEN>(STATE_ECEF_POS_START);
      this->converter = std::make_unique<LocalCoord>((ECEF) {
.x = ecef_pos[0], .y = ecef_pos[1], .z = ecef_pos[2] });
    }

    void
Localizer::build_live_location(cereal::LiveLocationKalman::Bui
lder& fix) {
      VectorXd predicted_state = this->kf->get_x();
      MatrixXdr predicted_cov = this->kf->get_P();
      VectorXd predicted_std =
predicted_cov.diagonal().array().sqrt();

      VectorXd fix_ecef =
predicted_state.segment<STATE_ECEF_POS_LEN>(STATE_ECEF_POS_STA
RT);
      ECEF fix_ecef_ecef = { .x = fix_ecef(0), .y =
fix_ecef(1), .z = fix_ecef(2) };
      VectorXd fix_ecef_std =
predicted_std.segment<STATE_ECEF_POS_ERR_LEN>(STATE_ECEF_POS_E
RR_START);
```

```
      VectorXd vel_ecef =
predicted_state.segment<STATE_ECEF_VELOCITY_LEN>(STATE_ECEF_VE
LOCITY_START);
      VectorXd vel_ecef_std =
predicted_std.segment<STATE_ECEF_VELOCITY_ERR_LEN>(STATE_ECEF_
VELOCITY_ERR_START);
      VectorXd fix_pos_geo_vec = this-
>get_position_geodetic();
      VectorXd orientation_ecef =
quat2euler(vector2quat(predicted_state.segment<STATE_ECEF_ORIE
NTATION_LEN>(STATE_ECEF_ORIENTATION_START)));
      VectorXd orientation_ecef_std =
predicted_std.segment<STATE_ECEF_ORIENTATION_ERR_LEN>(STATE_EC
EF_ORIENTATION_ERR_START);
      MatrixXdr orientation_ecef_cov =
predicted_cov.block<STATE_ECEF_ORIENTATION_ERR_LEN,
STATE_ECEF_ORIENTATION_ERR_LEN>(STATE_ECEF_ORIENTATION_ERR_STA
RT, STATE_ECEF_ORIENTATION_ERR_START);
      MatrixXdr device_from_ecef =
euler2rot(orientation_ecef).transpose();
      VectorXd calibrated_orientation_ecef = rot2euler((this-
>calib_from_device * device_from_ecef).transpose());

      VectorXd acc_calib = this->calib_from_device *
predicted_state.segment<STATE_ACCELERATION_LEN>(STATE_ACCELERA
TION_START);
      MatrixXdr acc_calib_cov =
predicted_cov.block<STATE_ACCELERATION_ERR_LEN,
STATE_ACCELERATION_ERR_LEN>(STATE_ACCELERATION_ERR_START,
STATE_ACCELERATION_ERR_START);
      VectorXd acc_calib_std = rotate_cov(this-
>calib_from_device, acc_calib_cov).diagonal().array().sqrt();
      VectorXd ang_vel_calib = this->calib_from_device *
predicted_state.segment<STATE_ANGULAR_VELOCITY_LEN>(STATE_ANGU
LAR_VELOCITY_START);
```

```
      MatrixXdr vel_angular_cov =
predicted_cov.block<STATE_ANGULAR_VELOCITY_ERR_LEN,
STATE_ANGULAR_VELOCITY_ERR_LEN>(STATE_ANGULAR_VELOCITY_ERR_STA
RT, STATE_ANGULAR_VELOCITY_ERR_START);
      VectorXd ang_vel_calib_std = rotate_cov(this-
>calib_from_device,
vel_angular_cov).diagonal().array().sqrt();

      VectorXd vel_device = device_from_ecef * vel_ecef;
      VectorXd device_from_ecef_eul =
quat2euler(vector2quat(predicted_state.segment<STATE_ECEF_ORIE
NTATION_LEN>(STATE_ECEF_ORIENTATION_START))).transpose();
      MatrixXdr condensed_cov(STATE_ECEF_ORIENTATION_ERR_LEN +
STATE_ECEF_VELOCITY_ERR_LEN, STATE_ECEF_ORIENTATION_ERR_LEN +
STATE_ECEF_VELOCITY_ERR_LEN);

condensed_cov.topLeftCorner<STATE_ECEF_ORIENTATION_ERR_LEN,
STATE_ECEF_ORIENTATION_ERR_LEN>() =
        predicted_cov.block<STATE_ECEF_ORIENTATION_ERR_LEN,
STATE_ECEF_ORIENTATION_ERR_LEN>(STATE_ECEF_ORIENTATION_ERR_STA
RT, STATE_ECEF_ORIENTATION_ERR_START);

condensed_cov.topRightCorner<STATE_ECEF_ORIENTATION_ERR_LEN,
STATE_ECEF_VELOCITY_ERR_LEN>() =
        predicted_cov.block<STATE_ECEF_ORIENTATION_ERR_LEN,
STATE_ECEF_VELOCITY_ERR_LEN>(STATE_ECEF_ORIENTATION_ERR_START,
STATE_ECEF_VELOCITY_ERR_START);

condensed_cov.bottomRightCorner<STATE_ECEF_VELOCITY_ERR_LEN,
STATE_ECEF_VELOCITY_ERR_LEN>() =
        predicted_cov.block<STATE_ECEF_VELOCITY_ERR_LEN,
STATE_ECEF_VELOCITY_ERR_LEN>(STATE_ECEF_VELOCITY_ERR_START,
STATE_ECEF_VELOCITY_ERR_START);
```

```
condensed_cov.bottomLeftCorner<STATE_ECEF_VELOCITY_ERR_LEN,
STATE_ECEF_ORIENTATION_ERR_LEN>() =
         predicted_cov.block<STATE_ECEF_VELOCITY_ERR_LEN,
STATE_ECEF_ORIENTATION_ERR_LEN>(STATE_ECEF_VELOCITY_ERR_START,
STATE_ECEF_ORIENTATION_ERR_START);
       VectorXd H_input(device_from_ecef_eul.size() +
vel_ecef.size());
       H_input << device_from_ecef_eul, vel_ecef;
       MatrixXdr HH = this->kf->H(H_input);
       MatrixXdr vel_device_cov = (HH * condensed_cov) *
HH.transpose();
       VectorXd vel_device_std =
vel_device_cov.diagonal().array().sqrt();

       VectorXd vel_calib = this->calib_from_device *
vel_device;
       VectorXd vel_calib_std = rotate_cov(this-
>calib_from_device, vel_device_cov).diagonal().array().sqrt();

       VectorXd orientation_ned =
ned_euler_from_ecef(fix_ecef_ecef, orientation_ecef);
       VectorXd orientation_ned_std = rotate_cov(this-
>converter->ecef2ned_matrix,
orientation_ecef_cov).diagonal().array().sqrt();
       VectorXd calibrated_orientation_ned =
ned_euler_from_ecef(fix_ecef_ecef,
calibrated_orientation_ecef);
       VectorXd nextfix_ecef = fix_ecef + vel_ecef;
       VectorXd ned_vel = this->converter->ecef2ned((ECEF) { .x
= nextfix_ecef(0), .y = nextfix_ecef(1), .z = nextfix_ecef(2)
}).to_vector() - converter-
>ecef2ned(fix_ecef_ecef).to_vector();
```

```
      VectorXd accDevice =
predicted_state.segment<STATE_ACCELERATION_LEN>(STATE_ACCELERA
TION_START);
      VectorXd accDeviceErr =
predicted_std.segment<STATE_ACCELERATION_ERR_LEN>(STATE_ACCELE
RATION_ERR_START);

      VectorXd angVelocityDevice =
predicted_state.segment<STATE_ANGULAR_VELOCITY_LEN>(STATE_ANGU
LAR_VELOCITY_START);
      VectorXd angVelocityDeviceErr =
predicted_std.segment<STATE_ANGULAR_VELOCITY_ERR_LEN>(STATE_AN
GULAR_VELOCITY_ERR_START);

      Vector3d nans = Vector3d(NAN, NAN, NAN);

      // TODO fill in NED and Calibrated stds
      // write measurements to msg
      init_measurement(fix.initPositionGeodetic(),
fix_pos_geo_vec, nans, this->gps_mode);
      init_measurement(fix.initPositionECEF(), fix_ecef,
fix_ecef_std, this->gps_mode);
      init_measurement(fix.initVelocityECEF(), vel_ecef,
vel_ecef_std, this->gps_mode);
      init_measurement(fix.initVelocityNED(), ned_vel, nans,
this->gps_mode);
      init_measurement(fix.initVelocityDevice(), vel_device,
vel_device_std, true);
      init_measurement(fix.initAccelerationDevice(),
accDevice, accDeviceErr, true);
      init_measurement(fix.initOrientationECEF(),
orientation_ecef, orientation_ecef_std, this->gps_mode);
      init_measurement(fix.initCalibratedOrientationECEF(),
calibrated_orientation_ecef, nans, this->calibrated && this-
>gps_mode);
```

```
      init_measurement(fix.initOrientationNED(),
orientation_ned, orientation_ned_std, this->gps_mode);
      init_measurement(fix.initCalibratedOrientationNED(),
calibrated_orientation_ned, nans, this->calibrated && this-
>gps_mode);
      init_measurement(fix.initAngularVelocityDevice(),
angVelocityDevice, angVelocityDeviceErr, true);
      init_measurement(fix.initVelocityCalibrated(),
vel_calib, vel_calib_std, this->calibrated);
      init_measurement(fix.initAngularVelocityCalibrated(),
ang_vel_calib, ang_vel_calib_std, this->calibrated);
      init_measurement(fix.initAccelerationCalibrated(),
acc_calib, acc_calib_std, this->calibrated);

      double old_mean = 0.0, new_mean = 0.0;
      int i = 0;
      for (double x : this->posenet_stds) {
        if (i < POSENET_STD_HIST_HALF) {
          old_mean += x;
        } else {
          new_mean += x;
        }
        i++;
      }
      old_mean /= POSENET_STD_HIST_HALF;
      new_mean /= POSENET_STD_HIST_HALF;
      // experimentally found these values, no false positives
in 20k minutes of driving
      bool std_spike = (new_mean / old_mean > 4.0 && new_mean
> 7.0);

      fix.setPosenetOK(!(std_spike && this->car_speed > 5.0));
      fix.setDeviceStable(!this->device_fell);
      fix.setExcessiveResets(this->reset_tracker >
MAX_RESET_TRACKER);
```

```
      this->device_fell = false;

      //fix.setGpsWeek(this->time.week);
      //fix.setGpsTimeOfWeek(this->time.tow);
      fix.setUnixTimestampMillis(this->unix_timestamp_millis);

      double time_since_reset = this->kf->get_filter_time() -
this->last_reset_time;
      fix.setTimeSinceReset(time_since_reset);
      if (fix_ecef_std.norm() < VALID_POS_STD && this-
>calibrated && time_since_reset > VALID_TIME_SINCE_RESET) {

fix.setStatus(cereal::LiveLocationKalman::Status::VALID);
      } else if (fix_ecef_std.norm() < VALID_POS_STD &&
time_since_reset > VALID_TIME_SINCE_RESET) {

fix.setStatus(cereal::LiveLocationKalman::Status::UNCALIBRATED
);
      } else {

fix.setStatus(cereal::LiveLocationKalman::Status::UNINITIALIZE
D);
      }
    }

    VectorXd Localizer::get_position_geodetic() {
      VectorXd fix_ecef = this->kf-
>get_x().segment<STATE_ECEF_POS_LEN>(STATE_ECEF_POS_START);
      ECEF fix_ecef_ecef = { .x = fix_ecef(0), .y =
fix_ecef(1), .z = fix_ecef(2) };
      Geodetic fix_pos_geo = ecef2geodetic(fix_ecef_ecef);
      return Vector3d(fix_pos_geo.lat, fix_pos_geo.lon,
fix_pos_geo.alt);
    }
```

```
    VectorXd Localizer::get_state() {
      return this->kf->get_x();
    }

    VectorXd Localizer::get_stdev() {
      return this->kf->get_P().diagonal().array().sqrt();
    }

    void Localizer::handle_sensors(double current_time, const
capnp::List<cereal::SensorEventData,
capnp::Kind::STRUCT>::Reader& log) {
      // TODO does not yet account for double sensor readings
in the log
      for (int i = 0; i < log.size(); i++) {
        const cereal::SensorEventData::Reader& sensor_reading
= log[i];

        // Ignore empty readings (e.g. in case the
magnetometer had no data ready)
        if (sensor_reading.getTimestamp() == 0) {
          continue;
        }

        double sensor_time = 1e-9 *
sensor_reading.getTimestamp();

        // sensor time and log time should be close
        if (std::abs(current_time - sensor_time) > 0.1) {
          LOGE("Sensor reading ignored, sensor timestamp more
than 100ms off from log time");
          return;
        }

          // TODO: handle messages from two IMUs at the same
time
```

```
        if (sensor_reading.getSource() ==
cereal::SensorEventData::SensorSource::BMX055) {
          continue;
        }

        // Gyro Uncalibrated
        if (sensor_reading.getSensor() ==
SENSOR_GYRO_UNCALIBRATED && sensor_reading.getType() ==
SENSOR_TYPE_GYROSCOPE_UNCALIBRATED) {
          auto v =
sensor_reading.getGyroUncalibrated().getV();
          auto meas = Vector3d(-v[2], -v[1], -v[0]);
          if (meas.norm() < ROTATION_SANITY_CHECK) {
            this->kf->predict_and_observe(sensor_time,
OBSERVATION_PHONE_GYRO, { meas });
          }
        }

        // Accelerometer
        if (sensor_reading.getSensor() == SENSOR_ACCELEROMETER
&& sensor_reading.getType() == SENSOR_TYPE_ACCELEROMETER) {
          auto v = sensor_reading.getAcceleration().getV();

          // TODO: reduce false positives and re-enable this
check
          // check if device fell, estimate 10 for g
          // 40m/s**2 is a good filter for falling detection,
no false positives in 20k minutes of driving
          //this->device_fell |= (floatlist2vector(v) -
Vector3d(10.0, 0.0, 0.0)).norm() > 40.0;

          auto meas = Vector3d(-v[2], -v[1], -v[0]);
          if (meas.norm() < ACCEL_SANITY_CHECK) {
            this->kf->predict_and_observe(sensor_time,
OBSERVATION_PHONE_ACCEL, { meas });
```

```
            }
          }
        }
      }

      void Localizer::input_fake_gps_observations(double
current_time) {
        // This is done to make sure that the error estimate of
the position does not blow up
        // when the filter is in no-gps mode
        // Steps : first predict -> observe current obs with
reasonable STD
        this->kf->predict(current_time);

        VectorXd current_x = this->kf->get_x();
        VectorXd ecef_pos =
current_x.segment<STATE_ECEF_POS_LEN>(STATE_ECEF_POS_START);
        VectorXd ecef_vel =
current_x.segment<STATE_ECEF_VELOCITY_LEN>(STATE_ECEF_VELOCITY
_START);
        MatrixXdr ecef_pos_R = this->kf->get_fake_gps_pos_cov();
        MatrixXdr ecef_vel_R = this->kf->get_fake_gps_vel_cov();

        this->kf->predict_and_observe(current_time,
OBSERVATION_ECEF_POS, { ecef_pos }, { ecef_pos_R });
        this->kf->predict_and_observe(current_time,
OBSERVATION_ECEF_VEL, { ecef_vel }, { ecef_vel_R });
      }

      void Localizer::handle_gps(double current_time, const
cereal::GpsLocationData::Reader& log) {
        // ignore the message if the fix is invalid
        bool gps_invalid_flag = (log.getFlags() % 2 == 0);
        bool gps_unreasonable = (Vector2d(log.getAccuracy(),
log.getVerticalAccuracy()).norm() >= SANE_GPS_UNCERTAINTY);
```

```
      bool gps_accuracy_insane = ((log.getVerticalAccuracy()
<= 0) || (log.getSpeedAccuracy() <= 0) ||
(log.getBearingAccuracyDeg() <= 0));
      bool gps_lat_lng_alt_insane =
((std::abs(log.getLatitude()) > 90) ||
(std::abs(log.getLongitude()) > 180) ||
(std::abs(log.getAltitude()) > ALTITUDE_SANITY_CHECK));
      bool gps_vel_insane =
(floatlist2vector(log.getVNED()).norm() > TRANS_SANITY_CHECK);

      if (gps_invalid_flag || gps_unreasonable ||
gps_accuracy_insane || gps_lat_lng_alt_insane ||
gps_vel_insane){
        this->determine_gps_mode(current_time);
        return;
      }

      // Process message
      this->last_gps_fix = current_time;
      this->gps_mode = true;
      Geodetic geodetic = { log.getLatitude(),
log.getLongitude(), log.getAltitude() };
      this->converter =
std::make_unique<LocalCoord>(geodetic);

      VectorXd ecef_pos = this->converter->ned2ecef({ 0.0,
0.0, 0.0 }).to_vector();
      VectorXd ecef_vel = this->converter->ned2ecef({
log.getVNED()[0], log.getVNED()[1], log.getVNED()[2]
}).to_vector() - ecef_pos;
      MatrixXdr ecef_pos_R = Vector3d::Constant(std::pow(10.0
* log.getAccuracy(),2) + std::pow(10.0 *
log.getVerticalAccuracy(),2)).asDiagonal();
```

```
      MatrixXdr ecef_vel_R =
Vector3d::Constant(std::pow(log.getSpeedAccuracy() * 10.0,
2)).asDiagonal();

      this->unix_timestamp_millis = log.getTimestamp();
      double gps_est_error = (this->kf-
>get_x().segment<STATE_ECEF_POS_LEN>(STATE_ECEF_POS_START) -
ecef_pos).norm();

      VectorXd orientation_ecef = quat2euler(vector2quat(this-
>kf-
>get_x().segment<STATE_ECEF_ORIENTATION_LEN>(STATE_ECEF_ORIENT
ATION_START)));
      VectorXd orientation_ned = ned_euler_from_ecef({
ecef_pos(0), ecef_pos(1), ecef_pos(2) }, orientation_ecef);
      VectorXd orientation_ned_gps = Vector3d(0.0, 0.0,
DEG2RAD(log.getBearingDeg()));
      VectorXd orientation_error = (orientation_ned -
orientation_ned_gps).array() - M_PI;
      for (int i = 0; i < orientation_error.size(); i++) {
        orientation_error(i) = std::fmod(orientation_error(i),
2.0 * M_PI);
        if (orientation_error(i) < 0.0) {
          orientation_error(i) += 2.0 * M_PI;
        }
        orientation_error(i) -= M_PI;
      }
      VectorXd initial_pose_ecef_quat =
quat2vector(euler2quat(ecef_euler_from_ned({ ecef_pos(0),
ecef_pos(1), ecef_pos(2) }, orientation_ned_gps)));

      if (ecef_vel.norm() > 5.0 && orientation_error.norm() >
1.0) {
        LOGE("Locationd vs ubloxLocation orientation
difference too large, kalman reset");
```

```
        this->reset_kalman(NAN, initial_pose_ecef_quat,
ecef_pos, ecef_vel, ecef_pos_R, ecef_vel_R);
        this->kf->predict_and_observe(current_time,
OBSERVATION_ECEF_ORIENTATION_FROM_GPS, {
initial_pose_ecef_quat });
      } else if (gps_est_error > 100.0) {
        LOGE("Locationd vs ubloxLocation position difference
too large, kalman reset");
        this->reset_kalman(NAN, initial_pose_ecef_quat,
ecef_pos, ecef_vel, ecef_pos_R, ecef_vel_R);
      }

      this->kf->predict_and_observe(current_time,
OBSERVATION_ECEF_POS, { ecef_pos }, { ecef_pos_R });
      this->kf->predict_and_observe(current_time,
OBSERVATION_ECEF_VEL, { ecef_vel }, { ecef_vel_R });
    }

    void Localizer::handle_car_state(double current_time,
const cereal::CarState::Reader& log) {
      this->car_speed = std::abs(log.getVEgo());
      if (log.getStandstill()) {
        this->kf->predict_and_observe(current_time,
OBSERVATION_NO_ROT, { Vector3d(0.0, 0.0, 0.0) });
        this->kf->predict_and_observe(current_time,
OBSERVATION_NO_ACCEL, { Vector3d(0.0, 0.0, 0.0) });
      }
    }

    void Localizer::handle_cam_odo(double current_time, const
cereal::CameraOdometry::Reader& log) {
      VectorXd rot_device = this->device_from_calib *
floatlist2vector(log.getRot());
      VectorXd trans_device = this->device_from_calib *
floatlist2vector(log.getTrans());
```

```
      if ((rot_device.norm() > ROTATION_SANITY_CHECK) ||
(trans_device.norm() > TRANS_SANITY_CHECK)) {
        return;
      }

      VectorXd rot_calib_std =
floatlist2vector(log.getRotStd());
      VectorXd trans_calib_std =
floatlist2vector(log.getTransStd());

      if ((rot_calib_std.minCoeff() <= MIN_STD_SANITY_CHECK)
|| (trans_calib_std.minCoeff() <= MIN_STD_SANITY_CHECK)) {
        return;
      }

      if ((rot_calib_std.norm() > 10 * ROTATION_SANITY_CHECK)
|| (trans_calib_std.norm() > 10 * TRANS_SANITY_CHECK)) {
        return;
      }

      this->posenet_stds.pop_front();
      this->posenet_stds.push_back(trans_calib_std[0]);

      // Multiply by 10 to avoid to high certainty in kalman
filter because of temporally correlated noise
      trans_calib_std *= 10.0;
      rot_calib_std *= 10.0;
      MatrixXdr rot_device_cov = rotate_std(this-
>device_from_calib,
rot_calib_std).array().square().matrix().asDiagonal();
      MatrixXdr trans_device_cov = rotate_std(this-
>device_from_calib,
trans_calib_std).array().square().matrix().asDiagonal();
```

```
      this->kf->predict_and_observe(current_time,
OBSERVATION_CAMERA_ODO_ROTATION,
        { rot_device }, { rot_device_cov });
      this->kf->predict_and_observe(current_time,
OBSERVATION_CAMERA_ODO_TRANSLATION,
        { trans_device }, { trans_device_cov });
    }

    void Localizer::handle_live_calib(double current_time,
const cereal::LiveCalibrationData::Reader& log) {
      if (log.getRpyCalib().size() > 0) {
        auto live_calib = floatlist2vector(log.getRpyCalib());
        if ((live_calib.minCoeff() < -CALIB_RPY_SANITY_CHECK)
|| (live_calib.maxCoeff() > CALIB_RPY_SANITY_CHECK)) {
          return;
        }

        this->calib = live_calib;
        this->device_from_calib = euler2rot(this->calib);
        this->calib_from_device = this-
>device_from_calib.transpose();
        this->calibrated = log.getCalStatus() == 1;
      }
    }

    void Localizer::reset_kalman(double current_time) {
      VectorXd init_x = this->kf->get_initial_x();
      MatrixXdr init_P = this->kf->get_initial_P();
      this->reset_kalman(current_time, init_x, init_P);
    }

    void Localizer::finite_check(double current_time) {
      bool all_finite = this->kf-
>get_x().array().isFinite().all() or this->kf-
>get_P().array().isFinite().all();
```

```
    if (!all_finite) {
      LOGE("Non-finite values detected, kalman reset");
      this->reset_kalman(current_time);
    }
  }

  void Localizer::time_check(double current_time) {
    if (std::isnan(this->last_reset_time)) {
      this->last_reset_time = current_time;
    }
    double filter_time = this->kf->get_filter_time();
    bool big_time_gap = !std::isnan(filter_time) &&
(current_time - filter_time > 10);
    if (big_time_gap) {
      LOGE("Time gap of over 10s detected, kalman reset");
      this->reset_kalman(current_time);
    }
  }

  void Localizer::update_reset_tracker() {
    // reset tracker is tuned to trigger when over
1reset/10s over 2min period
    if (this->isGpsOK()) {
      this->reset_tracker *= .99995;
    } else {
      this->reset_tracker = 0.0;
    }
  }

  void Localizer::reset_kalman(double current_time, VectorXd
init_orient, VectorXd init_pos, VectorXd init_vel, MatrixXdr
init_pos_R, MatrixXdr init_vel_R) {
    // too nonlinear to init on completely wrong
    VectorXd current_x = this->kf->get_x();
    MatrixXdr current_P = this->kf->get_P();
```

```
        MatrixXdr init_P = this->kf->get_initial_P();
        MatrixXdr reset_orientation_P = this->kf-
>get_reset_orientation_P();
        int non_ecef_state_err_len = init_P.rows() -
(STATE_ECEF_POS_ERR_LEN + STATE_ECEF_ORIENTATION_ERR_LEN +
STATE_ECEF_VELOCITY_ERR_LEN);


current_x.segment<STATE_ECEF_ORIENTATION_LEN>(STATE_ECEF_ORIEN
TATION_START) = init_orient;

current_x.segment<STATE_ECEF_VELOCITY_LEN>(STATE_ECEF_VELOCITY
_START) = init_vel;

current_x.segment<STATE_ECEF_POS_LEN>(STATE_ECEF_POS_START) =
init_pos;

        init_P.block<STATE_ECEF_POS_ERR_LEN,
STATE_ECEF_POS_ERR_LEN>(STATE_ECEF_POS_ERR_START,
STATE_ECEF_POS_ERR_START).diagonal() = init_pos_R.diagonal();
        init_P.block<STATE_ECEF_ORIENTATION_ERR_LEN,
STATE_ECEF_ORIENTATION_ERR_LEN>(STATE_ECEF_ORIENTATION_ERR_STA
RT, STATE_ECEF_ORIENTATION_ERR_START).diagonal() =
reset_orientation_P.diagonal();
        init_P.block<STATE_ECEF_VELOCITY_ERR_LEN,
STATE_ECEF_VELOCITY_ERR_LEN>(STATE_ECEF_VELOCITY_ERR_START,
STATE_ECEF_VELOCITY_ERR_START).diagonal() =
init_vel_R.diagonal();
        init_P.block(STATE_ANGULAR_VELOCITY_ERR_START,
STATE_ANGULAR_VELOCITY_ERR_START, non_ecef_state_err_len,
non_ecef_state_err_len).diagonal() =
current_P.block(STATE_ANGULAR_VELOCITY_ERR_START,
STATE_ANGULAR_VELOCITY_ERR_START, non_ecef_state_err_len,
non_ecef_state_err_len).diagonal();
```

```
    this->reset_kalman(current_time, current_x, init_P);
  }

  void Localizer::reset_kalman(double current_time, VectorXd
init_x, MatrixXdr init_P) {
    this->kf->init_state(init_x, init_P, current_time);
    this->last_reset_time = current_time;
    this->reset_tracker += 1.0;
  }

  void Localizer::handle_msg_bytes(const char *data, const
size_t size) {
    AlignedBuffer aligned_buf;

    capnp::FlatArrayMessageReader
cmsg(aligned_buf.align(data, size));
    cereal::Event::Reader event =
cmsg.getRoot<cereal::Event>();

    this->handle_msg(event);
  }

  void Localizer::handle_msg(const cereal::Event::Reader&
log) {
    double t = log.getLogMonoTime() * 1e-9;
    this->time_check(t);
    if (log.isSensorEvents()) {
      this->handle_sensors(t, log.getSensorEvents());
    } else if (log.isGpsLocationExternal()) {
      this->handle_gps(t, log.getGpsLocationExternal());
    } else if (log.isCarState()) {
      this->handle_car_state(t, log.getCarState());
    } else if (log.isCameraOdometry()) {
      this->handle_cam_odo(t, log.getCameraOdometry());
    } else if (log.isLiveCalibration()) {
```

```
          this->handle_live_calib(t, log.getLiveCalibration());
        }
        this->finite_check();
        this->update_reset_tracker();
      }

      kj::ArrayPtr<capnp::byte>
Localizer::get_message_bytes(MessageBuilder& msg_builder, bool
inputsOK,

bool sensorsOK, bool gpsOK, bool msgValid) {
        cereal::Event::Builder evt = msg_builder.initEvent();
        evt.setValid(msgValid);
        cereal::LiveLocationKalman::Builder liveLoc =
evt.initLiveLocationKalman();
        this->build_live_location(liveLoc);
        liveLoc.setSensorsOK(sensorsOK);
        liveLoc.setGpsOK(gpsOK);
        liveLoc.setInputsOK(inputsOK);
        return msg_builder.toBytes();
      }


      bool Localizer::isGpsOK() {
        return this->kf->get_filter_time() - this->last_gps_fix
< 1.0;
      }

      void Localizer::determine_gps_mode(double current_time) {
        // 1. If the pos_std is greater than what's not
acceptible and localizer is in gps-mode, reset to no-gps-mode
        // 2. If the pos_std is greater than what's not
acceptible and localizer is in no-gps-mode, fake obs
        // 3. If the pos_std is smaller than what's not
acceptible, let gps-mode be whatever it is
```

```
      VectorXd current_pos_std = this->kf-
>get_P().block<STATE_ECEF_POS_ERR_LEN,
STATE_ECEF_POS_ERR_LEN>(STATE_ECEF_POS_ERR_START,
STATE_ECEF_POS_ERR_START).diagonal().array().sqrt();
      if (current_pos_std.norm() > SANE_GPS_UNCERTAINTY){
        if (this->gps_mode){
          this->gps_mode = false;
          this->reset_kalman(current_time);
        }
        else{
          this->input_fake_gps_observations(current_time);
        }
      }
    }

    int Localizer::locationd_thread() {
      const std::initializer_list<const char *> service_list =
{"gpsLocationExternal", "sensorEvents", "cameraOdometry",
"liveCalibration", "carState", "carParams"};
      PubMaster pm({"liveLocationKalman"});

      // TODO: remove carParams once we're always sending at
100Hz
      SubMaster sm(service_list, nullptr,
{"gpsLocationExternal", "carParams"});

      uint64_t cnt = 0;
      bool filterInitialized = false;

      while (!do_exit) {
        sm.update();
        if (filterInitialized){
          for (const char* service : service_list) {
            if (sm.updated(service) && sm.valid(service)){
              const cereal::Event::Reader log = sm[service];
```

```
            this->handle_msg(log);
          }
        }
      } else {
        filterInitialized = sm.allAliveAndValid();
      }

      // 100Hz publish for notcars, 20Hz for cars
      const char* trigger_msg =
sm["carParams"].getCarParams().getNotCar() ? "sensorEvents" :
"cameraOdometry";
      if (sm.updated(trigger_msg)) {
        bool inputsOK = sm.allAliveAndValid();
        bool sensorsOK = sm.alive("sensorEvents") &&
sm.valid("sensorEvents");
        bool gpsOK = this->isGpsOK();

        MessageBuilder msg_builder;
        kj::ArrayPtr<capnp::byte> bytes = this-
>get_message_bytes(msg_builder, inputsOK, sensorsOK, gpsOK,
filterInitialized);
        pm.send("liveLocationKalman", bytes.begin(),
bytes.size());

        if (cnt % 1200 == 0 && gpsOK) {  // once a minute
          VectorXd posGeo = this->get_position_geodetic();
          std::string lastGPSPosJSON = util::string_format(
            "{\"latitude\": %.15f, \"longitude\": %.15f,
\"altitude\": %.15f}", posGeo(0), posGeo(1), posGeo(2));

          std::thread([] (const std::string gpsjson) {
            Params().put("LastGPSPosition", gpsjson);
          }, lastGPSPosJSON).detach();
        }
        cnt++;
```

```
    }
  }
  return 0;
}

int main() {
  util::set_realtime_priority(5);

  Localizer localizer;
  return localizer.locationd_thread();
}
#ifndef _GNU_SOURCE
#define _GNU_SOURCE
#endif

#include "selfdrive/common/swaglog.h"

#include <cassert>
#include <cstring>
#include <limits>
#include <mutex>
#include <string>

#include <zmq.h>
#include "json11.hpp"

#include "selfdrive/common/util.h"
#include "selfdrive/common/version.h"
#include "selfdrive/hardware/hw.h"

class SwaglogState : public LogState {
 public:
  SwaglogState() : LogState("ipc:///tmp/logmessage") {}

  json11::Json::object ctx_j;
```

```
  inline void initialize() {
    ctx_j = json11::Json::object {};
    print_level = CLOUDLOG_WARNING;
    const char* print_lvl = getenv("LOGPRINT");
    if (print_lvl) {
      if (strcmp(print_lvl, "debug") == 0) {
        print_level = CLOUDLOG_DEBUG;
      } else if (strcmp(print_lvl, "info") == 0) {
        print_level = CLOUDLOG_INFO;
      } else if (strcmp(print_lvl, "warning") == 0) {
        print_level = CLOUDLOG_WARNING;
      }
    }

    char* dongle_id = getenv("DONGLE_ID");
    if (dongle_id) {
      ctx_j["dongle_id"] = dongle_id;
    }
    char* daemon_name = getenv("MANAGER_DAEMON");
    if (daemon_name) {
      ctx_j["daemon"] = daemon_name;
    }
    ctx_j["version"] = COMMA_VERSION;
    ctx_j["dirty"] = !getenv("CLEAN");

    // device type
    if (Hardware::TICI()) {
      ctx_j["device"] =  "tici";
    } else {
      ctx_j["device"] =  "pc";
    }
    LogState::initialize();
  }
};
```

```
    static SwaglogState s = {};
    bool LOG_TIMESTAMPS = getenv("LOG_TIMESTAMPS");
    uint32_t NO_FRAME_ID =
std::numeric_limits<uint32_t>::max();

    static void log(int levelnum, const char* filename, int
lineno, const char* func, const char* msg, const std::string&
log_s) {
      if (levelnum >= s.print_level) {
        printf("%s: %s\n", filename, msg);
      }
      char levelnum_c = levelnum;
      zmq_send(s.sock, (levelnum_c + log_s).c_str(),
log_s.length() + 1, ZMQ_NOBLOCK);
    }
    static void cloudlog_common(int levelnum, const char*
filename, int lineno, const char* func,
                                char* msg_buf,
json11::Json::object msg_j={}) {
      std::lock_guard lk(s.lock);
      if (!s.initialized) s.initialize();

      json11::Json::object log_j = json11::Json::object {
        {"ctx", s.ctx_j},
        {"levelnum", levelnum},
        {"filename", filename},
        {"lineno", lineno},
        {"funcname", func},
        {"created", seconds_since_epoch()}
      };
      if (msg_j.empty()) {
        log_j["msg"] = msg_buf;
      } else {
        log_j["msg"] = msg_j;
```

```
    }

    std::string log_s = ((json11::Json)log_j).dump();
    log(levelnum, filename, lineno, func, msg_buf, log_s);
    free(msg_buf);
  }

  void cloudlog_e(int levelnum, const char* filename, int
lineno, const char* func,
                  const char* fmt, ...) {
    va_list args;
    va_start(args, fmt);
    char* msg_buf = nullptr;
    int ret = vasprintf(&msg_buf, fmt, args);
    va_end(args);
    if (ret <= 0 || !msg_buf) return;
    cloudlog_common(levelnum, filename, lineno, func,
msg_buf);
  }

  void cloudlog_t_common(int levelnum, const char* filename,
int lineno, const char* func,
                         uint32_t frame_id, const char* fmt,
va_list args) {
    if (!LOG_TIMESTAMPS) return;
    char* msg_buf = nullptr;
    int ret = vasprintf(&msg_buf, fmt, args);
    if (ret <= 0 || !msg_buf) return;
    json11::Json::object tspt_j = json11::Json::object{
      {"event", msg_buf},
      {"time", std::to_string(nanos_since_boot())}
    };
    if (frame_id < NO_FRAME_ID) {
      tspt_j["frame_id"] = std::to_string(frame_id);
    }
```

```
        tspt_j = json11::Json::object{{"timestamp", tspt_j}};
        cloudlog_common(levelnum, filename, lineno, func,
msg_buf, tspt_j);
    }


    void cloudlog_te(int levelnum, const char* filename, int
lineno, const char* func,
                     const char* fmt, ...) {
        va_list args;
        va_start(args, fmt);
        cloudlog_t_common(levelnum, filename, lineno, func,
NO_FRAME_ID, fmt, args);
        va_end(args);
    }
    void cloudlog_te(int levelnum, const char* filename, int
lineno, const char* func,
                     uint32_t frame_id, const char* fmt, ...)
{
        va_list args;
        va_start(args, fmt);
        cloudlog_t_common(levelnum, filename, lineno, func,
frame_id, fmt, args);
        va_end(args);
    }
```

ДОДАТОК Б
КОД КОНТРОЛЮ АВТОМОБІЛЯ

```
import pickle
import random
import socket
import logging
import socketserver
import numpy as np

from utils import *

# compatible with Windows
socket.SO_REUSEPORT = socket.SO_REUSEADDR


class
SignatureRequestHandler(socketserver.BaseRequestHandler):
    user_num = 0
    ka_pub_keys_map = {}    # {id: {c_pk: bytes, s_pk,
bytes, signature: bytes}}
    U_1 = []

    def handle(self) -> None:
        # receive data from the client
        data = SocketUtil.recv_msg(self.request)

        msg = pickle.loads(data)
        id = msg["id"]
        del msg["id"]

        self.ka_pub_keys_map[id] = msg
        self.U_1.append(id)

        received_num = len(self.U_1)
```

```
            logging.info("[%d/%d] | received user %s's
signature", received_num, self.user_num, id)


    class
SecretShareRequestHandler(socketserver.BaseRequestHandler):
        U_1_num = 0
        ciphertexts_map = {}         # {u:{v1: ciphertexts,
v2: ciphertexts}}
        U_2 = []

        def handle(self) -> None:
            # receive data from the client
            data = SocketUtil.recv_msg(self.request)

            msg = pickle.loads(data)
            id = msg[0]

            # retrieve each user's ciphertexts
            for key, value in msg[1].items():
                if key not in self.ciphertexts_map:
                    self.ciphertexts_map[key] = {}
                self.ciphertexts_map[key][id] = value

            self.U_2.append(id)

            received_num = len(self.U_2)

            logging.info("[%d/%d] | received user %s's
ciphertexts", received_num, self.U_1_num, id)


    class
MaskingRequestHandler(socketserver.BaseRequestHandler):
```

```
        U_2_num = 0
        masked_gradients_list = []
        U_3 = []

        def handle(self) -> None:
            # receive data from the client
            data = SocketUtil.recv_msg(self.request)

            msg = pickle.loads(data)
            id = msg[0]

            self.U_3.append(msg[0])
            self.masked_gradients_list.append(msg[1])

            received_num = len(self.U_3)

            logging.info("[%d/%d] | received user %s's masked
gradients", received_num, self.U_2_num, id)


    class
ConsistencyRequestHandler(socketserver.BaseRequestHandler):
        U_3_num = 0
        consistency_check_map = {}
        U_4 = []

        def handle(self) -> None:
            data = SocketUtil.recv_msg(self.request)

            msg = pickle.loads(data)
            id = msg[0]

            self.U_4.append(id)
            self.consistency_check_map[id] = msg[1]
```

```
            received_num = len(self.U_4)

            logging.info("[%d/%d] | received user %s's
consistency check", received_num, self.U_3_num, id)


    class
UnmaskingRequestHandler(socketserver.BaseRequestHandler):
        U_4_num = 0
        priv_key_shares_map = {}        # {id: []}
        random_seed_shares_map = {}     # {id: []}
        U_5 = []

        def handle(self) -> None:
            data = SocketUtil.recv_msg(self.request)

            msg = pickle.loads(data)
            id = msg[0]

            # retrieve the private key shares
            for key, value in msg[1].items():
                if key not in self.priv_key_shares_map:
                    self.priv_key_shares_map[key] = []
                self.priv_key_shares_map[key].append(value)

            # retrieve the ramdom seed shares
            for key, value in msg[2].items():
                if key not in self.random_seed_shares_map:
                    self.random_seed_shares_map[key] = []
                self.random_seed_shares_map[key].append(value)

            self.U_5.append(id)

            received_num = len(self.U_5)
```

```
            logging.info("[%d/%d] | received user %s's
shares", received_num, self.U_4_num, id)


    class Server:
        def __init__(self):
            self.id = "0"
            self.host = socket.gethostname()
            self.broadcast_port = 10000
            self.signature_port = 20000
            self.ss_port = 20001
            self.masking_port = 20002
            self.consistency_port = 20003
            self.unmasking_port = 20004


socketserver.ThreadingTCPServer.allow_reuse_address = True

            self.signature_server =
socketserver.ThreadingTCPServer(
                (self.host, self.signature_port),
SignatureRequestHandler)
            self.ss_server = socketserver.ThreadingTCPServer(
                (self.host, self.ss_port),
SecretShareRequestHandler)
            self.masking_server =
socketserver.ThreadingTCPServer(
                (self.host, self.masking_port),
MaskingRequestHandler)
            self.consistency_server =
socketserver.ThreadingTCPServer(
                (self.host, self.consistency_port),
ConsistencyRequestHandler)
            self.unmasking_server =
socketserver.ThreadingTCPServer(
```

```
                (self.host, self.unmasking_port),
UnmaskingRequestHandler)

        def broadcast_signatures(self, port: int):
            """Broadcasts all users' key pairs and
corresponding signatures.

            Args:
                port (int): the port used to broadcast the
message.
            """

            server = socket.socket(socket.AF_INET,
socket.SOCK_DGRAM)

            # reuse port so we will be able to run multiple
clients on single (host, port).
            server.setsockopt(socket.SOL_SOCKET,
socket.SO_REUSEPORT, 1)

            # enable broadcasting mode
            server.setsockopt(socket.SOL_SOCKET,
socket.SO_BROADCAST, 1)

            data =
pickle.dumps(SignatureRequestHandler.ka_pub_keys_map)

            SocketUtil.broadcast_msg(server, data, port)

            logging.info("broadcasted all signatures.")

            server.close()

        def send(self, msg: bytes, host: str, port: int):
            """Sends message to host:port.
```

```
            Args:
                msg (bytes): the message to be sent.
                host (str): the target host.
                port (int): the target port.
            """

            sock = socket.socket()
            sock.setsockopt(socket.SOL_SOCKET,
socket.SO_REUSEPORT, 1)
            sock.connect((host, port))

            SocketUtil.send_msg(sock, msg)

            sock.close()

        def unmask(self, shape: tuple) -> np.ndarray:
            """Unmasks gradients by reconstructing random
vectors and private mask vectors.

            Args:
                shape (tuple): the shape of the raw gradients.

            Returns:
                np.ndarray: the sum of the raw gradients.
            """

            # reconstruct random vectors p_v_u
            recon_random_vec_list = []
            for u in SecretShareRequestHandler.U_2:
                if u not in MaskingRequestHandler.U_3:
                    # the user drops out, reconstruct its
private keys and then generate the corresponding random
vectors
```

```
                    priv_key =
SS.recon(UnmaskingRequestHandler.priv_key_shares_map[u])
                    for v in MaskingRequestHandler.U_3:
                        shared_key = KA.agree(priv_key,
SignatureRequestHandler.ka_pub_keys_map[v]["s_pk"])

                        random.seed(shared_key)
                        rs =
np.random.RandomState(random.randint(0, 2**32 - 1))

                        if int(u) > int(v):

recon_random_vec_list.append(rs.random(shape))
                        else:
                            recon_random_vec_list.append(-
rs.random(shape))

            # reconstruct private mask vectors p_u
            recon_priv_vec_list = []
            for u in MaskingRequestHandler.U_3:
                random_seed =
SS.recon(UnmaskingRequestHandler.random_seed_shares_map[u])
                rs = np.random.RandomState(random_seed)
                priv_mask_vec = rs.random(shape)

                recon_priv_vec_list.append(priv_mask_vec)

            masked_gradients =
np.sum(np.array(MaskingRequestHandler.masked_gradients_list),
axis=0)
            recon_priv_vec =
np.sum(np.array(recon_priv_vec_list), axis=0)
            recon_random_vec =
np.sum(np.array(recon_random_vec_list), axis=0)
```

```
            output = masked_gradients - recon_priv_vec +
recon_random_vec

            return output
```

ДОДАТОК В
ВІДОМІСТЬ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Позначення	Найменування	Дод. відомості
	Текстові документи	
1.	Пояснювальна записка	104 с.
	Інші документи	
2.	Презентаційні матеріали	16 плакатів

.	.	Прізвище та ініціали.	Підп.	Дата			
Розробив		Ларка І.В			Федеративне машинне навчання для рішень автопілоту	Шифр групи	Код напр./спец.
Перевірив		Терзіян В.Я.				СШІм-20-1	122
Н.контр.		Малєєва І.А.				ХНУРЕ кафедра ШІ	
Затв.		Філатов В.О.					