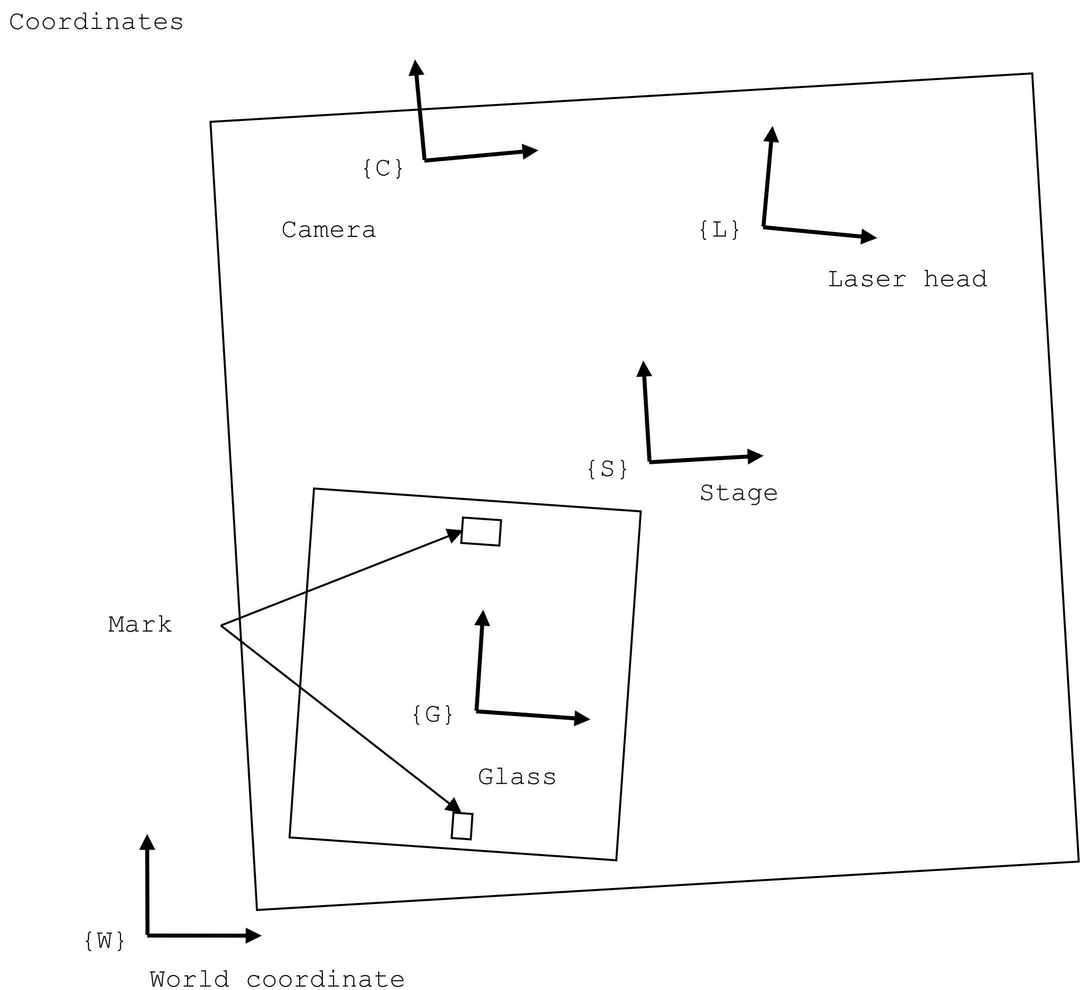
# Programming Study
# Align Algorithm

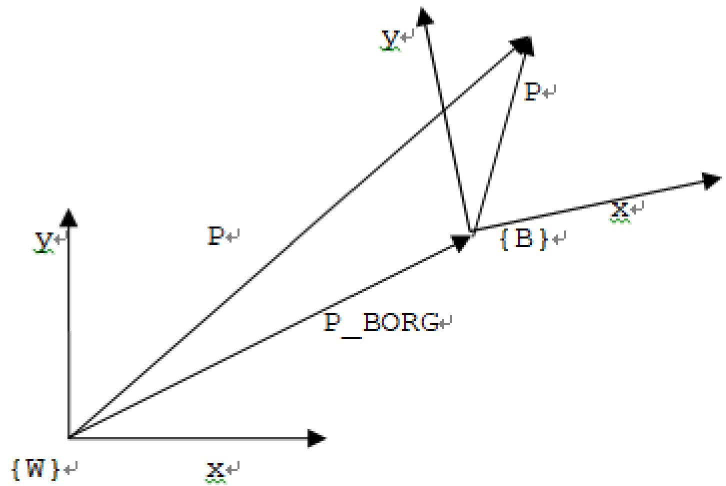
NAM SOFTWARE

2017

# Geometric model - Coordinates



# Coordinate Transform




$$^{W}P = {}^{B}_{W}R\,{}^{B}P = \begin{bmatrix} {}^{B}_{W}T & {}^{W}P_{BORG} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} {}^{B}P \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & P_{BORG\_x} \\ \sin\theta & \cos\theta & P_{BORG\_y} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} {}^{B}P_x \\ {}^{B}P_y \\ 1 \end{bmatrix}$$

# Geometric model – camera

Camera Assembly to World

Mark position w.r.t camera pixel

$$ {}^{W}M_i = {}^{CA}_{W}R \; {}^{C}_{CA}R \; {}^{P}_{C}R \; {}^{P}M_i $$

Mark position w.r.t World

Camera to Camera Assembly

Pixel to Camera

# Geometric model – stage

Servo to World

Stage to Servo

GlassCenter to GlassLoading

$$ {}^{W}M_{i} = {}^{S}_{W}R \; {}^{B}_{S}R \; {}^{GL}_{B}R \; {}^{GC}_{GL}R \; {}^{G}_{GC}R \; {}^{G}M_{i} $$

Mark position w.r.t World

GlassLoading to Stage

Glass to GlassCenter

Mark position w.r.t Glass

# Camera matrix

$$ {}^{P}P = {}^{C}_{P}R\ {}^{C}P = \begin{bmatrix} 1/r_x & 0 & W/2 \\ 0 & -1/r_y & H/2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} {}^{C}P_x \\ {}^{C}P_y \\ 1 \end{bmatrix} $$

Resolution
mm/pixel

Width, Height

# Calibration – finding camera offset

target

Mark at the stage

$$ {}^{CA}_{W}R \; {}^{C}_{CA}R \; {}^{P}_{C}R \; {}^{P}M_i = {}^{S}_{W}R \; {}^{B}_{S}R \; {}^{B}M_i $$

$$ {}^{C}_{CA}R \; {}^{P}_{C}R \; {}^{P}M_i = {}^{CA}_{W}R^{-1} \; {}^{S}_{W}R \; {}^{B}_{S}R \; {}^{B}M_i $$

$$ {}^{C}_{CA}R \, U_i = V_i $$

# Least Square Method

$$\begin{bmatrix} 1 & -\theta & x \\ \theta & 1 & y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} U_{i1} \\ U_{i2} \\ 1 \end{bmatrix} = \begin{bmatrix} V_{i1} \\ V_{i2} \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & -U_{i2} & U_{i1} \\ 0 & 1 & U_{i1} & U_{i2} \end{bmatrix} \begin{bmatrix} x \\ y \\ \theta \\ 1 \end{bmatrix} = \begin{bmatrix} V_{i1} \\ V_{i2} \end{bmatrix}$$

$$U \; X = V$$

$$X = (U^T U)^{-1} U^T V$$

# Calibration – glass

target

$$ {}^{S}_{W}R\ {}^{B}_{S}R\ {}^{GL}_{B}R\ {}^{GC}_{GL}R\ {}^{G}_{GC}R\ {}^{G}M_i = {}^{CA}_{W}R\ {}^{C}_{CA}R\ {}^{P}_{C}R\ {}^{P}M_i $$

$$ {}^{GC}_{GL}R\ {}^{G}_{GC}R\ {}^{G}M_i = ({}^{S}_{W}R\ {}^{B}_{S}R\ {}^{GL}_{B}R)^{-1}\ {}^{CA}_{W}R\ {}^{C}_{CA}R\ {}^{P}_{C}R\ {}^{P}M_i $$

$$ {}^{GC}_{GL}R\, U_i = V_i $$

# Servo control to mark

$$\vec{S} = {}^{CA}_{W}R \; {}^{C}_{CA}R \; {}^{C}M - {}^{B}_{S}R \; {}^{GL}_{B}R \; {}^{GC}_{GL}R \; {}^{G}_{GC}R \; {}^{G}M$$

$[0,0,1]^T$

# Calibration error

$$E_i = {}^{CA}_{W}R \; {}^{C}_{CA}R \; {}^{P}_{C}R \; {}^{P}M_i - {}^{S}_{W}R \; {}^{B}_{S}R \; {}^{B}M_i$$

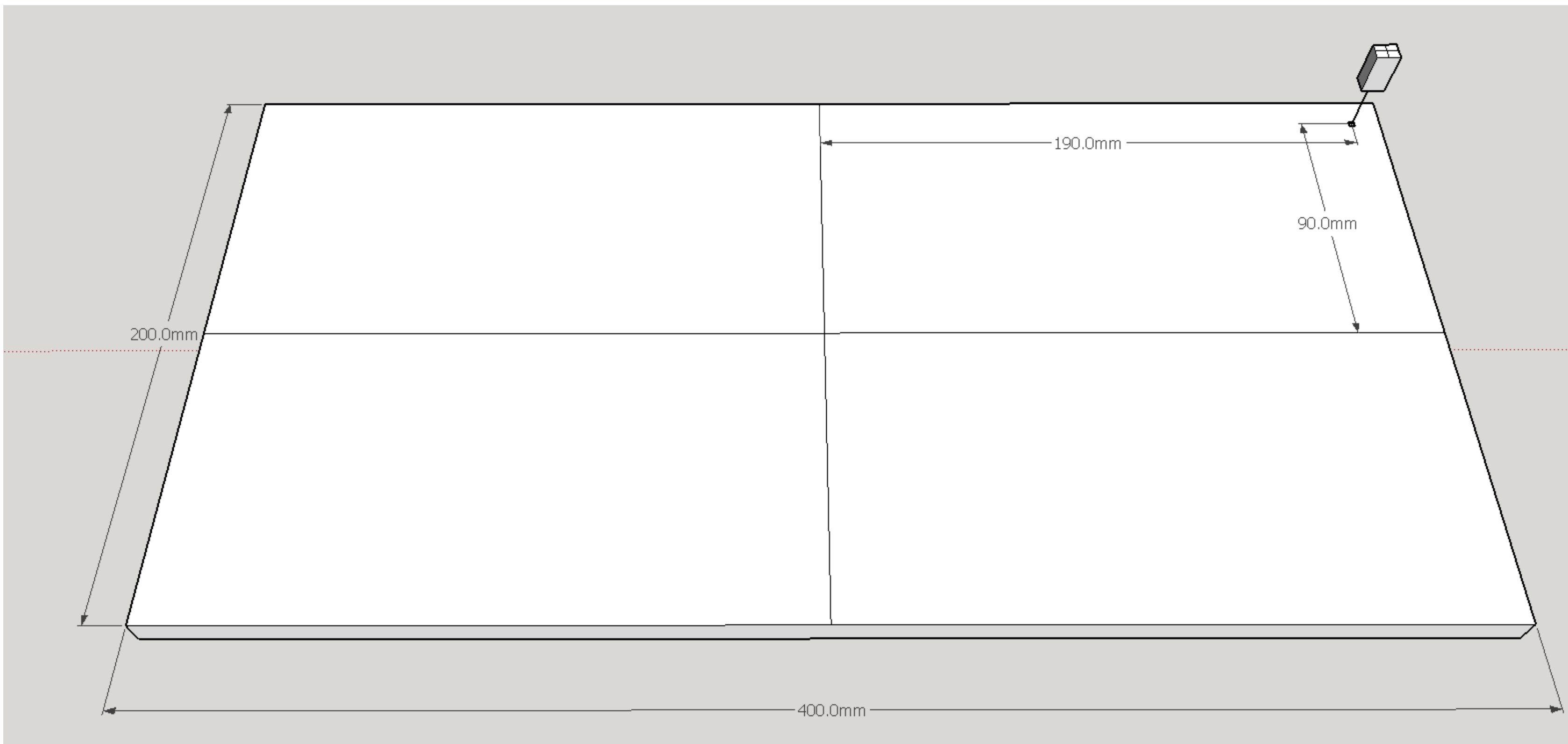
$$E_i = {}^{S}_{W}R^{L} \; {}^{B}_{S}R \; {}^{GL}_{B}R \; {}^{GC}_{GL}R \; {}^{G}_{GC}R \; {}^{G}M_i - {}^{CA}_{W}R \; {}^{C}_{CA}R \; {}^{P}_{C}R \; {}^{P}M_i$$

$$error = \frac{\sum_{i}^{N} |E_i|}{N}$$

# OpenCV matrix operation

```
Matx33d Orientation( double x, double y, double theta )
{
    return Matx33d(
        cos(theta), -sin(theta), x,
        sin(theta), cos(theta), y,
        0, 0, 1);
}


Matx33d CameraMatrix(double rx, double ry, double W, double H)
{
    return Matx33d(
        1 / rx, 0, W / 2,
        0, -1 / ry, H / 2,
        0, 0, 1);
}
```

# Camera, Stage and reference mark



# Calibrate Camera

```
Vec3d CalibrateCamera(const vector<Point2d>& M_p, const vector<Point2d>& S, double& error )
{
    Matx33d Rp2c = CameraMatrix(0.01, 0.01, 800, 600).inv(DECOMP_LU);
    Matx33d Rca2w = Orientation(190, 90, 0);
    Matx33d Rb2s = Orientation(0, 0, 0);
    Vec3d M_b(190, 90, 1);

    Mat V(2 * M_p.size(), 1, CV_64FC1);
    Mat U(2 * M_p.size(), 4, CV_64FC1);
    for (int i = 0; i < M_p.size(); ++i)
    {
        Matx33d Rs2w(Orientation(S[i].x, S[i].y, 0));
        Vec3d Vi = Rca2w.inv() * Rs2w * Rb2s * M_b;
        V.rowRange(i * 2, i * 2 + 2) = (Mat_<double>(2, 1) << Vi[0], Vi[1]) + 0;

        Vec3d Ui = Rp2c * Vec3d(M_p[i].x, M_p[i].y, 1);
        U.row(i * 2 + 0) = (Mat_<double>(1, 4) << 1, 0, -Ui[1], Ui[0]) + 0;
        U.row(i * 2 + 1) = (Mat_<double>(1, 4) << 0, 1, Ui[0], Ui[1]) + 0;
    }
    Mat X = (U.t() * U).inv(DECOMP_LU) * U.t() * V;
```

# Error check

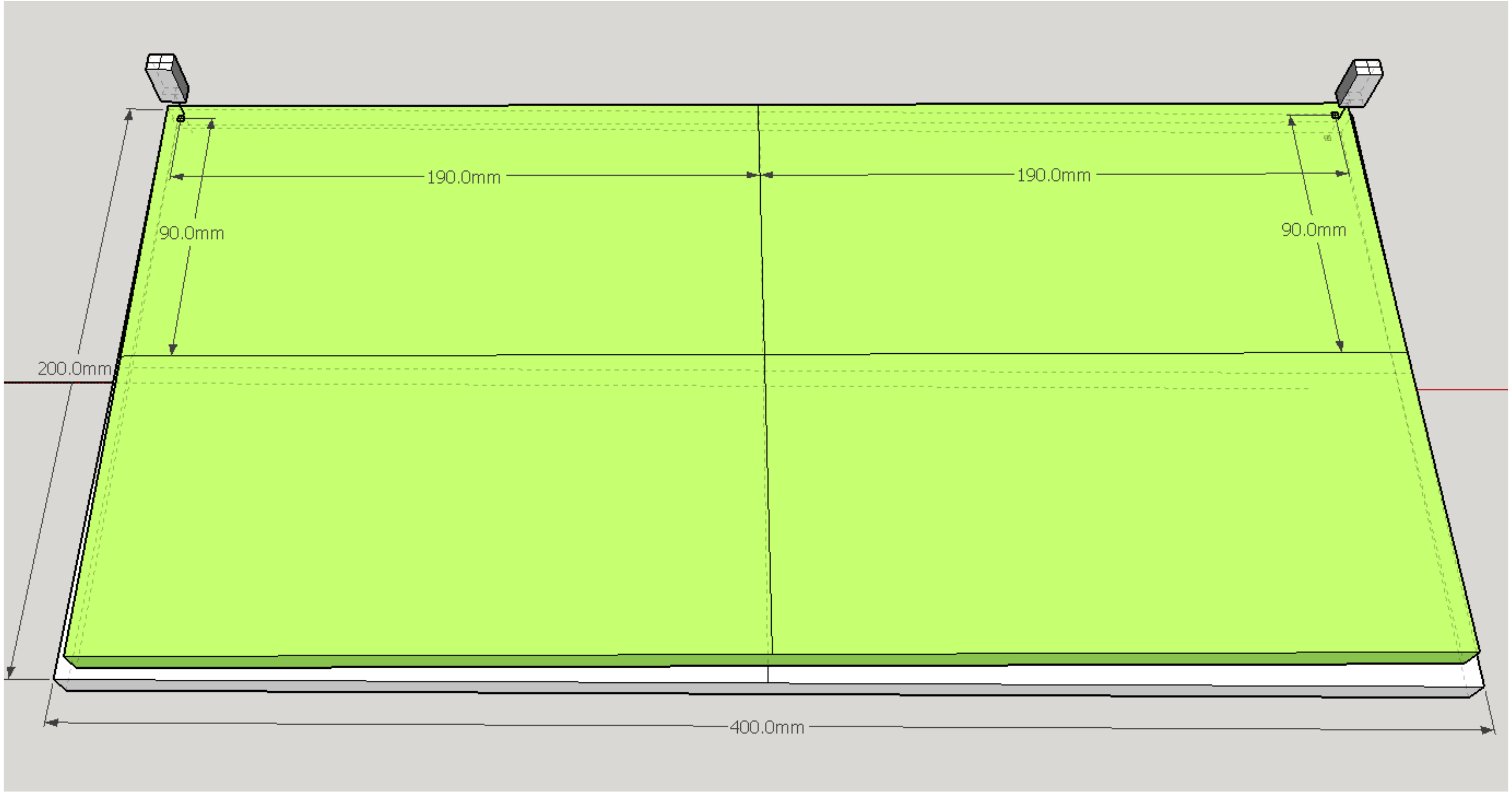
```
    // error
    Vec3d vX((const double*)X.data);
    Matx33d Rc2ca = Orientation(vX[0], vX[1], vX[2]);

    error = 0.0;
    for (int i = 0; i < M_p.size(); ++i)
    {
        Vec3d M_wc = Rca2w * Rc2ca * Rp2c * Vec3d(M_p[i].x, M_p[i].y, 1);

        Matx33d Rs2w(Orientation(S[i].x, S[i].y, 0));
        Vec3d M_wb = Rs2w * Rb2s * M_b;

        Vec3d ei = (M_wc - M_wb);
        error += sqrt( ei.dot( ei ));
    }
    error /= M_p.size();

    return vX;
}
```

# Cases

```
vector<Point2d> M_p = { Point2d(400 + 200, 300), Point2d(400 - 0, 300) };
vector<Point2d> S = { Point2d(1, 0), Point2d(-1, 0) };

double error = 0;
Vec3d vX = CalibrateCamera(M_p, S, error);

assertEqualDelta(-1.0, vX[0], 1e-6);
assertEqualDelta(0.0, vX[1], 1e-6);
assertEqualDelta(0.0, vX[2], 1e-6);
assertEqualDelta(0.0, error, 1e-6);
```

```
vector<Point2d> M_p = { Point2d(400 + 200, 300+100), Point2d(400 - 0, 300+100) };
vector<Point2d> S = { Point2d(1, 0), Point2d(-1, 0) };

double error = 0;
Vec3d vX = CalibrateCamera(M_p, S, error);

assertEqualDelta(-1.0, vX[0], 1e-6);
assertEqualDelta(1.0, vX[1], 1e-6);
assertEqualDelta(0.0, vX[2], 1e-6);
assertEqualDelta(0.0, error, 1e-6);
```

# Camera, Stage and Glass



# Calibrate Glass

```
Vec3d CalibrateGlass(const vector<Point2d>& M_p, const vector<Point2d>& M_g, double& error)
{
    Matx33d Rp2c = CameraMatrix(0.01, 0.01, 800, 600).inv(DECOMP_LU);
    Matx33d Rc2ca = Orientation(0, 0, 0);

    vector<Matx33d> Rca2w = { Orientation(190, 90, 0), Orientation(-190, 90, 0) };

    Matx33d Rb2s = Orientation(0, 0, 0);
    Matx33d Rs2w = Orientation(0, 0, 0);
    Matx33d Rgl2b = Orientation(0, 0, 0);
    Matx33d Rg2gc = Orientation(0, 0, 0);

    Mat V(2 * M_p.size(), 1, CV_64FC1);
    Mat U(2 * M_p.size(), 4, CV_64FC1);
    for (int i = 0; i < M_p.size(); ++i)
    {
        Vec3d Vi = (Rs2w * Rb2s * Rgl2b).inv(DECOMP_LU) * Rca2w[i] * Rc2ca * Rp2c * Vec3d(M_p[i].x, M_p[i].y, 1);
        V.rowRange(i * 2, i * 2 + 2) = (Mat_<double>(2, 1) << Vi[0], Vi[1]) + 0;

        Vec3d Ui = Rg2gc * Vec3d(M_g[i].x, M_g[i].y, 1);
        U.row(i * 2 + 0) = (Mat_<double>(1, 4) << 1, 0, -Ui[1], Ui[0]) + 0;
        U.row(i * 2 + 1) = (Mat_<double>(1, 4) << 0, 1, Ui[0], Ui[1]) + 0;
    }
    Mat X = (U.t() * U).inv(DECOMP_LU) * U.t() * V;
```

# Calibrate Glass

```
    // error
    Vec3d vX((const double*)X.data);
    Matx33d Rgc2gl = Orientation(vX[0], vX[1], vX[2]);
    error = 0.0;
    for (int i = 0; i < M_p.size(); ++i)
    {
        Vec3d M_wc = Rca2w[i] * Rc2ca * Rp2c * Vec3d(M_p[i].x, M_p[i].y, 1);
        Vec3d M_wb = Rs2w * Rb2s * Rgl2b * Rgc2gl * Rg2gc * Vec3d(M_g[i].x, M_g[i].y, 1);

        Vec3d ei = (M_wc - M_wb);
        error += sqrt(ei.dot(ei));
    }
    error /= M_p.size();
    return vX;
}
```

# Cases

```
vector<Point2d> M_p = { Point2d(400, 300), Point2d(400, 300) };
vector<Point2d> M_g = { Point2d(190, 90), Point2d(-190, 90) };

double error = 0;
Vec3d vX = CalibrateGlass(M_p, M_g, error);

assertEqualDelta(0.0, vX[0], 1e-6);
assertEqualDelta(0.0, vX[1], 1e-6);
assertEqualDelta(0.0, vX[2], 1e-6);
assertEqualDelta(0.0, error, 1e-6);
```

```
vector<Point2d> M_p = { Point2d(400+200, 300), Point2d(400+200, 300) };
vector<Point2d> M_g = { Point2d(190, 90), Point2d(-190, 90) };

double error = 0;
Vec3d vX = CalibrateGlass(M_p, M_g, error);

assertEqualDelta(2.0, vX[0], 1e-6);
assertEqualDelta(0.0, vX[1], 1e-6);
assertEqualDelta(0.0, vX[2], 1e-6);
assertEqualDelta(0.0, error, 1e-6);
```