Jacobian of Point Coordinates w.r.t. Parameters of General Calibrated Projective Camera

Karel Lebeda, Simon Hadfield, Richard Bowden

1 Introduction

This is a supplementary technical report for ACCV2014 paper: 2D Or Not 2D: Bridging the Gap Between Tracking and Structure from Motion [3]. It shows derivation of the Jacobian matrix of point x- and y-coordinates, with respect to parameters of a general, calibrated projective camera with 6 degrees of freedom (3 for position and 3 for rotation in the world coordinate frame). The Jacobian matrices are used in optimisation of the camera parameters, given a set of 3D points and their corresponding 2D projections. The optimisation using analytically estimated gradients generally converges faster, with lower computational costs than its numeric counterpart (using finite difference instead), while having a wider basin of convergence. We describe the derivation of the Jacobian in Section 3, and in Section 4 we provide the resulting C++ source code, which we believe will be of a practical value to the computer vision community. If you use this code in an academic work, please cite the paper [3].

2 Definition of Symbols and Relations

Hereby, we follow a general notation of [3]. Assuming a 3D point $\mathbf{X} = (x, y, z)^\top$ and a 2D point $\mathbf{u} = (u, v)^\top$, their respective homogeneous representations are $\widetilde{\mathbf{X}} = (\kappa x, \kappa y, \kappa z, \kappa)^\top$ and $\widetilde{\mathbf{u}} = (\lambda u, \lambda v, \lambda)^\top$ where κ and λ are unknown scalars. Without loss of generality we can assume that $\kappa = 1$. Then, the points are related by the projection equation of a general projective camera [2]:

$$\lambda \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} \widetilde{u}_1 \\ \widetilde{u}_2 \\ \widetilde{u}_3 \end{bmatrix} = \widetilde{\mathbf{u}} = \mathtt{P}\widetilde{\mathbf{X}} = \mathtt{P} \begin{bmatrix} \mathbf{X} \\ 1 \end{bmatrix} \ . \tag{1}$$

where $\mathtt{P}$ is the *projection matrix*:

$$\mathtt{P} = \frac{1}{f}\mathtt{K}\mathtt{R}\left[\mathtt{I}_{3\times 3}\middle| -\mathbf{C}\right] \ , \tag{2}$$

with f being the focal length (in world units, e.g. millimetres), $\mathtt{K}$ the matrix of intrinsic camera calibration parameters, $\mathtt{R}$ the rotation matrix relating the camera and world coordinate systems and $\mathbf{C}$ the position of the camera projection centre in the world coordinates. Assuming square pixels, we define $\mathtt{K}$ as:

$$\mathtt{K} = \begin{bmatrix} k_{11} & 0 & k_{13} \\ & k_{11} & k_{23} \\ & & 1 \end{bmatrix} , \tag{3}$$

where k_{11} is the focal length (in pixels, relating size of pixel to the world units and thus fixing the field of view of the camera) and $(k_{13}, k_{23})^\top$ are coordinates of the principal point. Both f and $\mathtt{K}$ are assumed known and fixed during the camera pose estimation.

While $\mathbf{C}$ may be parameterised directly as

$$\mathbf{C} = \begin{bmatrix} C_x \\ C_y \\ C_z \end{bmatrix} , \tag{4}$$

this would be highly over-parameterised in the case of $\mathtt{R}$ as its full (orthonormal matrix) representation has 9 elements, but only 3 degrees of freedom. Therefore we reparametrise the rotation using the *angle-axis* formulation:

$$\mathtt{R} = \cos(\alpha)\mathtt{I}_{3\times 3} + \sin(\alpha)\left[\mathbf{a}\right]_\times + (1-\cos(\alpha))\mathbf{a}\mathbf{a}^\top , \tag{5}$$

using a rotation axis $\mathbf{a}$ and a rotation angle α, where $\left[\cdot\right]_\times$ denotes the *skew-symmetric* matrix of cross product. These have in total 4 elements, which is still an over-parameterisation. However, we can exploit the nature of the rotation axis vector $\mathbf{a}$ and assume it is normalised to unit length. Then it is possible to represent both $\mathbf{a}$ and α by one entity

$$\mathbf{r} = \begin{bmatrix} r_1 \\ r_2 \\ r_3 \end{bmatrix} = \alpha\mathbf{a} , \tag{6}$$

$$\alpha = ||\mathbf{r}|| , \quad \mathbf{a} = \mathbf{r}/||\mathbf{r}|| . \tag{7}$$

Substituting Equation (5) into (2), we get the following scalar projection formulae:

$$\begin{aligned} \widetilde{u}_1 = {} & (x-C_x)\left(k_{11}\left(\tfrac{r_1^2(1-\cos(\alpha))}{\alpha^2} + \cos(\alpha)\right) + k_{13}\left(\tfrac{r_1 r_3(1-\cos(\alpha))}{\alpha^2} - \tfrac{r_2\sin(\alpha)}{\alpha}\right)\right) \\ & +(y-C_y)\left(k_{11}\left(\tfrac{r_1 r_2(1-\cos(\alpha))}{\alpha^2} - \tfrac{r_3\sin(\alpha)}{\alpha}\right) + k_{13}\left(\tfrac{r_2 r_3(1-\cos(\alpha))}{\alpha^2} + \tfrac{r_1\sin(\alpha)}{\alpha}\right)\right) \\ & +(z-C_z)\left(k_{11}\left(\tfrac{r_1 r_3(1-\cos(\alpha))}{\alpha^2} + \tfrac{r_2\sin(\alpha)}{\alpha}\right) + k_{13}\left(\tfrac{r_3^2(1-\cos(\alpha))}{\alpha^2} + \cos(\alpha)\right)\right) , \end{aligned} \tag{8}$$

$$\begin{aligned} \widetilde{u}_2 = {} & (x-C_x)\left(k_{11}\left(\tfrac{r_1 r_2(1-\cos(\alpha))}{\alpha^2} + \tfrac{r_3\sin(\alpha)}{\alpha}\right) + k_{23}\left(\tfrac{r_1 r_3(1-\cos(\alpha))}{\alpha^2} - \tfrac{r_2\sin(\alpha)}{\alpha}\right)\right) \\ & +(y-C_y)\left(k_{11}\left(\tfrac{r_2^2(1-\cos(\alpha))}{\alpha^2} + \cos(\alpha)\right) + k_{23}\left(\tfrac{r_2 r_3(1-\cos(\alpha))}{\alpha^2} + \tfrac{r_1\sin(\alpha)}{\alpha}\right)\right) \\ & +(z-C_z)\left(k_{11}\left(\tfrac{r_2 r_3(1-\cos(\alpha))}{\alpha^2} - \tfrac{r_1\sin(\alpha)}{\alpha}\right) + k_{23}\left(\tfrac{r_3^2(1-\cos(\alpha))}{\alpha^2} + \cos(\alpha)\right)\right) , \end{aligned} \tag{9}$$

and

$$\begin{aligned}\widetilde{u}_3 = &(x - C_x)\left(\tfrac{r_1 r_3(1-\cos(\alpha))}{\alpha^2} - \tfrac{r_2 \sin(\alpha)}{\alpha}\right)\\ &+(y - C_y)\left(\tfrac{r_2 r_3(1-\cos(\alpha))}{\alpha^2} + \tfrac{r_1 \sin(\alpha)}{\alpha}\right)\\ &+(z - C_z)\left(\tfrac{r_3^2(1-\cos(\alpha))}{\alpha^2} + \cos(\alpha)\right) .\end{aligned} \tag{10}$$

Then we can compute u and v simply as

$$u = \frac{\widetilde{u}_1}{\widetilde{u}_3} , \tag{11}$$

$$v = \frac{\widetilde{u}_2}{\widetilde{u}_3} . \tag{12}$$

3 Jacobian Derivation

The next step is symbolic differentiation of the point coordinates and simplification of the resulting formulae [4]. We substitute Equations (7) and (8–10) into Equations (11,12), and differentiate with respect to all the camera parameters:

$$\mathtt{J} = \begin{bmatrix} \frac{\partial u}{\partial r_1} & \frac{\partial u}{\partial r_2} & \frac{\partial u}{\partial r_3} & \frac{\partial u}{\partial C_x} & \frac{\partial u}{\partial C_y} & \frac{\partial u}{\partial C_z} \\ \frac{\partial v}{\partial r_1} & \frac{\partial v}{\partial r_2} & \frac{\partial v}{\partial r_3} & \frac{\partial v}{\partial C_x} & \frac{\partial v}{\partial C_y} & \frac{\partial v}{\partial C_z} \end{bmatrix} . \tag{13}$$

This yields a rather complex set of derivatives. Therefore we show only two of them symbolically, as examples. Section 4 includes the expressions for the full Jacobian as source code.

$$\begin{aligned}\tfrac{\partial u}{\partial C_x} = \; & k_{11}\alpha^4(r_2(C_z r_2 - C_y r_3 + r_3 y - r_2 z)\\ &+(C_z(r_1^2 + r_3^2) + r_2 r_3(C_y - y) - (r_1^2 + r_3^2)z)\cos(\alpha)\\ &+r_1\alpha(C_y - y)\sin(\alpha))\\ &(r_3\alpha(C_x r_1 + C_y r_2 + C_z r_3 - r_1 x - r_2 y - r_3 z)\\ &+\alpha(C_z(r_1^2 + r_2^2) + r_3(-C_x r_1 - C_y r_2 + r_1 x + r_2 y) - (r_1^2 + r_2^2)z)\cos(\alpha)\\ &+\alpha^2(C_y r_1 - C_x r_2 + r_2 x - r_1 y)\sin(\alpha))^{-2} ,\end{aligned} \tag{14}$$

$$
\begin{aligned}
\frac{\partial u}{\partial r_1} = \;& \alpha^6(\alpha^{-8}(-(k_{11}r_1+k_{13}r_3)\alpha(C_xr_1+C_yr_2+C_zr_3-r_1x-r_2y-r_3z)\\
&+\alpha(C_yk_{11}r_1r_2-C_xk_{11}r_2^2-C_zk_{13}(r_1^2+r_2^2)+C_zk_{11}r_1r_3+C_xk_{13}r_1r_3\\
&+C_yk_{13}r_2r_3-C_xk_{11}r_3^2+k_{11}r_2^2x-k_{13}r_1r_3x+k_{11}r_3^2x\\
&-k_{11}r_1r_2y-k_{13}r_2r_3y+k_{13}r_1^2z+k_{13}r_2^2z-k_{11}r_1r_3z)\cos(\alpha)\\
&-\alpha^2(C_yk_{13}r_1+C_zk_{11}r_2-C_xk_{13}r_2-C_yk_{11}r_3+k_{13}r_2x-k_{13}r_1y\\
&+k_{11}r_3y-k_{11}r_2z)\sin(\alpha))\\
&(r_3\alpha(-2C_yr_1r_2-2C_zr_1r_3+C_x(-r_1^2+r_2^2+r_3^2)+r_1^2x-r_2^2x-r_3^2x\\
&+2r_1r_2y+2r_1r_3z)\\
&+\alpha(2C_zr_1r_3^2-C_x(r_1^3r_2+r_1r_2^3-r_1^2r_3+r_2^2r_3+r_1r_2r_3^2+r_3^3)\\
&+C_yr_1(r_1^3+2r_2r_3+r_1(r_2^2+r_3^2))+r_1^3r_2x+r_1r_2^3x-r_1^2r_3x+r_2^2r_3x\\
&+r_1r_2r_3^2x+r_3^3x-r_1^4y-r_1^2r_2^2y-2r_1r_2r_3y-r_1^2r_3^2y-2r_1r_3^2z)\cos(\alpha)\\
&-\alpha^2(-C_yr_2^2+C_zr_1(r_1^2+r_2^2)-C_yr_1r_2r_3-C_yr_3^2-C_xr_1(r_2+r_1r_3)\\
&+r_1r_2x+r_1^2r_3x+r_2^2y+r_1r_2r_3y+r_3^2y-r_1(r_1^2+r_2^2)z)\sin(\alpha))\\
&+\alpha^{-3}(r_3\alpha(-C_xr_1-C_yr_2-C_zr_3+r_1x+r_2y+r_3z)\\
&+\alpha(-C_z(r_1^2+r_2^2)+r_3(C_xr_1+C_yr_2-r_1x-r_2y)+(r_1^2+r_2^2)z)\cos(\alpha)\\
&-\alpha^2(C_yr_1-C_xr_2+r_2x-r_1y)\sin(\alpha))\\
&(\alpha^{-4}(C_xk_{13}r_1^3r_2+2C_xk_{11}r_1r_2^2+C_xk_{13}r_1r_2^3-C_xk_{13}r_1^2r_3+C_xk_{13}r_2^2r_3\\
&+2C_xk_{11}r_1r_3^2+C_xk_{13}r_1r_2r_3^2+C_xk_{13}r_3^3\\
&-C_z(2k_{13}r_1r_3^2+k_{11}(r_1^3r_2+r_1^2r_3+r_1r_2(r_2^2+r_3^2)-r_3(r_2^2+r_3^2)))\\
&+C_y(-k_{13}r_1(r_1^3+2r_2r_3+r_1(r_2^2+r_3^2))\\
&+k_{11}(-r_1^2r_2+r_1^3r_3+r_2(r_2^2+r_3^2)+r_1r_3(r_2^2+r_3^2)))\\
&-k_{13}r_1^3r_2x-2k_{11}r_1r_2^2x-k_{13}r_1r_2^3x+k_{13}r_1^2r_3x-k_{13}r_2^2r_3x\\
&-2k_{11}r_1r_3^2x-k_{13}r_1r_2r_3^2x-k_{13}r_3^3x+k_{13}r_1^4y+k_{11}r_1^2r_2y\\
&+k_{13}r_1^2r_2^2y-k_{11}r_2^3y-k_{11}r_1^3r_3y+2k_{13}r_1r_2r_3y-k_{11}r_1r_2^2r_3y\\
&+k_{13}r_1^2r_3^2y-k_{11}r_2r_3^2y-k_{11}r_1r_3^3y\\
&+(2k_{13}r_1r_3^2+k_{11}(r_1^3r_2+r_1^2r_3+r_1r_2(r_2^2+r_3^2)-r_3(r_2^2+r_3^2)))z)\cos(\alpha)\\
&+\alpha^{-5}(\alpha(C_zk_{11}r_1^2r_3-C_zk_{11}r_2^2r_3+2C_zk_{13}r_1r_3^2-C_zk_{11}r_3^3\\
&+C_yr_2(2k_{13}r_1r_3+k_{11}(r_1^2-r_2^2-r_3^2))\\
&-C_x(2k_{11}r_1(r_2^2+r_3^2)+k_{13}r_3(-r_1^2+r_2^2+r_3^2))\\
&+2k_{11}r_1r_2^2x-k_{13}r_1^2r_3x+k_{13}r_2^2r_3x+2k_{11}r_1r_3^2x+k_{13}r_3^3x\\
&-k_{11}r_1^2r_2y+k_{11}r_2^3y-2k_{13}r_1r_2r_3y+k_{11}r_2r_3^2y\\
&+r_3(-2k_{13}r_1r_3+k_{11}(-r_1^2+r_2^2+r_3^2))z)\\
&+\alpha^2(-C_yk_{11}r_1^2r_2-C_yk_{13}r_2^2-C_yk_{11}r_1r_3-C_yk_{13}r_1r_2r_3-C_yk_{13}r_3^2\\
&+C_zr_1(k_{13}(r_1^2+r_2^2)+k_{11}(r_2-r_1r_3))\\
&+C_xr_1(-k_{13}(r_2+r_1r_3)+k_{11}(r_2^2+r_3^2))\\
&+k_{13}r_1r_2x-k_{11}r_1r_2^2x+k_{13}r_1^2r_3x-k_{11}r_1r_3^2x+k_{11}r_1^2r_2y\\
&+k_{13}r_2^2y+k_{11}r_1r_3y+k_{13}r_1r_2r_3y+k_{13}r_3^2y\\
&-r_1(k_{13}(r_1^2+r_2^2)+k_{11}(r_2-r_1r_3))z)\sin(\alpha))))\\
&(r_3\alpha(C_xr_1+C_yr_2+C_zr_3-r_1x-r_2y-r_3z)\\
&+\alpha(C_z(r_1^2+r_2^2)+r_3(-C_xr_1-C_yr_2+r_1x+r_2y)-(r_1^2+r_2^2)z)\cos(\alpha)\\
&+\alpha^2(C_yr_1-C_xr_2+r_2x-r_1y)\sin(\alpha))^{-2}\ .
\end{aligned}
\tag{15}
$$

4 Common Subexpression Elimination

It is apparent that the partial derivatives contain many repeated subexpressions. Avoiding multiple evaluation of these decreases computation cost. Additionally, some of the expressions are independent of the particular point ($\mathbf{X}$, $\mathbf{u}$) and can be precomputed for further speed-up. We used an approach of [1], which exploits the internal subexpression elimination of Mathematica [4] to generate a C++ code with minimal execution time.

```
    double v000 = pow(r3,2);
    double v001 = pow(r2,2);
    double v002 = pow(r1,2);
    double v003 = pow(r3,3);
    double v004 = pow(r2,3);
    double v005 = pow(r1,3);
    double v006 = v000 + v001 + v002;
    double v007 = 1/v006;
    double v008 = pow(v006,-2);
    double v009 = sqrt(v006);
    double v010 = pow(v006,-1.5);
    double v011 = sin(v009);
    double v012 = cos(v009);
    double v013 = -v012;
    double v014 = 1 + v013;
    double v015 = r1 * r3 * v007 * v012;
    double v016 = v011/sqrt(v006);
    double v017 = r2 * r3 * v007 * v012;
    double v018 = r1 * r2 * v007 * v012;
    double v019 = r3 * v016;
    double v020 = r1 * v016;
    double v021 = r2 * v016;
    double v022 = r2 * r3 * v007 * v013;
    double v023 = r1 * r3 * v007 * v013;
    double v024 = -v016;
    double v025 = r1 * r2 * v007 * v013;
    double v026 = -v021;
    double v027 = -v020;
    double v028 = -v019;
    double v029 = r2 * r3 * v010 * v011;
    double v030 = r1 * r3 * v010 * v011;
    double v031 = r1 * r2 * v010 * v011;
    double v032 = -v030;
    double v033 = r2 * v007 * v014;
    double v034 = r1 * v007 * v014;
    double v035 = r1 * v029;
    double v036 = -v029;
    double v037 = v000 * v010 * v011;
    double v038 = v001 * v010 * v011;
    double v039 = -v031;
    double v040 = r3 * v007 * v014;
    double v041 = v002 * v010 * v011;
    double v042 = r1 * v033;
    double v043 = r3 * v034;
    double v044 = r3 * v033;
    double v045 = r2 * v037;
    double v046 = r1 * v037;
    double v047 = r3 * v038;
    double v048 = r3 * v041;
    double v049 = r1 * v038;
    double v050 = r2 * v041;
    double v051 = v000 * v007 * v014;
    double v052 = -2 * r1 * r2 * r3 * v008 * v014;
    double v053 = -2 * r2 * v000 * v008 * v014;
    double v054 = -2 * r1 * v000 * v008 * v014;
    double v055 = -2 * r3 * v001 * v008 * v014;
    double v056 = -2 * r3 * v002 * v008 * v014;
    double v057 = -2 * r1 * v001 * v008 * v014;
    double v058 = -2 * r2 * v002 * v008 * v014;
    double v059 = v012 + v051;
    double v060 = k11 * (v012 + v001 * v007 * v014);
    double v061 = k23 * v059;
    double v062 = k13 * v059;
    double v063 = k11 * (v012 + v002 * v007 * v014);
    double v064 = v013 - v051;
    double v065 = v020 + v044;
    double v066 = v026 + v043;
    double v067 = v021 - v043;
    double v068 = k11 * (v019 + v042);
    double v069 = k23 * v065;
    double v070 = k13 * v065;
    double v071 = k11 * (v021 + v043);
    double v072 = v027 - v044;
    double v073 = k23 * v066;
    double v074 = k11 * (v027 + v044);
    double v075 = k11 * (v028 + v042);
    double v076 = k13 * v066;
    double v077 = v026 + v045 + v053;
    double v078 = v027 + v046 + v054;
    double v079 = v060 + v069;
    double v080 = v062 + v071;
    double v081 = v061 + v074;
    double v082 = v063 + v076;
    double v083 = v003 * v010 * v011 - 2 * v003 * v008 * v014 + v028
           + 2 * v040;
    double v084 = v068 + v073;
    double v085 = v070 + v075;
    double v086 = v002 * v007 * v012 + v016 + v035 - v041 + v052;
    double v087 = v001 * v007 * v013 + v024 + v035 + v038 + v052;
    double v088 = v015 + v032 + v033 + v045 + v053;
    double v089 = v022 + v029 + v034 + v046 + v054;
    double v090 = v018 + v039 + v040 + v047 + v055;
    double v091 = v025 + v031 + v040 + v048 + v056;
    double v092 = k11 * (v001 * v007 * v012 + v016 + v035 - v038 + v052)
           + k13 * v077;
    double v093 = k11 * (v027 + v049 + v057) + k23 * v086;
    double v094 = k11 * (v026 + v050 + v058) + k13 * v087;
    double v095 = k11 * (v002 * v007 * v013 + v024 + v035 + v041 + v052)
           + k23 * v078;
    double v096 = k11 * (v028 + v047 + v055) + k23 * v088;
    double v097 = k11 * (v028 + v048 + v056) + k13 * v089;
    double v098 = k11 * (v025 + v031 + v040 + v047 + v055) + k23 * v077;
    double v099 = k11 * (v018 + v039 + v040 + v048 + v056) + k13 * v078;
    double v100 = k11 * (v023 + v030 + v033 + v045 + v053) + k23 * v083;
    double v101 = k11 * (v017 + v034 + v036 + v046 + v054) + k13 * v083;
    double v102 = k11 * (v004 * v010 * v011 - 2 * v004 * v008 * v014 + v026
           + 2 * v033)+ k23 * v090;
    double v103 = k11 * (v005 * v010 * v011 - 2 * v005 * v008 * v014 + v027
           + 2 * v034) + k13 * v091;
    double v104 = k11 * (v000 * v007 * v012 + v016 + v035 - v037 + v052)
           + k23 * v089;
    double v105 = k11 * (v023 + v030 + v033 + v050 + v058) + k13 * v086;
    double v106 = k11 * (v000 * v007 * v013 + v024 + v035 + v037 + v052)
           + k13 * v088;
    double v107 = k11 * (v017 + v034 + v036 + v049 + v057) + k23 * v087;
    double v108 = k11 * (v022 + v029 + v034 + v049 + v057) + k13 * v090;
    double v109 = k11 * (v015 + v032 + v033 + v050 + v058) + k23 * v091;
```

```
// Everything so far was point-independent, therefore could be precomputed only once for all.
// Since now, we start using elements of X and therefore this needs to be done once for each X.
for ( unsigned p = 0; p < NUM_POINTS; ++p ) {
    double x = X[p][0];
    double y = X[p][1];
    double z = X[p][2];

    double v110 = -(Cz * v059) - Cy * v065 - Cx * v066 + v066 * x + v065 * y + v059 * z;
    double v111 = pow(v110,-2);
    double v112 = 1/v110;
    double v113 = -(Cy * v079) - Cz * v081 - Cx * v084 + v084 * x + v079 * y + v081 * z;
    double v114 = -(Cz * v080) - Cx * v082 - Cy * v085 + v082 * x + v085 * y + v080 * z;
    double v115 = -(Cz * v078) - Cy * v086 - Cx * v091 + v091 * x + v086 * y + v078 * z;
```

```
    double v116 = -(Cz * v077) - Cx * v087 - Cy * v090 + v087 * x + v090 * y + v077 * z;
    double v117 = -(Cz * v083) - Cy * v088 - Cx * v089 + v089 * x + v088 * y + v083 * z;

    // COEFFICIENTS FOR THE 1ST ROW (du by _)
    // dr1
    J[p][0][0] = -(v111 * v114 * v115) + v112 * (-(Cz * v099) - Cx * v103 - Cy * v105 + v103 * x + v105 * y + v099 * z);
    // dr2
    J[p][0][1] = -(v111 * v114 * v116) + v112 * (-(Cz * v092) - Cx * v094 - Cy * v108 + v094 * x + v108 * y + v092 * z);
    // dr3
    J[p][0][2] = -(v111 * v114 * v117) + v112 * (-(Cx * v097) - Cz * v101 - Cy * v106 + v097 * x + v106 * y + v101 * z);
    // dCx
    J[p][0][3] = (-v063 - v076) * v112 - v067 * v111 * v114;
    // dCy
    J[p][0][4] = (-v070 - v075) * v112 - v072 * v111 * v114;
    // dCz
    J[p][0][5] = (-v062 - v071) * v112 - v064 * v111 * v114;


    //COEFFICIENTS FOR THE 2ND ROW (dv by _)
    // dr1
    J[p][1][0] = -(v111 * v113 * v115) + v112 * (-(Cy * v093) - Cz * v095 - Cx * v109 + v109 * x + v093 * y + v095 * z);
    // dr2
    J[p][1][1] = -(v111 * v113 * v116) + v112 * (-(Cz * v098) - Cy * v102 - Cx * v107 + v107 * x + v102 * y + v098 * z);
    // dr3
    J[p][1][2] = -(v111 * v113 * v117) + v112 * (-(Cy * v096) - Cz * v100 - Cx * v104 + v104 * x + v096 * y + v100 * z);
    // dCx
    J[p][1][3] = (-v068 - v073) * v112 - v067 * v111 * v113;
    // dCy
    J[p][1][4] = (-v060 - v069) * v112 - v072 * v111 * v113;
    // dCz
    J[p][1][5] = (-v061 - v074) * v112 - v064 * v111 * v113;
}
```

References

[1] R. S. Ballard. Converting symbolic Mathematica expressions to C code, 2009.

[2] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, second edition, 2004.

[3] K. Lebeda, S. Hadfield, and R. Bowden. 2D or not 2D: Bridging the gap between tracking and structure from motion. In *Proc. of ACCV*, 2014.

[4] Wolfram Research, Inc. Mathematica, 2012.