Treball de Fi de Grau

Grau en Enginyeria en Tecnologies Industrials

Control d'impedància variable amb primitives de moviment en espais latents amb comportament dòcil

ANNEXOS

David Parent Alonso

Directors: Adrià Colomé Figueras
Carme Torras Genís

Ponent: Enric Fossas Colet

Juny 2019

Escola Tècnica Superior d'Enginyeria Industrial de Barcelona

UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

Institut de Robòtica i Informàtica Industrial

CSIC
Consejo Superior de Investigaciones Científicas

Institut de Robòtica i Informàtica Industrial (IRI)
Consejo Superior de Investigaciones Científicas (CSIC)
Universitat Politècnica de Catalunya (UPC)
Llorens i Artigas 4-6, 08028, Barcelona, Spain
Tel (fax): +34 93 401 5750 (5751)
http://www.iri.upc.edu

Corresponding author:
David Parent Alonso
dparent@iri.upc.edu
http://www.iri.upc.edu/staff/dparent

Continguts

A

Pseudo-inversa esmorteïda

Tota matriu real $\boldsymbol{A}$ de dimensió $m \times n$ es pot descomposar en el seus valors singulars (SVD) segons

$$\boldsymbol{A} = \boldsymbol{U}\boldsymbol{S}\boldsymbol{V}^T \tag{A.1}$$

on

$\boldsymbol{U}$ és una $m \times n$ ortogonal construïda amb els vectors propis de $\boldsymbol{A}\boldsymbol{A}^T$.

$\boldsymbol{S}$ és una matriu $n \times n$ diagonal amb els valors propis al quadrat de $\boldsymbol{A}$.

$\boldsymbol{V}$ és una matriu $n \times n$ ortogonal construïda amb els vectors propis de $\boldsymbol{A}^T\boldsymbol{A}$.

Per invertir una matriu qualsevol mitjançant la SVD s'aprofita que tant $\boldsymbol{U}$ com $\boldsymbol{V}$ són ortogonals i, per tant, la seva inversa és la seva transposada. A més, la matriu $\boldsymbol{S}$ és diagonal, per tant, la seva inversa és la inversa dels valors de la seva diagonal. D'aquesta manera s'obté:

$$\boldsymbol{A}^{-1} = \boldsymbol{V}\boldsymbol{S}^{-1}\boldsymbol{U}^T \tag{A.2}$$

Si la matriu $\boldsymbol{A}$ no té rang màxim, algun dels seus valors propis serà 0. Per tant, quan invertim la matriu $\boldsymbol{S}$ aquell element corresponent al vap 0 quedarà indeterminat (1/0). Per solucionar aquest problema s'implementa la pseudo-inversa esmorteïda:

$$\boldsymbol{S}_i^{-1} = \frac{\boldsymbol{\sigma}_i}{\boldsymbol{\sigma}_i^2 + \boldsymbol{\lambda}^2} \tag{A.3}$$

D'aquesta manera si $\boldsymbol{\sigma}$ és 0, s'obté $\boldsymbol{S}_i^{-1}$ 0 també. Per un valor de λ suficientment petit (de l'odre de 10^{-3}), els altres elements de $\boldsymbol{S}^{-1}$ no queden alterats.

B

Quaternió sense la component escalar

L'orientació d'un sòlid rígid es pot descriure de diferents maneres. La més utilitzada en el camp de la robòtica és el quaternió, ja que no té els problemes dels angles d'Euler (*gimbal lock*) i, a més, permet interpolar-los fàcilment, donant lloc a transicions suaus.

Normalment les trajectòries generades es descriuen amb 7 components: 3 de posició i 4 d'orientació $[x, y, z, \eta, \epsilon_x, \epsilon_y, \epsilon_z]$. Tot i això, realment només es tenen 6 graus de llibertat a l'espai operacional. Aquesta diferència es deu a que les 4 components del quaternió no són independents, sinó que estan relacionades per

$$\eta^2 + \epsilon_x^2 + \epsilon_y^2 + \epsilon_z^2 = 1 \tag{B.1}$$

El problema és que, com s'ha vist als apartats 5.3 i 6 en tot moment s'utilitzen equacions i matrius que treballen amb els 6 graus de llibertat de l'espai i no amb les 7 components que descriuen la trajectòria. Això es deu a que quan es fa aprenentatge per reforçament (*reinforcement learning*) es treballa amb els graus de llibertat de la tasca i, per tant, l'orientació s'ha d'expressar únicament amb 3 components. Si es treballés amb les 4 components del quaternió, els models pertorbats del quaternió generarien un quaternió no unitari, és a dir, no complirien l'equació que els relaciona.

Sabent que un quaternió es pot expressar en funció d'un angle girat respecte d'un eix com

$$\boldsymbol{\mathcal{Q}} = e^{\frac{\theta}{2}(u_x\boldsymbol{i}+u_y\boldsymbol{j}+u_z\boldsymbol{k})} = \cos\frac{\theta}{2} + (u_x\boldsymbol{i} + u_y\boldsymbol{j} + u_z\boldsymbol{k})\sin\frac{\theta}{2} \tag{B.2}$$

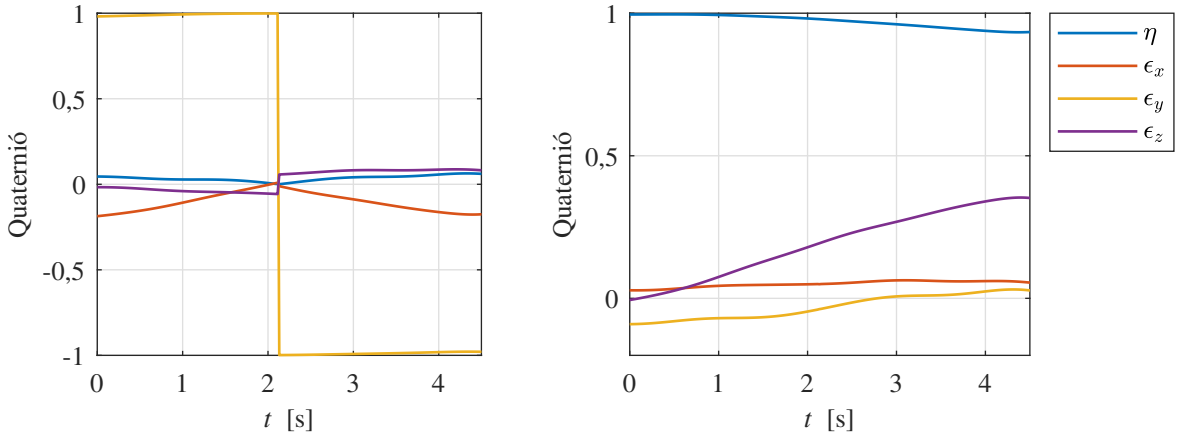
es pot excloure la part del cosinus (així obtenint una expressió de 3 components) i, posteriorment, reconstruir-la amb

$$\eta = \cos(\arcsin(\|\epsilon_x\epsilon_y\epsilon_z\|)) \tag{B.3}$$

Amb aquesta última expressió queda clar que el fet de reconstruir aquesta component comporta un problema: es perd el signe. Els quaternions representen un doble recobriment del grup SO(3). En conseqüència, $\mathcal{Q} = -\mathcal{Q}$. Per conveni, s'utilitza que la part escalar del quaternió és positiva i, per tant, queden definits els signes de les altres components. El problema és que si no es té η, no se sap quins dels dos quaternions s'està utilitzant, si $\mathcal{Q}$ o $-\mathcal{Q}$.

Per resoldre-ho, es fa un canvi de base. En comptes de treballar en la base del robot on els angles són grans (figura B.1a) , es passa a treballar a la base de l'efector final on els angles són més petits (figura B.1b). D'aquesta manera, es pot treballar només amb la part vectorial del quaternió i evitar els canvis de signe. Només cal tenir en compte que quan es reconstrueix cal rotar-lo un altre cop a la base del robot amb la matriu $R0$.

El problema de la pèrdua de signe també pot aparèixer quan es troba un quaternió a partir d'una matriu de rotació. La conversió d'una matriu de rotació a un quaternió no és única ja que, com s'ha dit, un quaternió i el seu negatiu representen la mateixa orientació. Per resoldre-ho, normalment es treballa amb el quaternió amb $\eta > 0$. El problema ve quan η és molt propera



(a) Base del robot **(b)** Base de l'efector final

Fig. B.1: Comparació del quaternió expressat en diferents bases

a zero. Per solucionar-ho, s'ha trobat a la literatura [1] un mètode per reconstruir el quaternió d'una manera més acurada. Aplicant la tècnica descrita, s'obté la figura de la dreta

A la vista d'aquests resultats, s'ha decidit utilitzar sempre aquesta metodologia per trobar un quaternió a partir d'una matriu de rotació i així evitar canvis de signe no desitjats.

C

Codi font

A les següents pàgines es troben els codis en C++ creats en aquest projecte. Només es presenten el bucles principals dels codis o, en alguns casos, només les parts més rellevants (càlcul de la $\boldsymbol{K}_P$). Tot i això, en aquest projecte s'han utilitzat altres codis ja desenvolupats a l'IRI, com la generació de les ProMP i el càlcul de la reducció de la dimensionalitat. D'aquests últims es poden consultar els algoritmes a les referències [2] i [3], respectivament.

C.1 Generadors de trajectòries

C.1.1 Interpolar entre dos punts

```
virtual void operate() {

if(this->timeValue.valueDefined() && startflag_==1) {
    local_time = this->timeValue.getValue();

    if(local_time <finalTime_){
        q_cart_aux = (local_time/finalTime_)*q_cart_fin + (1-
local_time/finalTime_)*q_cart_ini;
        double quat_time;
        quat_time=local_time/finalTime_;
        q_quat_aux = ac_math::SLERP(q_quat_ini,q_quat_fin,quat_time);

    }else{
        q_cart_aux=q_cart_fin;
        q_quat_aux=q_quat_fin;
    }

    State_(0)=q_cart_aux(0);
    State_(1)=q_cart_aux(1);
    State_(2)=q_cart_aux(2);
    State_(3)=q_quat_aux(0);
    State_(4)=q_quat_aux(1);
    State_(5)=q_quat_aux(2);
    State_(6)=q_quat_aux(3);

    if(local_time>finalTime_ && endflag_==0){
        endflag_=1;
    }
}
this->StateOutput_value->setData(&State_);
}
```

C.1.2 Trajectòria ProMP

```
virtual void operate() {

if(this->timeValue.valueDefined() && startflag==1){
    local_time = this->timeValue.getValue();
    jp_type q(this->jpInput.getValue());
    Sigma_t = this->Sigma_t_input.getValue();

    traj_time = traj_time*startflag + Ts*(1.0-pauseflag);

    assert(!std::isnan(traj_time));

    if(traj_time<tau_){
        ongoing_ = true;
        ac_promp::EvalProMPCartesian(y,Nf,C,D,dof,mw,traj_time/tau_);

        if (isCartesian_==1){
            y=ac_promp::y2quat(y,R0_);
            index++;
        }

        Sigma_t_aux.block(0,0,dof,dof)=Sigma_t.block(0,0,dof,dof);

        for(int ii=0; ii<7; ii++){
            local_pos(ii)=y(ii);
        }

        }else{
            ongoing_ = false;

            for(int ii=0; ii<7; ii++){
                local_pos(ii)=y(ii);
            }

            local_vel.fill(0.0);
            local_acc.fill(0.0);
        }

    }else{
        for(int ii=0; ii<7; ii++){
            local_pos(ii)=y(ii);
        }

        local_vel.fill(0.0);
        local_acc.fill(0.0);
    }

this->out_pos_value->setData(&local_pos);
this->out_vel_value->setData(&local_vel);
this->out_accel_value->setData(&local_acc);


for(int i=0;i<6;i++){
    for(int j=0;j<6;j++){
        Sout(i,j)=Sigma_t_aux(i,j);
    }
}

this->Sigma_t_value->setData(&Sout);
```

C.1.3 Adaptació de la trajectòria

```
T_aux=WAM_kinematics.forwardkinematics(q);

actual_pos(0) = T_aux(0,3); // current position vector [x,y,z]
actual_pos(1) = T_aux(1,3);
actual_pos(2) = T_aux(2,3);

actual_rot = T_aux.block(0,0,3,3); // current rotation matrix 3x3

// Compute cartesian error
Eigen::VectorXd poserr(3),orierr(3);
Eigen::MatrixXd desired_rot(3,3);
Eigen::Vector3d r0,r1,r2,rd0,rd1,rd2;
double nu,ex,ey,ez;

Eigen::VectorXd local_pos_eigen(7);
local_pos_eigen = local_pos; // Convert from jptype to eigen vector

desired_pos(0)=local_pos_eigen(0);
desired_pos(1)=local_pos_eigen(1);
desired_pos(2)=local_pos_eigen(2);

nu = local_pos_eigen(3); // quaternion scalar part
ex = local_pos_eigen(4); // quaternion x-vector part
ey = local_pos_eigen(5); // quaternion y-vector part
ez = local_pos_eigen(6); // quaternion z-vector part

// Compute desired rotation matrix 3x3
desired_rot(0,0) = 2*(pow(nu,2)+pow(ex,2))-1;
desired_rot(0,1) = 2*(ex*ey-nu*ez);
desired_rot(0,2) = 2*(ex*ez+nu*ey);
desired_rot(1,0) = 2*(ex*ey+nu*ez);
desired_rot(1,1) = 2*(pow(nu,2)+pow(ey,2))-1;
desired_rot(1,2) = 2*(ey*ez-nu*ex);
desired_rot(2,0) = 2*(ex*ez-nu*ey);
desired_rot(2,1) = 2*(ey*ez+nu*ex);
desired_rot(2,2) = 2*(pow(nu,2)+pow(ez,2))-1;

poserr = desired_pos-actual_pos; // position error

r0=actual_rot.col(0);
r1=actual_rot.col(1);
r2=actual_rot.col(2);
rd0=desired_rot.col(0);
rd1=desired_rot.col(1);
rd2=desired_rot.col(2);

orierr = 0.5*( r0.cross(rd0) + r1.cross(rd1) + r2.cross(rd2) ) ; // orientation error

carterror2=0.0;
pauseflag=0.0;
for(int kk=0;kk<3;kk++){
    if(fabs(poserr(kk))>0.02){ // 2 cm
        carterror2=carterror2+5.0*poserr(kk)*poserr(kk);
    }
    if(fabs(poserr(kk))>0.10){ // 10 cm
        pauseflag=1.0;
    }
}

traj_time=traj_time+((double) startflag)*DT/(1+Kt*carterror2)*(1.0-pauseflag);
```

C.2 Controladors

C.2.1 Controlador cartesià

```
virtual void operate(){

if(startflag==1){

    // Get Jacobian Matrix
    cp_type tcp(this->TCP.getValue());
    jacfound=bt_kinematics_eval_jacobian(this-
>kin_Input.getValue().impl,DOF,tcp.asGslType(),J);

    J_foo.copyFrom(J); // from gsl type to math matrix type
    Je.resize(6,7);
    Je=J_foo; // from math matrix to eigen matrix type

    // Get desired position and orientation (xyz and rotation matrix)
    jp_type pd(this->pd_Input.getValue()); // pd = [x,y,z,nu,ex,ey,ez]

    posd(0)= pd(0); // desired position vector [x,y,z]
    posd(1)= pd(1);
    posd(2)= pd(2);

    nu = pd(3); // quaternion scalar part
    ex = pd(4); // quaternion x-vector part
    ey = pd(5); // quaternion y-vector part
    ez = pd(6); // quaternion z-vector part

    // Compute Rd (desired rotation matrix 3x3)
    Rd(0,0) = 2*(pow(nu,2)+pow(ex,2))-1;
    Rd(0,1) = 2*(ex*ey-nu*ez);
    Rd(0,2) = 2*(ex*ez+nu*ey);
    Rd(1,0) = 2*(ex*ey+nu*ez);
    Rd(1,1) = 2*(pow(nu,2)+pow(ey,2))-1;
    Rd(1,2) = 2*(ey*ez-nu*ex);
    Rd(2,0) = 2*(ex*ez-nu*ey);
    Rd(2,1) = 2*(ey*ez+nu*ex);
    Rd(2,2) = 2*(pow(nu,2)+pow(ez,2))-1;

    // Get current position and orientation (read joint position-> forward
kinematics-> [x,y,z] and [R])
    jp_type jp(this->jp_Input.getValue());
    T_aux=WAM_kinematics.forwardkinematics(jp);

    pos(0) = T_aux(0,3); // current position vector [x,y,z]
    pos(1) = T_aux(1,3);
    pos(2) = T_aux(2,3);

    R = T_aux.block<3,3>(0,0); // current rotation matrix 3x3

    // Compute end-effector desired velocity
    if (iteration_index==0){
        posdbef = posd; // initialize for the first iteration (initial
velocities=0.0)
        Rdbef = Rd;
    }

    dpd = (posd-posdbef)/0.02; // linear velocities vector [vx vy vz]

    S = ((Rd-Rdbef)/0.02) * (Rdbef.transpose()); // skew-symmetric matrix containing
angular velocities

    wd(0) = (S(2,1) - S(1,2))/2; // angular velocities vector [wx wy wz]
```

```
    wd(1) = (S(0,2) - S(2,0))/2;
    wd(2) = (S(1,0) - S(0,1))/2;

    vd(0) = dpd(0);
    vd(1) = dpd(1);
    vd(2) = dpd(2);
    vd(3) = wd(0);
    vd(4) = wd(1);
    vd(5) = wd(2);

    // Update posd and Rd
    posdbef = posd;
    Rdbef = Rd;

    // Compute end-effector actual velocity (not using v=J*dq because the system
becomes unstable)
    if (iteration_index==0){
        posbef = pos;
        Rbef = R;
    }

    dp = (pos-posbef)/0.02; // linear velocities vector [vx vy vz]

    Sd = ((R-Rbef)/0.02) * (Rbef.transpose()); // skew-symmetric matrix containing
angular velocities

    w(0) = (Sd(2,1) - Sd(1,2))/2; // angular velocities vector [wx wy wz]
    w(1) = (Sd(0,2) - Sd(2,0))/2;
    w(2) = (Sd(1,0) - Sd(0,1))/2;

    v(0) = dp(0);
    v(1) = dp(1);
    v(2) = dp(2);
    v(3) = w(0);
    v(4) = w(1);
    v(5) = w(2);

    // Update pos and R
    posbef = pos;
    Rbef = R;

    // Compute PD signal
    poserr = posd-pos; // position error

    r0=R.col(0);
    r1=R.col(1);
    r2=R.col(2);
    rd0=Rd.col(0);
    rd1=Rd.col(1);
    rd2=Rd.col(2);

    orierr = 0.5*( r0.cross(rd0) + r1.cross(rd1) + r2.cross(rd2) ) ; // orientation
error

    perr(0) = poserr(0); // pose error
    perr(1) = poserr(1);
    perr(2) = poserr(2);
    perr(3) = orierr(0);
    perr(4) = orierr(1);
    perr(5) = orierr(2);
```

```
114
115     verr.resize(6);
116     verr = vd-v;
117
118     // Null space
119     Jinv.resize(7,6);
120     Jinv.fill(0.0);
121     Iden.resize(7,7);
122     Iden.setIdentity(7,7);
123
124     // Compute the Jacobian pseudoinverse
125     test.copyFrom(J);
126     Je.resize(6,7);
127     Je=test;
128     int nn=Je.rows(); //=6
129     int mm=Je.cols(); //=7
130     U.resize(nn,nn);
131     V.resize(mm,nn);
132     Ssvd.resize(mm,nn);
133     double maxim=Je.maxCoeff();
134     double minim=Je.minCoeff();
135
136     if (abs(maxim)<0.00000001 && abs(minim)<0.00000001){
137         U.setIdentity();
138         V.setIdentity();
139         Ssvd.fill(0.0);
140     }else{
141         //we use the transposed for the SVD because it does not support matrices with
more columns than rows
142         Eigen::JacobiSVD<Eigen::MatrixXd> svd(Je.transpose(),Eigen::ComputeFullU |
Eigen::ComputeFullV);
143         U=svd.matrixV();
144         V=svd.matrixU();
145         S0=svd.singularValues();
146
147         Ssvd.fill(0.0);
148         for (int i=0;i<nn;i++){
149             if (S0(i)<0.000000001){
150                 Ssvd(i,i)=0.0;
151             }else{
152                 Ssvd(i,i)=1/S0(i);
153             }
154         }
155     }
156
157     Eigen::MatrixXd UT = U.transpose();
158     Eigen::MatrixXd aux_mat =  Ssvd*UT;
159     Jinv=V*aux_mat;
160
161     // Control singularities
162
163     // Wrist singularity (jp6 = 0)
164     double lambda,jp6;
165     jp6=jp(5);
166     lambda=exp(-4.0*jp6*jp6); //lambda=[0,1]. Takes 1 if jp6=0
167     jv_type jv(this->jv_Input.getValue());
168     Eigen::VectorXd wristnull(7);
169     wristnull.fill(0.0);
170     wristnull(4)=-2.0*jv(4)*lambda;
171     wristnull(6)=-2.0*jv(6)*lambda;
```

```
172
173     // Elbow singularity (jp4 = 0)
174     double mu,jp4;
175     jp4=jp(3);
176     mu=exp(-4.0*jp4*jp4); //mu=[0,1]. Takes 1 if jp4=0
177     Eigen::VectorXd elbownull(7);
178     elbownull.fill(0.0);
179     elbownull(2)=-2.0*jv(2)*mu;
180     elbownull(4)=-2.0*jv(4)*mu;
181
182     Eigen::VectorXd nullvector(7);
183     nullvector=wristnull+elbownull;
184
185     // Compute controller (note that friction, gravity and coriolis forces are
    compensated somewhere else)
186     controlSignaleigen=(Je.transpose())*(Spgain*perr+Sdgain*verr)+(Iden-
    (Je.transpose())*(Jinv.transpose()))*Kref*nullvector;
187
188     // Limit torques assigned
189     for(int iaux2=0;iaux2<7;iaux2++){
190         if(controlSignaleigen(iaux2)>controlSignalLimit(iaux2)){
191             controlSignaleigen(iaux2)=controlSignalLimit(iaux2);
192         }else if(controlSignaleigen(iaux2)<-controlSignalLimit(iaux2)){
193             controlSignaleigen(iaux2)=-controlSignalLimit(iaux2);
194         }
195     }
196
197     // Joint limits rejection
198     double treshold;
199     treshold=0.001; //rad
200
     controlSignaleigen=WAM_kinematics.jointLimitsRejection(treshold,controlSignaleigen,j
    p);
201
202 }else{
203     controlSignaleigen.resize(7);
204     controlSignaleigen.fill(0.0);
205 }
206
207 // put into correct format
208 controlSignal=controlSignaleigen;
209 this->control_torque_value->setData(&controlSignal);
210 iteration_index++;
211
212 }
```

C.2.2 Guanys direccionals

```
// Convert gains from the velocity direction to the base frame
Spgain.resize(6,6);
Spgain.fill(0.0);
Sdgain.resize(6,6);
Sdgain.fill(0.0);

Eigen::Vector3d base1,base2,base3;
Eigen::Vector3d poserr_dir, dpd_dir,last_dpd_dir;
double norm_dpd,norm_poserr;

norm_dpd=dpd.norm();

if(norm_dpd<0.0001){
    base=last_base; // if desired velocity is small, takes the last directions

}else{
    base1=dpd;
    base2<<0.0,0.0,1.0;
    base3=base1.cross(base2);
    base2=base3.cross(base1);

    double norm_b3;
    norm_b3=base3.norm();

    if(norm_b3<0.001){
        base=last_base; // if desired velocity is vertical, takes the last directions
    }else{
        base1=base1/(base1.norm());
        base2=base2/(base2.norm());
        base3=base3/(base3.norm());

        base.col(0)=base1;
        base.col(1)=base2;
        base.col(2)=base3;

        last_base=base; //update last base
    }

}

Eigen::MatrixXd Kpdiag1(3,3);
Kpdiag1=DirGain.block(0,0,3,3); //store the desired gains

Spgain.block(0,0,3,3) = (base) * (Kpdiag1) * (base.transpose()); //we use the
transpose because base its ortonormal
Spgain.block(3,3,3,3) = DirGain.block(3,3,3,3);

for (int k=0;k<3;k++){
    Sdgain(k,k) = 2.0*pow(Spgain(k,k),0.5)*1.0; //Kd for position
}
for (int k=3;k<6;k++){
    Sdgain(k,k) = 2.0*pow(Spgain(k,k),0.5)*0.2; //Kd for orientation
}
```

C.2.3 Direcció de l'esfera

```
// Convert gains from the sphere directions to the base frame
Spgain.resize(6,6);
Spgain.fill(0.0);
Sdgain.resize(6,6);
Sdgain.fill(0.0);

Eigen::Vector3d base1,base2,base3;

// Compute distance
Eigen::Vector3d distance,center;
center<<0.0,0.0,0.45;
distance=center-pos;

// Compute the base
base1=distance;
base2<<0.0,0.0,1.0;
base3=base1.cross(base2);
base2=base3.cross(base1);

double norm_b3;
norm_b3=base3.norm();

if(norm_b3<0.001){
    base=last_base;
}else{
    base1=base1/(base1.norm());
    base2=base2/(base2.norm());
    base3=base3/(base3.norm());

    base.col(0)=base1;
    base.col(1)=base2;
    base.col(2)=base3;

    last_base=base; //update last base
}

// Kp rigid interpolation
double dmin,dmax,dmax2,Kpmin,Kpmax,Kpdura,Kptova;
double distance_norm,distance_norm_bef;

distance_norm=distance.norm();
dmin=0.20;
dmax=0.45; //this is to interpolate when the WAM is getting closer
dmax2=0.40; //this is to interpolate when the WAM is going away

Kpmax=1000.0;
Kpmin=300.0;
Kptova=200.0;

double getting_closer;
if(iteration_index==0){
    getting_closer = 1.0; //arbitrary value, just to indicate that it is getting
closer
}else{
    getting_closer = distance_norm_bef-distance_norm; //if positive, it is getting
near
}

if(getting_closer>-0.001){ //it is getting closer
    if(distance_norm<=dmin){
```

```
59          Kpdura=Kpmax;
60      }else if(distance_norm>=dmax){
61          Kpdura=Kpmin;
62      }else{
63          double pendent;
64          pendent = (distance_norm-dmin)/(dmax-dmin);
65          Kpdura = pendent*Kpmin + (1.0 - pendent)*Kpmax;
66      }
67  }else{ //it is going away
68      if(distance_norm<=dmin){
69          Kpdura=Kpmax;
70      }else if(distance_norm>=dmax2){
71          Kpdura=Kpmin;
72      }else{
73          double pendent;
74          pendent = (distance_norm-dmin)/(dmax2-dmin);
75          Kpdura = pendent*Kpmin + (1.0 - pendent)*Kpmax;
76      }
77
78  }
79
80  distance_norm_bef = distance_norm; //update
81
82  Eigen::VectorXd Kpdiag1(3);
83  Kpdiag1<<Kpdura,Kptova,Kptova;
84
85  Spgain.block(0,0,3,3) = (base) * (Kpdiag1.asDiagonal()) * (base.transpose());

86  Spgain.block(3,3,3,3) = DirGain.block(3,3,3,3);
87
88  for (int k=0;k<3;k++){
89      Sdgain(k,k) = 2.0*pow(Spgain(k,k),0.5)*1.0; //Kd for position
90  }
91  for (int k=3;k<6;k++){
92      Sdgain(k,k) = 2.0*pow(Spgain(k,k),0.5)*0.2; //Kd for orientation
93  }
```

C.2.4 Direcció del pla

```
// Convert gains from the plane directions to the base frame
Spgain.resize(6,6);
Spgain.fill(0.0);
Sdgain.resize(6,6);
Sdgain.fill(0.0);

// Compute distance
Eigen::VectorXd plane(4);
plane<<1.0,0.0,-1.0,1.2; //define the plane
double numerator,denominator;
numerator = plane(0)*pos(0) + plane(1)*pos(1) + plane(2)*pos(2) + plane(3);
denominator = pow( (plane(0)*plane(0) + plane(1)*plane(1) + plane(2)*plane(2)),0.5);
double distance_norm,distance_norm_bef;
distance_norm = (fabs(numerator)) / denominator; //distance point-plane

// Compute the base
Eigen::Vector3d base1,base2,base3;
base1(0)=plane(0); //the normal vector of the plane is the rigid direction
base1(1)=plane(1);
base1(2)=plane(2);
base2<<0.0,0.0,1.0;
base3=base1.cross(base2);
base2=base3.cross(base1);

double norm_b3;
norm_b3=base3.norm();

if(norm_b3<0.001){
    base=last_base;
}else{
    base1=base1/(base1.norm());
    base2=base2/(base2.norm());
    base3=base3/(base3.norm());

    base.col(0)=base1;
    base.col(1)=base2;
    base.col(2)=base3;

    last_base=base; //update last base
}

// Kp rigid interpolation
double dmin,dmax,dmax2,Kpmin,Kpmax,Kpdura,Kptova;
dmin=0.10;
dmax=0.35; //this is to interpolate when the WAM is getting closer
dmax2=0.30; //this is to interpolate when the WAM is going away

Kpmax=1000.0;
Kpmin=300.0;
Kptova=200.0;

double getting_closer;
if(iteration_index==0){
    getting_closer = 1.0; //arbitrary value, just to indicate that it is getting
closer
}else{
    getting_closer = distance_norm_bef-distance_norm; //if positive, it is getting
closer
}

```

```
59 if(getting_closer>-0.01){ //it is getting closer
60     if(distance_norm<=dmin){
61         Kpdura=Kpmax;
62     }else if(distance_norm>=dmax){
63         Kpdura=Kpmin;
64     }else{
65         double pendent;
66         pendent = (distance_norm-dmin)/(dmax-dmin);
67         Kpdura = pendent*Kpmin + (1.0 - pendent)*Kpmax;
68     }
69 
70 }else{ //it is going away
71     if(distance_norm<=dmin){
72         Kpdura=Kpmax;
73     }else if(distance_norm>=dmax2){
74         Kpdura=Kpmin;
75     }else{
76         double pendent;
77         pendent = (distance_norm-dmin)/(dmax2-dmin);
78         Kpdura = pendent*Kpmin + (1.0 - pendent)*Kpmax;
79     }
80 
81 }
82 
83 distance_norm_bef = distance_norm; //update
84 
85 Eigen::VectorXd Kpdiag1(3);
86 Kpdiag1<<Kpdura,Kptova,Kptova;
87 
88 Spgain.block(0,0,3,3) = (base) * (Kpdiag1.asDiagonal()) * (base.transpose());
89 Spgain.block(3,3,3,3) = DirGain.block(3,3,3,3);
90 
91 for (int k=0;k<3;k++){
92     Sdgain(k,k) = 2.0*pow(Spgain(k,k),0.5)*1.0; //Kd for position
93 }
94 for (int k=3;k<6;k++){
95     Sdgain(k,k) = 2.0*pow(Spgain(k,k),0.5)*0.2; //Kd for orientation
96 }
```

C.2.5 Controlador ProMP

```
// COMPUTE KP AND KD FROM PROMP COVARIANCE
// Take eigenvalues and eigenvectors of sigmainv
Eigen::MatrixXd posVEP(this->posVEP_Input.getValue());
Eigen::MatrixXd oriVEP(this->oriVEP_Input.getValue());
Eigen::MatrixXd posVAP(this->posVAP_Input.getValue());
Eigen::MatrixXd oriVAP(this->oriVAP_Input.getValue());

// Define rescaling factors
maxGainPos = 800.0;
minGainPos = 150.0;
vapmaxPos = 13.3; //ln(vapmax)
vapminPos = 6.9;

maxGainOri = 6.0;
minGainOri = 0.5;
vapmaxOri = 21.8; //ln(vapmax)
vapminOri = 4.6;

// Compute D matrix rescaled (position part)
DPos.resize(3,3);
DPos.fill(0.0);
DOri.resize(3,3);
DOri.fill(0.0);

for (int k=0;k<3;k++){
    if(posVAP(k,0)>1.5){ //protect ln()
        DPos(k,k) = minGainPos + (maxGainPos-minGainPos)*(std::log(posVAP(k,0))-
vapminPos)/(vapmaxPos-vapminPos);
    }else{
        DPos(k,k) = lastDPos(k,k);
    }
}
lastDPos = DPos;

// Compute D matrix rescaled (orientation part)
for (int k=0;k<3;k++){
    if(oriVAP(k,0)>1.5){ //protect ln()
        DOri(k,k) = minGainOri + (maxGainOri-minGainOri)*(std::log(oriVAP(k,0))-
vapminOri)/(vapmaxOri-vapminOri);
    }else{
        DOri(k,k) = lastDOri(k,k);
    }
}
lastDOri = DOri;

// Compute Kp and Kd
Spgain.resize(6,6);
Spgain.fill(0.0);
Sdgain.resize(6,6);
Sdgain.fill(0.0);

Spgain.block(0,0,3,3) = posVEP*DPos*(posVEP.transpose());
Spgain.block(3,3,3,3) = oriVEP*DOri*(oriVEP.transpose());

for (int k=0;k<3;k++){
    Sdgain(k,k) = 2.0*pow(Spgain(k,k),0.5)*1.0;
}
for (int k=3;k<6;k++){
    Sdgain(k,k) = 2.0*pow(Spgain(k,k),0.5)*0.1;
}
```

C.2.6 Controlador latent

```
virtual void operate(){

if(startflag==1){

    // Get Jacobian Matrix
    cp_type tcp(this->TCP.getValue());
    jacfound=bt_kinematics_eval_jacobian(this-
>kin_Input.getValue().impl,DOF,tcp.asGslType(),J);

    J_foo.copyFrom(J); // from gsl type to math matrix type
    Je.resize(6,7);
    Je=J_foo; // from math matrix to eigen matrix type

    // Get desired position and orientation
    jp_type pd(this->pd_Input.getValue()); // pd = [x,y,z,nu,ex,ey,ez]

    // Get current position and orientation
    jp_type jp(this->jp_Input.getValue());

    // Convert desired position to latent space
    Eigen::VectorXd q_temp(4), q_temp2;
    Eigen::MatrixXd R_temp;
    q_temp(0) = pd(3);
    q_temp(1) = pd(4);
    q_temp(2) = pd(5);
    q_temp(3) = pd(6);

    R_temp = ac_math::Quat2Rot(q_temp);
    q_temp2 = ac_math::Rot2QuatSigned(R_temp*R0_);

    ydes.resize(6);
    ydes(0) = pd(0);
    ydes(1) = pd(1);
    ydes(2) = pd(2);
    ydes(3) = q_temp2(1);
    ydes(4) = q_temp2(2);
    ydes(5) = q_temp2(3);

    // Convert current position to latent space
    T_aux=WAM_kinematics.forwardkinematics(jp);
    Ract = T_aux.block<3,3>(0,0); // current rotation matrix 3x3
    qAct = ac_math::Rot2QuatSigned(Ract*R0_);

    yact.resize(6);
    yact(0) = T_aux(0,3);
    yact(1) = T_aux(1,3);
    yact(2) = T_aux(2,3);
    yact(3) = qAct(1);
    yact(4) = qAct(2);
    yact(5) = qAct(3);

    // Compute latent variables
    xdes = omegainv_*ydes; //desired latent "position"
    xact = omegainv_*yact; //actual latent "position"

    // Compute end-effector desired velocity
    if (iteration_index==0){ //initial velocities=0
        vdes.resize(r); //r=dimension latent space
        vdes.fill(0.0);
        vact.resize(r);
```

```
60          vact.fill(0.0);
61      }else{
62          vdes = (xdes-xdes_bef)/0.02;
63          vact = (xact-xact_bef)/0.02;
64      }
65
66      // Update latent variables
67      xdes_bef = xdes;
68      xact_bef = xact;
69
70      // Compute gain matrices
71      Sdgain.resize(6,6);
72      Sdgain.fill(0.0);
73      for (int k=0;k<3;k++){
74          Sdgain(k,k) = 2.0*pow(Gain_(k,k),0.5)*1.0; //Kd for position
75      }
76      for (int k=3;k<6;k++){
77          Sdgain(k,k) = 2.0*pow(Gain_(k,k),0.5)*0.1; //Kd for orientation
78      }
79
80      // Convert gain matrices into latent space
81      Splatent = omegainv_*Gain_*omega_;
82      Sdlatent = omegainv_*Sdgain*omega_;
83
84      // Compute PD signal
85      xerr = xdes-xact; // position error
86      verr = vdes-vact; // velocity error
87      PD_latent = Splatent*xerr + Sdlatent*verr;
88
89      //Control null latent space
90      Eigen::MatrixXd Iden6(6,6), nullGain(6,6);
91      Eigen::VectorXd force_null(6), vecGain(6);
92
93      Iden6.setIdentity(6,6);
94      vecGain<<100.0,100.0,100.0,4.0,4.0,4.0;
95      nullGain.resize(6,6);
96      nullGain.fill(0.0);
97      nullGain = vecGain.asDiagonal();
98
99      force_null = nullGain*(ydes-yact);
100
101     // Compute controller (note that friction, gravity and coriolis forces are
compensated somewhere else)
102     controlSignaleigen = (Je.transpose()) * omega_ * PD_latent  + (Je.transpose()) *
(Iden6-omega_*omegainv_)*force_null;
103
104     // limit torques assigned
105     for(int iaux2=0;iaux2<7;iaux2++){
106         if(controlSignaleigen(iaux2)>controlSignalLimit(iaux2)){
107             controlSignaleigen(iaux2)=controlSignalLimit(iaux2);
108         }else if(controlSignaleigen(iaux2)<-controlSignalLimit(iaux2)){
109             controlSignaleigen(iaux2)=-controlSignalLimit(iaux2);
110         }
111     }
112
113     // joint limits rejection
114     double treshold;
115     treshold=0.001; //rad
116

 controlSignaleigen=WAM_kinematics.jointLimitsRejection(treshold,controlSignaleigen,
```

```
    jp);
117
118 }else{
119     controlSignaleigen.resize(7);
120     controlSignaleigen.fill(0.0);
121     ydes.resize(6);
122     ydes.fill(0.0);
123 }
124 // put into correct format
125 controlSignal=controlSignaleigen;
126
127 this->control_torque_value->setData(&controlSignal);
128
129 iteration_index++;
130 }
```

C.3 Rigidesa a partir de la covariància

C.3.1 Precalculat fora de temps real

```
int main(int argc, char** argv){
// Check for trajectory names
if(argv[1]==NULL ){
    std::cout << "Missing file name  [Trajectory]" << std::endl;
}
std::string traj_filename(argv[1]);

Eigen::MatrixXd mw,Sw,Om, R0, Sigma_t, Sigma_t_geo, Sout, Sout2;
Eigen::VectorXd C, Y0mean, y, y_quat;
int Nf,dof, isCartesian;
double D,T;
double relativetime;
Eigen::MatrixXd sigma_pos, sigma_ori, sigma_pos_inv, sigma_ori_inv;
Eigen::MatrixXd vepPos, vepOri;
Eigen::VectorXd vapPos, vapOri;

ac_promp::readProMPdata(dof, Om, R0, C, D, Sw, mw, T, Nf, Y0mean, traj_filename);

// Open the .txt file
std::ofstream WFile;
std::string WfileName(traj_filename);
WfileName += "PosVEP.txt";
WFile.open(WfileName.c_str());

std::ofstream WFile2;
std::string WfileName2(traj_filename);
WfileName2 += "OriVEP.txt";
WFile2.open(WfileName2.c_str());

std::ofstream WFile3;
std::string WfileName3(traj_filename);
WfileName3 += "PosVAP.txt";
WFile3.open(WfileName3.c_str());

std::ofstream WFile4;
std::string WfileName4(traj_filename);
WfileName4 += "OriVAP.txt";
WFile4.open(WfileName4.c_str());

for(int i=0;i<51;i++){
    relativetime=i/50.0;
    // Calculate sigma_t from pro-mp
    ac_promp::CalculateSigmat(y, Sigma_t, Nf, C, D, dof, mw, Sw, relativetime);
    Sigma_t_geo=ac_promp::sigma2geo(y, R0, Sigma_t); // convert to geometric sigma_t
(to use it in cartesian controller)

    // Extract position covariance and orientation covariance
    sigma_pos = Sigma_t_geo.block(0,0,3,3);
    sigma_ori = Sigma_t_geo.block(3,3,3,3);

    // Calculate pos and ori inverse
    sigma_pos_inv = sigma_pos.inverse();
    sigma_ori_inv = sigma_ori.inverse();

    // Calculate eigenvalues and eigenvectors
    Eigen::EigenSolver<Eigen::MatrixXd> esPos(sigma_pos_inv);
    vepPos = esPos.eigenvectors().real();
    vapPos = esPos.eigenvalues().real();
    Eigen::EigenSolver<Eigen::MatrixXd> esOri(sigma_ori_inv);
    vepOri = esOri.eigenvectors().real();
```

```
60      vapOri = esOri.eigenvalues().real();
61
62      // Put vep_pos in a row vector
63      Eigen::VectorXd fila;
64      Sout.resize(1,9);
65      for(int i=0;i<3;i++){
66          fila=vepPos.row(i);
67          Sout(0,i*3)=fila(0);
68          Sout(0,i*3+1)=fila(1);
69          Sout(0,i*3+2)=fila(2);
70      }
71      // Put vep_ori in a row vector
72      Eigen::VectorXd fila2;
73      Sout2.resize(1,9);
74      for(int i=0;i<3;i++){
75          fila2=vepOri.row(i);
76          Sout2(0,i*3)=fila2(0);
77          Sout2(0,i*3+1)=fila2(1);
78          Sout2(0,i*3+2)=fila2(2);
79      }
80      // Write vep_pos in a file
81      for (int i=0;i<Sout.rows();i++){
82          WFile<<Sout(i,0);
83          for(int j=1;j<Sout.cols();j++){
84              WFile<<","<<Sout(i,j);
85          }
86          WFile<<"\n";
87      }
88      // Write vep_ori in a file
89      for (int i=0;i<Sout2.rows();i++){
90          WFile2<<Sout2(i,0);
91          for(int j=1;j<Sout2.cols();j++){
92              WFile2<<","<<Sout2(i,j);
93          }
94          WFile2<<"\n";
95      }
96      // Write vap_pos in a file
97      for (int i=0;i<vapPos.rows();i++){
98          WFile3<<vapPos(i);
99          if(i<2){
100             WFile3<<",";
101         }else{
102             WFile3<<"\n";
103         }
104     }
105     // Write vap_ori in a file
106     for (int i=0;i<vapOri.rows();i++){
107         WFile4<<vapOri(i);
108         if(i<2){
109             WFile4<<",";
110         }else{
111             WFile4<<"\n";
112         }
113     }
114 }
115 WFile.close();
116 WFile2.close();
117 WFile3.close();
118 WFile4.close();
119 }
```

C.3.2 Càlcul el temps real

```
virtual void operate() {

if( startflag_==1 ) {
    local_time = this->timeValue.getValue();
    inter=floor(local_time/h);

    if(local_time <= final_time_){
        vep_pos_now.resize(9);
        vep_ori_now.resize(9);
        vep_pos_now = vep_pos_stored.col(inter);
        vep_ori_now = vep_ori_stored.col(inter);

        vap_pos_now.resize(3);
        vap_ori_now.resize(3);
        vap_pos_now = vap_pos_stored.col(inter);
        vap_ori_now = vap_ori_stored.col(inter);

        // Reconstruct veps as a 3x3 matrix
        vep_pos_out.resize(3,3);
        vep_ori_out.resize(3,3);
        for(int i=0;i<3;i++){
            fila_pos=vep_pos_now.segment(i*3,3);
            vep_pos_out.row(i)=fila_pos;
            fila_ori=vep_ori_now.segment(i*3,3);
            vep_ori_out.row(i)=fila_ori;
        }

        // Reconstruct vaps as a 3x1 matrix
        vap_pos_out.resize(3,1);
        vap_ori_out.resize(3,1);
        for(int i=0;i<3;i++){
            vap_pos_out(i,0)=vap_pos_now(i);
            vap_ori_out(i,0)=vap_ori_now(i);
        }

    }else{
        vep_pos_out.resize(3,3);
        vep_pos_out.setIdentity();
        vep_ori_out.resize(3,3);
        vep_ori_out.setIdentity();
        vap_pos_out.resize(3,1);
        vap_pos_out(0,0)=1.0;
        vap_pos_out(1,0)=1.0;
        vap_pos_out(2,0)=1.0;
        vap_ori_out.resize(3,1);
        vap_ori_out(0,0)=1.0;
        vap_ori_out(1,0)=1.0;
        vap_ori_out(2,0)=1.0;
    }
}

for(int i=0;i<3;i++){
    for(int j=0;j<3;j++){
        vep_pos_matrix(i,j)=vep_pos_out(i,j); //conversion from eigen to math::Matrix
        vep_ori_matrix(i,j)=vep_ori_out(i,j); //conversion from eigen to math::Matrix
    }
}
```

```
61 for(int i=0;i<3;i++){
62     vap_pos_matrix(i,0)=vap_pos_out(i,0); //conversion from eigen to math::Matrix
63     vap_ori_matrix(i,0)=vap_ori_out(i,0); //conversion from eigen to math::Matrix

64 }
65
66 for(int i=0;i<6;i++){
67     for(int j=0;j<6;j++){
68         Sout_zeros(i,j)=0.0;
69     }
70 }
71
72 this->VepPosOutput_value->setData(&vep_pos_matrix);
73 this->VepOriOutput_value->setData(&vep_ori_matrix);
74 this->VapPosOutput_value->setData(&vap_pos_matrix);
75 this->VapOriOutput_value->setData(&vap_ori_matrix);
76 this->SigmaZerosOutput_value->setData(&Sout_zeros);
77 iteration_index++;
78 }
```

C.4 Llibreries

C.4.1 Llibreria math

```
Eigen::VectorXd ac_math::Rot2QuatSigned(const Eigen::MatrixXd& R){
    double treshold;
    treshold = 0.0;

    double r11,r12,r13,r21,r22,r23,r31,r32,r33;
    r11 = R(0,0);
    r12 = R(0,1);
    r13 = R(0,2);
    r21 = R(1,0);
    r22 = R(1,1);
    r23 = R(1,2);
    r31 = R(2,0);
    r32 = R(2,1);
    r33 = R(2,2);

    double qw,qx,qy,qz,sign_qx,sign_qy,sign_qz;
    Eigen::VectorXd q_output(4);

    // Calculate qw
    if( (r11+r22+r33)>treshold ){
        qw = 0.5*std::sqrt(1.0+r11+r22+r33);
    }else{
        qw = 0.5*std::sqrt( (pow(r32-r23,2.0)+pow(r13-r31,2.0)+pow(r21-r12,2.0)) /
(3.0-r11-r22-r33) );
    }

    // Calculate qx
    if( (r11-r22-r33)>treshold ){
        qx = 0.5*std::sqrt(1.0+r11-r22-r33);
    }else{
        qx = 0.5*std::sqrt( (pow(r32-r23,2.0)+pow(r12+r21,2.0)+pow(r31+r13,2.0)) /
(3.0-r11+r22+r33) );
    }

    // Calculate qy
    if( (-r11+r22-r33)>treshold ){
        qy = 0.5*std::sqrt(1.0-r11+r22-r33);
    }else{
        qy = 0.5*std::sqrt( (pow(r13-r31,2.0)+pow(r12+r21,2.0)+pow(r23+r32,2.0)) /
(3.0+r11-r22+r33) );
    }

    // Calculate qz
    if( (-r11-r22+r33)>treshold ){
        qz = 0.5*std::sqrt(1.0-r11-r22+r33);
    }else{
        qz = 0.5*std::sqrt( (pow(r21-r12,2.0)+pow(r31+r13,2.0)+pow(r32+r23,2.0)) /
(3.0+r11+r22-r33) );
    }

    // Calculate signs
    sign_qx = ac_math::signe(r32-r23);
    sign_qy = ac_math::signe(r13-r31);
    sign_qz = ac_math::signe(r21-r12);

    q_output(0) = qw;
    q_output(1) = qx*sign_qx;
    q_output(2) = qy*sign_qy;
    q_output(3) = qz*sign_qz;
```

```
57      return q_output; //[IMPORTANT]: q = (qw, qx, qy, qz)
58  }
59
60  Eigen::MatrixXd ac_math::SampleQwVariance(Eigen::VectorXd y, Eigen::MatrixXd Sigma_t)
    {
61      Eigen::MatrixXd QwVariance(7,7);
62      int n=10; // number of samples
63      Eigen::VectorXd aux(3);
64      double lambda, qw;
65      Eigen::VectorXd y_rand(6), y_aux;
66      Eigen::VectorXd
    storepx(n),storepy(n),storepz(n),storeqx(n),storeqy(n),storeqz(n),storeqw(n);
67      double cov_px_qw,cov_py_qw,cov_pz_qw,cov_qx_qw,cov_qy_qw,cov_qz_qw,cov_qw_qw;
68
69      for(int i=0;i<n;i++){
70          y_aux.resize(6);
71          y_aux(0)=y(0);
72          y_aux(1)=y(1);
73          y_aux(2)=y(2);
74          y_aux(3)=y(3);
75          y_aux(4)=y(4);
76          y_aux(5)=y(5);
77          y_rand=multivariate_normalrandomsample(y_aux,Sigma_t);
78
79          aux(0)=y_rand(3);
80          aux(1)=y_rand(4);
81          aux(2)=y_rand(5);
82          lambda=aux.norm();
83
84          qw=std::cos(std::asin(std::max(std::min(lambda,1.0),-1.0)));
85
86          storepx(i)=y_rand(0);
87          storepy(i)=y_rand(1);
88          storepz(i)=y_rand(2);
89          storeqx(i)=y_rand(3);
90          storeqy(i)=y_rand(4);
91          storeqz(i)=y_rand(5);
92          storeqw(i)=qw;
93      }
94
95      // get the variances
96      cov_px_qw=covariance_test(storepx, storeqw);
97      cov_py_qw=covariance_test(storepy, storeqw);
98      cov_pz_qw=covariance_test(storepz, storeqw);
99      cov_qx_qw=covariance_test(storeqx, storeqw);
100     cov_qy_qw=covariance_test(storeqy, storeqw);
101     cov_qz_qw=covariance_test(storeqz, storeqw);
102     cov_qw_qw=covariance_test(storeqw, storeqw);
103
104     QwVariance.block(0,0,6,6)=Sigma_t;
105
106     QwVariance(0,6)=cov_px_qw;
107     QwVariance(6,0)=cov_px_qw;
108
109     QwVariance(1,6)=cov_py_qw;
110     QwVariance(6,1)=cov_py_qw;
111
112     QwVariance(6,2)=cov_pz_qw;
113     QwVariance(2,6)=cov_pz_qw;
114
```

```
115     QwVariance(6,3)=cov_qx_qw;
116     QwVariance(3,6)=cov_qx_qw;
117 
118     QwVariance(6,4)=cov_qy_qw;
119     QwVariance(4,6)=cov_qy_qw;
120 
121     QwVariance(6,5)=cov_qz_qw;
122     QwVariance(5,6)=cov_qz_qw;
123 
124     QwVariance(6,6)=cov_qw_qw;
125 
126     return QwVariance;
127 }
128 
129 Eigen::VectorXd ac_math::multivariate_normalrandomsample(Eigen::VectorXd means,
    Eigen::MatrixXd sigma){
130     int n=means.size(); // n=6
131     Eigen::VectorXd u(n);
132     Eigen::VectorXd vap(n),randomy(n),output(n);
133     Eigen::MatrixXd vep(n,n),vapmat(n,n);
134     bool Initialize;
135 
136     Initialize=true;
137     for(int i=0;i<n;i++){
138         u(i) = normalrandomsample(0.0,1.0,Initialize);
139         Initialize=false;
140     }
141 
142     Eigen::EigenSolver<Eigen::MatrixXd> es(sigma);
143     vep = es.eigenvectors().real();
144     vap = es.eigenvalues().real();
145 
146     vapmat.fill(0.0);
147     for(int j=0;j<n;j++){
148         if(vap(j)<0.0){
149             vapmat(j,j) = 0.0;
150         }else{
151             vapmat(j,j) = sqrt(vap(j));
152         }
153     }
154 
155     randomy = vep*vapmat*u;
156     output = means + randomy;
157 
158     return output;
159 }
160 
161 double ac_math::covariance_test(Eigen::VectorXd v1, Eigen::VectorXd v2){
162     int n=v1.size();
163     double mu1=v1.mean();
164     double mu2=v2.mean();
165     double covariance=0.0;
166 
167     for(int i=0;i<n;i++){
168         covariance=covariance + (v1(i)-mu1)*(v2(i)-mu2);
169     }
170 
171     covariance = covariance/((double)n-1.0);
172     return covariance;
173 }
```

C.4.2 Llibreria ProMP

```
Eigen::VectorXd ac_promp::y2quat(Eigen::VectorXd y, Eigen::MatrixXd R0){
    Eigen::VectorXd q_output,y_output(7),u(3),quat_aux;
    Eigen::MatrixXd R_aux;

    // convert to rotation matrix
    u(0)=y(3);
    u(1)=y(4);
    u(2)=y(5);
    quat_aux=ac_math::TriVec2Quat(u);
    R_aux=ac_math::Quat2Rot(quat_aux);

    // convert to quaternion
    q_output=ac_math::Rot2QuatSigned(R0*R_aux);
    y_output(0)=y(0);
    y_output(1)=y(1);
    y_output(2)=y(2);
    y_output(3)=q_output(0);
    y_output(4)=q_output(1);
    y_output(5)=q_output(2);
    y_output(6)=q_output(3);

    return y_output;
}

Eigen::MatrixXd ac_promp::sigma2geo(Eigen::VectorXd y, Eigen::MatrixXd R0,
Eigen::MatrixXd Sigma_t){
    Eigen::MatrixXd param2base(6,6), Sigma_t_basef(6,6), sigma_quat(7,7), Hquat(3,4),
squat2sgeo(6,7), Sigma_t_geo(6,6);

    // convert sigma_t(6,6) to sigma_quat(7,7) with qw-covariances estimated
    sigma_quat=ac_math::SampleQwVariance(y,Sigma_t);

    Eigen::VectorXd vec(3),quat_aux(4);
    vec(0)=y(3);
    vec(1)=y(4);
    vec(2)=y(5);
    quat_aux=ac_math::TriVec2Quat(vec); //reconstruct qw

    quat_aux = quat_aux/quat_aux.norm();

    // convert sigma_quat(7,7) to sigma_geo(6,6)
    Hquat(0,0)=-quat_aux(1);
    Hquat(1,0)=-quat_aux(2);
    Hquat(2,0)=-quat_aux(3);
    Hquat(0,1)=quat_aux(0);
    Hquat(1,1)=quat_aux(3);
    Hquat(2,1)=-quat_aux(2);
    Hquat(0,2)=-quat_aux(3);
    Hquat(1,2)=quat_aux(0);
    Hquat(2,2)=quat_aux(1);
    Hquat(0,3)=quat_aux(2);
    Hquat(1,3)=-quat_aux(1);
    Hquat(2,3)=quat_aux(0);

    squat2sgeo.fill(0.0);
    squat2sgeo(0,0)=1.0;
    squat2sgeo(1,1)=1.0;
    squat2sgeo(2,2)=1.0;
    squat2sgeo.block(3,3,3,4)=2.0*Hquat;

```

```
59      Sigma_t_geo = squat2sgeo * sigma_quat * squat2sgeo.transpose();
60
61      // rotate sigma from end-effector-frame to base-frame
62      param2base.fill(0.0);
63      param2base(0,0)=1.0;
64      param2base(1,1)=1.0;
65      param2base(2,2)=1.0;
66      param2base.block(3,3,3,3)=R0;
67      Sigma_t_basef=param2base *Sigma_t_geo * param2base.transpose();
68
69      return Sigma_t_basef;
70  }
71
72  void ac_promp::CalculateSigmat(Eigen::VectorXd& y, Eigen::MatrixXd& Sigma_t, int Nf,
    Eigen::VectorXd Centers, double Width, int d, Eigen::MatrixXd weights ,
    Eigen::MatrixXd Sigma_w, double t){
73      Eigen::VectorXd GT(Nf);
74
75      // compute kernels and assign to GT matrix
76      y.resize(d);
77      y.fill(0.0);
78      Eigen::MatrixXd y_aux;
79      Eigen::VectorXd D(Nf);
80      D.fill(1.0);
81      D=D*Width;
82      GetSmallPhi_t(t,Centers,D);
83
84      for(int s=0;s<Nf;s++){
85          GT(s)=ac_math::evalexp(t,Centers(s),Width);
86      }
87
88      Sigma_t=ac_promp::KronProdMat(Sigma_w,GT,d,Nf);
89
90      y_aux=ac_promp::KronProdLeft(weights,GT,d,Nf);
91
92      for(int j=0;j<d;j++){
93          y(j)=y_aux(j,0);
94      }
95  }
```

Bibliografia

[1] Soheil Sarabandi and Federico Thomas. Accurate computation of quaternions from rotation matrices. In Jadran Lenarcic and Vincenzo Parenti-Castelli, editors, *Advances in Robot Kinematics 2018*, pages 39–46, Cham, 2019. Springer International Publishing.

[2] A. Paraschos, C. Daniel, J. Peters, and G. Neumann. Using probabilistic movement primitives in robotics. *Autonomous Robots*, 42(3):529–551, March 2018.

[3] A. Colomé, G. Neumann, J. Peters, and C. Torras. Dimensionality reduction for probabilistic movement primitives. In *2014 IEEE-RAS International Conference on Humanoid Robots*, pages 794–800, Nov 2014.