hxt--MIT minicheetah控制部分源码解析

/*

说明：

1. 以下代码块均来源于MIT cheetah github上的源码，只添加注释并无修改。
2. 工程有点大，水平有限，尚有未理解的重要细节待补全，后续加上。
3. 此文档由个人结合论文和源码理解，主要还是想了解现在市场机器人步态的相关技术，供分享，有一定的学习参考价值，如有错误，望加评论指出。

*/

控制框架

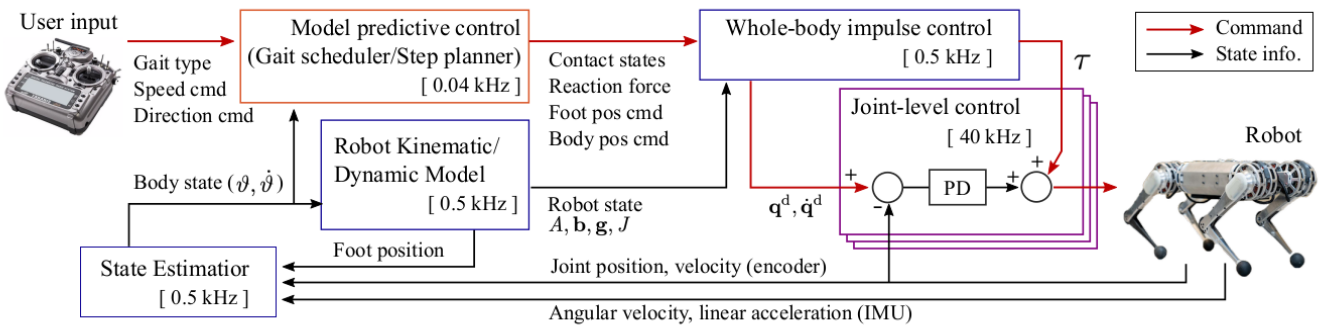

Fig. 2. **Overall Control Framework.** Using the user commanded gait type, speed, and direction from the RC-controller, the MPC computes desired reaction forces and foot/body position commands. From these, WBC computes joint torque, position, and velocity commands that are delivered to the each joint-level controller. Each component's update frequency is represented by the color of its box.

论文公式解释

动量守恒

$$m\ddot{p} = \Sigma_{i=1}^{n_c} f_i - mg$$

m 是机器人的质量

$\ddot{p}$ 是机器人的加速度

g 是重力加速度

角动量守恒

$$\frac{d}{dt}(I\omega) = \Sigma_{i=1}^{n_c} r_i \times f_i$$

I是转动惯量张量

ω 是角速度

r_i 是第i个接触点到质心的位移向量

f_i 是第i个接触点的反作用力

多体运动学

$$A \begin{pmatrix} \ddot{q}_f \\ \ddot{q}_j \end{pmatrix} + b + g = \begin{pmatrix} 0_6 \\ \tau \end{pmatrix} + J_c^\top f_r$$

A广义质量矩阵

b科里奥利力

g引力

τ 关节力矩

J_c 接触雅可比

f_r 反作用力

$\ddot{q}_f$ 浮基的加速度

$\ddot{q}_j$ 关节加速度的向量

0_6 表示6维0向量

MPC部分

MPC入门

由于力臂和方向动力学的叉积项，即使是简单的集中质量模型也不是完全线性的。为了实现凸MPC公式，我们采用了三种简化方法[8]。第一个假设是横滚和俯仰角很小。基于这个假设，我们可以将坐标变换简化如下。

$$\dot{\theta} \approx R_z(\psi)w$$
$$_GI \approx R_z(\psi)_BIR_Z(\psi)^\top$$

$\dot{\theta} = \begin{bmatrix}\dot{\phi} & \dot{\theta} & \dot{\psi}\end{bmatrix}$

yaw($\dot{\phi}$),pitch($\dot{\theta}$),roll($\dot{\psi}$)

R_z 是旋转矩阵将全局的w转换到局部(体)坐标系

$_GI$ 和 $_BI$ 分别是从全局和局部(体)坐标系的惯性张量。

假设俯仰和横滚速度很小并且惯性张量的非对角项也很小：

$$\frac{d}{dt}(I\omega) = I\dot{w} + w \times (Iw) \approx I\dot{w}$$

通过上述三个简化，离散动力学系统的可表示为:

$$x(k+1) = A_k x(k) + B_k \hat{f}(k) + \hat{g}$$

Where,

$$x = \begin{bmatrix}\theta^\top & p^\top & w^\top & \dot{p}^\top\end{bmatrix}^\top,$$
$$\hat{f} = \begin{bmatrix}f1 & \cdots & f_n\end{bmatrix}^\top, \quad (8)$$
$$\hat{g} = \begin{bmatrix}0_{1\times3} & 0_{1\times3} & 0_{1\times3} & g^\top\end{bmatrix}^\top,$$
$$A = \begin{bmatrix}1_{3\times3} & 0_{3\times3} & R_z(\psi_k)\triangle t & 0_{3\times3} \\ 0_{3\times3} & 1_{3\times3} & 0_{3\times3} & 1_{3\times3}\triangle t \\ 0_{3\times3} & 0_{3\times3} & 1_{3\times3} & 0_{3\times3} \\ 0_{3\times3} & 0_{3\times3} & 0_{3\times3} & 1_{3\times3}\end{bmatrix}$$
$$B = \begin{bmatrix}0_{3\times3} & \cdots & 0_{3\times3} \\ 0_{3\times3} & \cdots & 0_{3\times3} \\ {}_GI^{-1}[r_1]_\times\triangle t & \cdots & {}_GI^{-1}[r_n]_\times\triangle t \\ 1_{3\times3}\triangle t/m & \cdots & 1_{3\times3}\triangle t/m\end{bmatrix}$$

我们使用[8]中描述的公式来构造一个最小化的QP

$$\min_{\text{x,f}} \parallel x(x+1) - x^{ref}(k+1) \parallel_Q + \parallel f(k) \parallel_R$$

受动力学和初始条件约束。此外，摩擦锥由以下地面反作用力约束近似表示

$$\mid f_x \mid \leq \mu f_z, \mid f_y \mid \leq \mu f_z, f_z > 0.$$

我们用下面的公式来选择即将到来的脚步位置:

$$r_i^{cmd} = p_{shoulder,i} + p_{symmetry} + p_{centrifugal},$$

Where,

$$
\begin{aligned}
p_{shoudler,i} &= p_k + R_z(\psi_k) l_i, \\
p_{symmetry} &= \frac{t_{stance}}{2} v + k(v - v^{cmd}), \\
p_{centrifugal} &= \frac{1}{2}\sqrt{\frac{h}{g}} v \times w^{cmd}
\end{aligned}
$$

p_k 为第k个时间步的身体位置， l_i 为第i个腿肩相对于身体局部框架的位置。因此， $p_{shoulder,i}$ 是相对于全局框架的第i个肩部位置。对称也叫Raibert启发式[20]，如果机器人以命令速度行进，则迫使腿的着陆角度和离开角度相同。在我们的设置中，我们使用0.03作为反馈增益k。

优先任务执行

零空间（Null Space Projection，NSP）简介

NSB方法的主要做法是首先将任务行为分解成各个简单的任务子行为， 分别列出各个子行为的任务函数， 每个子行为的任务函数都能完成对应的行为， 然后再将行为区分一定的重量级， 将低优先级的任务向量向高优先级任务的零空间投影， 得到任务综合的输出函数， 传输给底层的运动控制器控制智能体的运动。

NSB方法的核心思想是将低优先级任务的任务向量向高优先级任务的零空间上投影， 在保证高优先级任务顺利完成的同时， 部分完成低优先级任务， 并且通过投影， 避免了任务之间的相互冲突。

1. NSB方法的冲突消解机制

NSB方法在同时协调多个任务输出时，是将低级任务向高级任务的零空间上投影，从而避免低级任务与高级任务之间的冲突，也就只是部分的执行低级任务，而当低级任务与高级任务完全冲突时，低级任务将不被执行； 或者高级任务的零空间为零，即高级任务矩阵满秩，低级任务也将不被执行。

另外，可以从低级任务和高级任务的雅可比矩阵上看出两个任务之间是否有 冲突，如果两个任务的 $J_{i+1}^{\dagger} J_i = 0$ 即说明两个任务向佩是正交的，那么这两个任务之间就没有冲突。 如果再加入第二个任务呢，判断方法也是一样的，即看这个任务向量与前两个任务向量是否正交，是，就说明不冲突； 否，就说明冲突。

2. NSB方法同时执行任务的个数

NSB方法同时可以执行任务的数量是一个要着重考虑的问题，对于一个n自由度的系统，定义优先任务的维度为 m_1 ，任务矩阵的秩为 γ_1 ，它的矩阵的零空间的维数就是 $n - \gamma_1$; 同样定义低级任务的维度为 m_2 ，任务矩阵的秩为 γ_2 。 如果加入的低级任务不与高级任务冲突， 那么它的矩阵的零空间的维数就是 $n - \gamma_1 - \gamma_2$ 。在后续任务不冲突的情况下，任务个数可以一直叠加到 $I \sum \gamma_i = n$ 。当任务的零空间维数叠加值等于自由度的值的时候，整个零空间已经被填满，这时再加入其它任何的任务都没有效果， 因为它在零空间上的任务投影将变为零。特别注意的是： 如果低优先级任务与高优先级冲突，那么低优先级任务向量向高优先级任务的投影在零空间上将为零，意味着低优先级的任务将不被执行， 这样保证了高优先级任务的执行。

3. NSB方法误差考量

NSB 方法在进行多智能体编队时，会产生一定的误差， 比如，通信延时会带来一定的误差，智能体的感知噪音也会带来误差， 这些误差的累积可能会导致得不到期望的编队效果。在NSB方法的实际运用时，应该根据具体的实物平台的特点和结构， 仔细分析误差的产生和来源，在合理的减小误差产生的基础上，加入合适的误差补偿佩，来使得NSB方法的机器人编队效果更佳。

这里利用**零空间投影技术**来执行优先级任务的方法，使得任务层次结构可以在计算上高效地执行。它的基本思路是，当**q**表示全局配置空间时，迭代规则如下：

1. **Δqi**是第一个任务的位移增量，受控于误差项和雅可比伪逆投影（式16）。
2. **q□_cmd** 是第i个任务的期望速度增量，受控于当前加速度的变化和雅可比伪逆投影（式17）。
3. **q□_cmd** 是第i个任务的期望加速度，基于反馈增益计算，包括比例和微分反馈控制（式18、22）。
4. **J_i_pre** 是任务的雅可比矩阵投影到先前任务的零空间中（式19），确保每个新任务不干扰优先级更高的任务。

文中还引入了两种伪逆计算方式，SVD分解和动态一致伪逆，后者用于计算加速度时的雅可比矩阵（式23）。

$$\triangle q_i = \triangle q_{i-1} + {J_{i|pre}}^{\dagger}(e_i - J_i \triangle q_{i-1})$$

$$\dot{q}_i^{cmd} = \dot{q}_{i-1}^{cmd} + {J_{i|pre}}^{\dagger}(\dot{x}_i^{des} - J_i \dot{q}_{i-1}^{cmd})$$

$$\ddot{q}_i^{cmd} = \ddot{q}_{i-1}^{cmd} + \overline{J_{i|pre}^{dyn}}(\ddot{x}_i^{cmd} - \dot{J}_i \dot{q} - J_i \ddot{q}_{i-1}^{cmd})$$

where,

$$J_{i|pre} = J_i N_{i-1},$$
$$N_{i-1} = N_0 N_{1|0} \cdots N_{i-1|i-2},$$
$$N_0 = I - J_c^{\dagger} J_c,$$
$$N_{i|i-1} = I - {J_{i|i-1}}^{\dagger} J_{i|i-1}$$

here, $i \geq 1$,and

$$\triangle q_0, \dot{q}_0^{cmd} = 0,$$
$$\ddot{q}_0^{cmd} = \overline{J_c^{dyn}}(-J_c \dot{q})$$

$$\ddot{x}_i^{cmd} = \ddot{x}^{des} + K_p(x_i^{des} - x_i) + K_d(\dot{x}^{des} - \dot{x})$$
$$\overline{J} = A^{-1} J^{\top}(J A^{-1} J^{\top})^{-1}$$

$$q_j^{cmd} = q_j + \triangle q_j$$

二次规划

通过QP求解器来计算最终的反作用力和加速度命令。QP求解器根据加速度命令和从MPC（模型预测控制）中得到的反作用力求解。

$$\underset{\delta_{\mathrm{fr}},\delta_{\mathrm{f}}}{min}(\delta_{fr}^{\top}Q_1\delta_{fr}+\delta_f^{\top}Q_2\delta_f)$$

目标是最小化与浮动基座的误差 δ_{fr} 和反作用力误差 δ_f 相关的加权误差函数。

s.t.

$$S_f(A\ddot{q}+b+g)=S_fJ_c^{\top}f_r$$

这个约束表达式中， S_f 是浮动基座选择矩阵， J_c 是接触雅可比矩阵， f_r 是反作用力。

$$\ddot{q}=\ddot{q}^{cmd}+\begin{bmatrix}\delta_f\\0_{nj}\end{bmatrix}$$

$$f_r=f_r^{MPC}+\delta_{fr}$$

$$Wf_r\geq 0$$

考虑到浮动基座加速度的松弛变量 δ_{fr} ，实际的基座加速度和模型预测控制 (MPC) 计算的反作用力可能有差异。因此，QP中的误差松弛变量允许在飞行阶段中保持基座的非控制状态。通过二次规划方法优化控制中的浮动基座加速度和反作用力。目标是在满足力和加速度约束的同时，最小化浮动基座的误差，从而提高系统的动态稳定性。

WBIC的最后一步是从反作用力fr和构型空间加速度 $\ddot{q}$ 中计算扭矩命令。将这两项代入式(3)，我们得到

$$\begin{bmatrix}\tau_f\\\tau_j\end{bmatrix}=A\ddot{q}+b+g-J_c^{\top}f_r$$

由于右边的所有项都是已知的，我们可以很容易地解出它并得到关节扭矩命令 τ_j 。

浮动基动力学流程

对应代码

```
template<typename T>
void WBC_Ctrl<T>::run(void* input, ControlFSMData<T> & data){
  ++_iter;

  // Update Model
  _UpdateModel(data._stateEstimator->getResult(), data._legController->datas);

  // Task & Contact Update
  _ContactTaskUpdate(input, data);

  // WBC Computation
  _ComputeWBC();

```

```
  // TEST
  //T dt(0.002);
  //for(size_t i(0); i<12; ++i){
    //_des_jpos[i] = _state.q[i] + _state.qd[i] * dt + 0.5 * _wbic_data->_qddot[i+6] * dt * dt;
    //_des_jvel[i] = _state.qd[i] + _wbic_data->_qddot[i+6]*dt;
  //}

  //_ContactTaskUpdateTEST(input, data);
  //_ComputeWBC();
  // END of TEST

  // Update Leg Command
  _UpdateLegCMD(data);

  // LCM publish
  _LCM_PublishData();
}
```

1. _UpdateModel(data._stateEstimator->getResult(), data._legController->datas);//模型更新
2. _ContactTaskUpdate(input, data);//任务更新
3. _ComputeWBC();//WBC解算
4. _UpdateLegCMD(data);//命令更新，下发到电机的命令更新
5. _LCM_PublishData();//数据发布，通过LCM进行数据发布，主要目的是为了查看机器的运行状态

以下对以上5个部分进行解读

模型更新_UpdateModel

输入参数：状态估计器的估计参数，腿部控制器参数

对应代码

```
template<typename T>
void WBC_Ctrl<T>::_UpdateModel(AttitudeData* attidata, LegData* legdata, LinearKFPositionVelocityEstimator<float> *posvelest){
  (void)legdata;
  _state.bodyOrientation << attidata->quat[0], attidata->quat[1], attidata->quat[2], attidata->quat[3];
  _state.bodyPosition = posvelest->_stateEstimatorData.result.position;
  for(size_t i(0); i<3; ++i){
    _state.bodyVelocity[i] = attidata->omega_body[i];
    _state.bodyVelocity[i+3] = posvelest->_stateEstimatorData.result.vBody[i];
```

```
 9
10      for(size_t leg(0); leg<4; ++leg){
11        if(leg > 1)
12        {
13          _state.q[3*leg + i] = (i == 0 ? -(legdata->q_abad[leg] + 1.5708):
    (i == 1 ? -legdata->q_hip[leg] : -legdata->q_knee[leg]));
14          _state.qd[3*leg + i] = (i == 0 ? -legdata->qd_abad[leg] : (i == 1 ?
    -legdata->qd_hip[leg] : -legdata->qd_knee[leg]));
15        }
16        else
17        {
18          _state.q[3*leg + i] = (i == 0 ? legdata->q_abad[leg] + 1.5708: (i
    == 1 ? legdata->q_hip[leg] : legdata->q_knee[leg]));
19          _state.qd[3*leg + i] = (i == 0 ? legdata->qd_abad[leg] : (i == 1 ?
    legdata->qd_hip[leg] : legdata->qd_knee[leg]));
20        }
21        _full_config[3*leg + i + 6] = _state.q[3*leg + i];
22      }
23    }
24
25    _model.setState(_state);
26    _model.contactJacobians();
27    _model.massMatrix();
28    _model.generalizedGravityForce();
29    _model.generalizedCoriolisForce();
30
31    _A = _model.getMassMatrix();
32    _grav = _model.getGravityForce();
33    _coriolis = _model.getCoriolisForce();
34    _Ainv = _A.inverse();
35  }
```

从状态估计器得到_state，包括身体的方位、位置、旋转角、速度以及各关节的角度和角速度，并将其传递给浮动基模型_model，然后对模型进行更新。

最终得到所需要的参数，_A（质量矩阵，也就是公式中的H），_grav（广义重力G），_coriolis（离心力，哥氏力等），_Ainv（_A的逆矩阵）。

这部分可以得到 _A、_grav、 _coriolis，再结合后面得到的关节，以及足底反力

就可以根据浮动基公式并结合Qp优化策略计算得到关节力了。

正运动学求解

以下图片介绍了正运动学的一个计算过程，来自书籍《Rigid Body Dynamics Algorithms》。

- $H(q) \in R^{n_q \times n_q}$ $--$广义质量矩阵
- $q, \dot{q}, \ddot{q}, \in R^{n_q}$ $--$广义位置、速度和加速度向量
- $C(q, \dot{q}) \in R^{n_q}$ $--$离心力
- $g(q) \in R^{n_q}$ $--$引力项
- $\tau \in R^{n_q}$ $--$外部广义力
- $F_c \in R^{n_q}$ $--$外部笛卡尔力
- $J_c(q) \in R^{n_q \times n_q}$ $--$外部广义
- S选择矩阵，过滤掉留个虚拟自由度

Algorithm 1 Forward Kinematics

Inputs: $\mathbf{q}, \dot{\mathbf{q}}$, model

Outputs: Body Jacobian ($\mathbf{J}_i$), Motion Transformation from origin (${}^i\mathbf{X}_0$), Body Velocity ($\mathbf{v}_i$), Body Bias Acceleration ($\mathbf{a}_i$), Body Velocity-Product Acceleration ($\mathbf{a}_{\mathrm{vp},i}$), Rotor Velocity ($\mathbf{v}_k$), Rotor Bias Acceleration ($\mathbf{a}_k$), Contact point orientation (${}^0\mathbf{X}_{\mathrm{c,rot}}$), Contact point position ($\mathbf{p}$), Contact Jacobian ($\mathbf{J}_c$), Contact point velocity ($\mathbf{v}_c$), Contact point bias acceleration ($\dot{\mathbf{J}}_{c,i}\dot{\mathbf{q}}$)

for $i = 1$ **to** N_b **do**
  $\mathbf{J}_i = \mathbf{0}$
  $\mathbf{X}_J = \mathrm{jcalc}_i(q_i)$
  $\mathbf{v}_J = \boldsymbol{\phi}_i \dot{\mathbf{q}}_i$
  ${}^i\mathbf{X}_{\lambda(i)} = \mathbf{X}_J \mathbf{X}_{T,i}$
  if $\lambda(i) = 0$ **then**
    ${}^i\mathbf{X}_0 = {}^i\mathbf{X}_{\lambda(i)}$ ▷ transform from origin
    $\mathbf{v}_i = \mathbf{v}_J$
    $\mathbf{a}_i = {}^i\mathbf{X}_{\lambda(i)} \mathbf{a}_{\mathrm{grav}}$ ▷ bias acceleration
    $\mathbf{a}_{vp,i} = \mathbf{0}$ ▷ velocity product acceleration
  else
    ${}^i\mathbf{X}_0 = {}^i\mathbf{X}_{\lambda(i)} {}^{\lambda(i)}\mathbf{X}_0$
    $\mathbf{v}_i = {}^i\mathbf{X}_{\lambda(i)} \mathbf{v}_{\lambda(i)} + \mathbf{v}_J$
    $\mathbf{a}_i = {}^i\mathbf{X}_{\lambda(i)} \mathbf{a}_{\lambda(i)} + \mathbf{v}_i \times \mathbf{v}_J$
    $\mathbf{a}_{vp,i} = {}^i\mathbf{X}_{\lambda(i)} \mathbf{a}_{vp,\lambda(i)} + \mathbf{v}_i \times \mathbf{v}_J$
for $k = 1$ **to** N_r **do**
  $i = \gamma(k)$
  $\mathbf{v}_J = N_k \boldsymbol{\phi}_k q_{\gamma(k)}$
  $\mathbf{v}_k = {}^k\mathbf{X}_{\mu(k)} \mathbf{v}_{\mu(k)} + \mathbf{v}_J$
  $\mathbf{a}_k = {}^k\mathbf{X}_{\mu(k)} \mathbf{a}_{\mu(k)} + \mathbf{v}_k \times \mathbf{v}_J$
for $i = 1$ **to** N_b **do**
  $j = i$
  ${}^i\mathbf{X}_j = \mathbf{1}$ ▷ $i = j$
  $\mathbf{J}_{i,i} = \boldsymbol{\phi}_i$ ▷ in body coordinates
  while $\lambda(j) > 0$ **do** ▷ Loop through i's ancestors
    ${}^i\mathbf{X}_j = {}^i\mathbf{X}_j {}^j\mathbf{X}_{\lambda(j)}$
    $j = \lambda(j)$
    $\mathbf{J}_{i,j} = {}^i\mathbf{X}_j \boldsymbol{\phi}_j$ ▷ in i's body coordinates
for $i = 1$ **to** N_c **do** ▷ Loop through contacts
  $j = \sigma(i)$
  $\mathbf{R} = \mathrm{MotionXform\_rotation}({}^j\mathbf{X}_0)$ ▷ Coordinate Rotation
  ${}^0\mathbf{X}_{\mathrm{rot}} = \mathrm{blkdiag}(\mathbf{R}^\top)$ ▷ Rotation from contact to origin coordinate system
  $\mathbf{p}_i = \mathrm{MotionXform\_translation}({}^{\mathrm{c}}\mathbf{X}_k {}^j\mathbf{X}_0)$ ▷ Contact Forward Kinematics
  $\mathbf{J}_{\mathrm{c,i}} = {}^0\mathbf{X}_{\mathrm{rot}} {}^{\mathrm{c}}\mathbf{X}_k \mathbf{J}_j$ ▷ Contact Jacobian (rotated)
  $\mathbf{v}_{\mathrm{c},i} = {}^0\mathbf{X}_{\mathrm{rot}} \mathbf{v}_j$ ▷ Contact Velocity (rotated)
  $\dot{\mathbf{J}}_{\mathrm{c},i}\dot{\mathbf{q}} = {}^0\mathbf{X}_{\mathrm{rot}} {}^{\mathrm{c}}\mathbf{X}_k \mathbf{a}_{vp,j} + \mathbf{v}^{\omega}_{\mathrm{c},i} \times \mathbf{v}^{v}_{\mathrm{c},i}$ ▷ Contact Bias Acceleration (rotated)

Algorithm 2 Add Rotors (Two-Step)

Inputs: $\mathbf{q}, \dot{\mathbf{q}}$, model
Outputs: $\Delta\mathbf{H}, \Delta\mathbf{C}$

for $k = 1$ **to** N_r **do**
  $i = \gamma(k)$
  $\Delta\mathbf{H}_{i,i} = \Delta\mathbf{H}_{i,i} + N_k^2 \mathbf{\Phi}_k^\top \mathbf{I}_k^{\text{rot}} \mathbf{\Phi}_k$ ▷ Gearing
  $\mathbf{b} = N_k \mathbf{I}_k^{\text{rot}} \mathbf{\Phi}_k$ ▷ Gearing
  $\mathbf{v}_J = N_k \mathbf{\Phi}_k q_{\gamma(k)}$
  $\mathbf{v}_k = {}^k\mathbf{X}_{\mu(k)} \mathbf{v}_{\mu(k)} + \mathbf{v}_J$
  $\mathbf{a}_k = {}^k\mathbf{X}_{\mu(k)} \mathbf{a}_{\mu(k)} + \mathbf{v}_k \times \mathbf{v}_J$
  $\mathbf{f}_k^r = \mathbf{I}_k \mathbf{a}_k + \mathbf{v}_k \times^* \mathbf{I}_k \mathbf{v}_k$
  $\Delta\mathbf{C}_{\gamma(k)} = N_k \mathbf{\Phi}_k^\top \mathbf{f}_k^r$ ▷ Gearing constraint, bias
  if $\mu(k) \neq 0$ **then**
    $\mathbf{f}_{\mu(k)} = \mathbf{f}_{\mu(k)} + {}^k\mathbf{X}_{\mu(k)}^\top \mathbf{f}_k^r$ ▷ non-geared $\mathbf{f}$
    $\mathbf{I}_{\mu(k)}^c = \mathbf{I}_{\mu(k)}^c + {}^k\mathbf{X}_{\mu(k)}^\top \mathbf{I}_k^{\text{rot}} {}^k\mathbf{X}_{\mu(k)}$ ▷ Initialize $\mathbf{I}^c$
  $j = i$
  while $\lambda(j) > 0$ **do** ▷ off-diagonal, gearing
    $\mathbf{b} = {}^j\mathbf{X}_{\lambda(j)} \mathbf{b}$
    $j = \lambda(j)$
    $\Delta\mathbf{H}_{i,j} = \Delta\mathbf{H}_{i,j} + \mathbf{\Phi}_j^\top \mathbf{b}$
    $\Delta\mathbf{H}_{j,i} = \Delta\mathbf{H}_{i,j}$
for $i = N_b$ **to** 1 **do**
  $\mathbf{I}_{\lambda(i)}^c = \mathbf{I}_{\lambda(i)}^c + {}^i\mathbf{X}_{\lambda(i)}^\top \mathbf{I}_i^c {}^i\mathbf{X}_{\lambda(i)}$
  $\Delta\mathbf{C}_i = \Delta\mathbf{C}_i + \mathbf{\Phi}_i \mathbf{f}_i$ ▷ RNEA for $\mathbf{f}$ change
  if $\lambda(i) \neq 0$ **then**
    $\mathbf{f}_{\lambda(i)} = \mathbf{f}_{\lambda(i)} + {}^i\mathbf{X}_{\lambda(i)} \mathbf{f}_i$
for $i = 1$ **to** N_b **do** ▷ CRBA change due to $\Delta\mathbf{I}^c$
  $\Delta\mathbf{H}_{i,i} = \mathbf{\Phi}_i^\top \mathbf{I}_i^c \mathbf{\Phi}_i$
  $\mathbf{b} = \mathbf{I}_i^c \mathbf{\Phi}_i$
  $j = i$
  while $\lambda(j) > 0$ **do**
    $\mathbf{b} = {}^j\mathbf{X}_{\lambda(j)}^\top \mathbf{b}$
    $j = \lambda(j)$
    $\Delta\mathbf{H}_{i,j} = \mathbf{\Phi}_j^\top \mathbf{b}$
    $\Delta\mathbf{H}_{j,i} = \Delta\mathbf{H}_{i,j}$

1. 四元数到旋转矩阵

$$R(n,\theta) = \begin{bmatrix} 1-2y^2-2z^2 & 2xy+2wz & 2xz-2wy \\ 2xy-2wz & 1-2x^2-2z^2 & 2yz+2wx \\ 2xz+2wy & 2yz-2wx & 1-2x^2-2y^2 \end{bmatrix}$$

```
1 /*!
```

```
 * Convert a quaternion to a rotation matrix.  This matrix represents a
 * coordinate transformation into the frame which has the orientation specified
 * by the quaternion
 */
template <typename T>
Mat3<typename T::Scalar> quaternionToRotationMatrix(
    const Eigen::MatrixBase<T>& q) {
  static_assert(T::ColsAtCompileTime == 1 && T::RowsAtCompileTime == 4,
                "Must have 4x1 quat");
  typename T::Scalar e0 = q(0);
  typename T::Scalar e1 = q(1);
  typename T::Scalar e2 = q(2);
  typename T::Scalar e3 = q(3);

  Mat3<typename T::Scalar> R;

  R << 1 - 2 * (e2 * e2 + e3 * e3), 2 * (e1 * e2 - e0 * e3),
      2 * (e1 * e3 + e0 * e2), 2 * (e1 * e2 + e0 * e3),
      1 - 2 * (e1 * e1 + e3 * e3), 2 * (e2 * e3 - e0 * e1),
      2 * (e1 * e3 - e0 * e2), 2 * (e2 * e3 + e0 * e1),
      1 - 2 * (e1 * e1 + e2 * e2);
  R.transposeInPlace();
  return R;
}
```

2. 坐标系转换

createSXform（）函数功能为了实现坐标系的转换：包括平移和旋转两部分

$$
{}^{A}X_{B} = \begin{bmatrix} I & 0 \\ r_{\times} & 0 \end{bmatrix} \begin{bmatrix} E^{T} & 0 \\ r_{\times} & E^{T} \end{bmatrix} = \begin{bmatrix} E^{T} & 0 \\ E^{T} r_{\times} & E^{T} \end{bmatrix}
$$

其中 $E = {}^{B}R_{A}$

实现代码，创建空间向量转换矩阵，含旋转和平移

```

/*!
 * Create spatial coordinate transformation from rotation and translation
 */
template <typename T, typename T2>
auto createSXform(const Eigen::MatrixBase<T>& R,
                  const Eigen::MatrixBase<T2>& r) {
  static_assert(T::ColsAtCompileTime == 3 && T::RowsAtCompileTime == 3,
                "Must have 3x3 matrix");
  static_assert(T2::ColsAtCompileTime == 1 && T2::RowsAtCompileTime == 3,
                "Must have 3x1 matrix");
```

```
  Mat6<typename T::Scalar> X = Mat6<typename T::Scalar>::Zero();
  X.template topLeftCorner<3, 3>() = R;
  X.template bottomRightCorner<3, 3>() = R;
  X.template bottomLeftCorner<3, 3>() = -R * vectorToSkewMat(r);
  return X;
}
```

3. 关节旋转变换

```
/*!
 * Compute joint transformation
 */
template <typename T>
Mat6<T> jointXform(JointType joint, CoordinateAxis axis, T q) {

  Mat6<T> X = Mat6<T>::Zero();
  if (joint == JointType::Revolute) {
    X = spatialRotation(axis, q);
  } else if (joint == JointType::Prismatic) {
    Vec3<T> v(0, 0, 0);
    if (axis == CoordinateAxis::X)
      v(0) = q;
    else if (axis == CoordinateAxis::Y)
      v(1) = q;
    else if (axis == CoordinateAxis::Z)
      v(2) = q;

    X = createSXform(RotMat<T>::Identity(), v);
  } else {
    throw std::runtime_error("Unknown joint xform\n");
  }
  return X;
}

```

4. 平移+旋转变换

$${}^{i}X_{\lambda} = X_J(i)X_T(i)$$

```
_Xup[i] = XJ * _Xtree[i];//旋转变换×移动变换
```

5. 运动子空间

子空间针对关节坐标系的，比如knee关节的旋转轴是Y轴，因此需要将关节角速度进行转换，程序中_S[i]为6×1矩阵，也就是空间速度的向量，因为只有角速度，没有线速度，因此矩阵后3项都为0.

```
1   _S[i] = jointMotionSubspace<T>(_jointTypes[i], _jointAxes[i]);//运动子空间
2      SVec<T> vJ = _S[i] * _state.qd[i - 6];//子空间×角速度
```

关节运动子空间：

```
1  /*!
2   * Compute joint motion subspace vector
3   */
4  template <typename T>
5  SVec<T> jointMotionSubspace(JointType joint, CoordinateAxis axis) {
6    Vec3<T> v(0, 0, 0);
7    SVec<T> phi = SVec<T>::Zero();
8    if (axis == CoordinateAxis::X)
9      v(0) = 1;
10   else if (axis == CoordinateAxis::Y)
11     v(1) = 1;
12   else
13     v(2) = 1;
14
15   if (joint == JointType::Prismatic)
16     phi.template bottomLeftCorner<3, 1>() = v;
17   else if (joint == JointType::Revolute)
18     phi.template topLeftCorner<3, 1>() = v;
19   else
20     throw std::runtime_error("Unknown motion subspace");
21
22   return phi;
23 }
```

6. 速度计算：

Calculate $\boldsymbol{C}$ and $\boldsymbol{p}_0^{\mathrm{c}}$:

$\boldsymbol{a}_0^{vp} = -{}^0\boldsymbol{a}_g$
for $i = 1$ **to** N_B **do**
  $[\boldsymbol{X}_{\mathrm{J}}, \boldsymbol{S}_i, \boldsymbol{v}_{\mathrm{J}}, \boldsymbol{c}_{\mathrm{J}}] = \mathrm{jcalc}(\mathrm{jtype}(i), \boldsymbol{q}_i, \dot{\boldsymbol{q}}_i)$
  ${}^i\boldsymbol{X}_{\lambda(i)} = \boldsymbol{X}_{\mathrm{J}}\, \boldsymbol{X}_{\mathrm{T}}(i)$
  if $\lambda(i) \neq 0$ **then**
    ${}^i\boldsymbol{X}_0 = {}^i\boldsymbol{X}_{\lambda(i)}\, {}^{\lambda(i)}\boldsymbol{X}_0$
  end
  $\boldsymbol{v}_i = {}^i\boldsymbol{X}_{\lambda(i)}\, \boldsymbol{v}_{\lambda(i)} + \boldsymbol{v}_{\mathrm{J}}$
  $\boldsymbol{a}_i^{vp} = {}^i\boldsymbol{X}_{\lambda(i)}\, \boldsymbol{a}_{\lambda(i)}^{vp} + \boldsymbol{c}_{\mathrm{J}} + \boldsymbol{v}_i \times \boldsymbol{v}_{\mathrm{J}}$
  $\boldsymbol{f}_i = \boldsymbol{I}_i\, \boldsymbol{a}_i^{vp} + \boldsymbol{v}_i \times^* \boldsymbol{I}_i\, \boldsymbol{v}_i - {}^i\boldsymbol{X}_0^*\, {}^0\boldsymbol{f}_i^x$
end
$\boldsymbol{f}_0 = \boldsymbol{I}_0\, \boldsymbol{a}_0^{vp} + \boldsymbol{v}_0 \times^* \boldsymbol{I}_0\, \boldsymbol{v}_0 - {}^0\boldsymbol{f}_0^x$
for $i = N_B$ **to** 1 **do**
  $\boldsymbol{C}_i = \boldsymbol{S}_i^{\mathrm{T}} \boldsymbol{f}_i$
  $\boldsymbol{f}_{\lambda(i)} = \boldsymbol{f}_{\lambda(i)} + {}^{\lambda(i)}\boldsymbol{X}_i^*\, \boldsymbol{f}_i$
end
$\boldsymbol{p}_0^{\mathrm{c}} = \boldsymbol{f}_0$

Calculate $\boldsymbol{H}$, $\boldsymbol{F}$ and $\boldsymbol{I}_0^{\mathrm{c}}$:

$\boldsymbol{H} = \boldsymbol{0}$
for $i = 0$ **to** N_B **do**
  $\boldsymbol{I}_i^{\mathrm{c}} = \boldsymbol{I}_i$
end
for $i = N_B$ **to** 1 **do**
  $\boldsymbol{I}_{\lambda(i)}^{\mathrm{c}} = \boldsymbol{I}_{\lambda(i)}^{\mathrm{c}} + {}^{\lambda(i)}\boldsymbol{X}_i^*\, \boldsymbol{I}_i^{\mathrm{c}}\, {}^i\boldsymbol{X}_{\lambda(i)}$
  $\boldsymbol{F}_i = \boldsymbol{I}_i^{\mathrm{c}}\, \boldsymbol{S}_i$
  $\boldsymbol{H}_{ii} = \boldsymbol{S}_i^{\mathrm{T}} \boldsymbol{F}_i$
  $j = i$
  while $\lambda(j) \neq 0$ **do**
    $\boldsymbol{F}_i = {}^{\lambda(j)}\boldsymbol{X}_j^*\, \boldsymbol{F}_i$
    $j = \lambda(j)$
    $\boldsymbol{H}_{ij} = \boldsymbol{F}_i^{\mathrm{T}} \boldsymbol{S}_j$
    $\boldsymbol{H}_{ji} = \boldsymbol{H}_{ij}^{\mathrm{T}}$
  end
  $\boldsymbol{F}_i = {}^0\boldsymbol{X}_j^*\, \boldsymbol{F}_i$
end

```
1   // total velocity of body i连杆
2     _v[i] = _Xup[i] * _v[_parents[i]] + vJ;//vi=vparent+si×qd
```

7. 科氏力计算

科氏力由于质点不仅作圆周运动，而且也做径向运动或轴向运动所产生的。

```
1   // Coriolis accelerations
2     _c[i] = motionCrossProduct(_v[i], vJ);//科里奥利线加速度=2v外积w
3     h_crot[i] = motionCrossProduct(_vrot[i], vJrot);//科里奥利角加速度
```

motionCrossProduct,用于计算运动叉乘积，科里奥利力介绍

$$
\begin{aligned}
mv_1 &= a_2 \cdot b_3 - a_3 \cdot b_2 \\
mv_2 &= a_3 \cdot b_1 - a_1 \cdot b_3 \\
mv_3 &= a_1 \cdot b_2 - a_2 \cdot b_1 \\
mv_4 &= a_2 \cdot b_6 - a_3 \cdot b_5 + a_5 \cdot b_3 - a_6 \cdot b_2 \\
mv_5 &= a_3 \cdot b_4 - a_1 \cdot b_6 - a_4 \cdot b_3 + a_6 \cdot b_1 \\
mv_6 &= a_1 \cdot b_5 - a_2 \cdot b_4 + a_4 \cdot b_2 - a_5 \cdot b_1
\end{aligned}
$$

```cpp
/*!
 * Compute spatial motion cross product.  Faster than the matrix multiplication
 * version
 */
template <typename T>
auto motionCrossProduct(const Eigen::MatrixBase<T>& a,
                        const Eigen::MatrixBase<T>& b) {
  static_assert(T::ColsAtCompileTime == 1 && T::RowsAtCompileTime == 6,
                "Must have 6x1 vector");
  SVec<typename T::Scalar> mv;
  mv << a(1) * b(2) - a(2) * b(1), a(2) * b(0) - a(0) * b(2),
      a(0) * b(1) - a(1) * b(0),
      a(1) * b(5) - a(2) * b(4) + a(4) * b(2) - a(5) * b(1),
      a(2) * b(3) - a(0) * b(5) - a(3) * b(2) + a(5) * b(0),
      a(0) * b(4) - a(1) * b(3) + a(3) * b(1) - a(4) * b(0);
  return mv;
}
```

计算质心加速度

Calculate $\boldsymbol{C}$ and $\boldsymbol{p}_0^c$:

$$
\begin{aligned}
&\boldsymbol{a}_0^{vp} = -{}^0\boldsymbol{a}_g \\
&\textbf{for } i = 1 \textbf{ to } N_B \textbf{ do} \\
&\quad [\boldsymbol{X}_\mathrm{J}, \boldsymbol{S}_i, \boldsymbol{v}_\mathrm{J}, \boldsymbol{c}_\mathrm{J}] = \mathrm{jcalc}(\mathrm{jtype}(i), \boldsymbol{q}_i, \dot{\boldsymbol{q}}_i) \\
&\quad {}^i\boldsymbol{X}_{\lambda(i)} = \boldsymbol{X}_\mathrm{J}\, \boldsymbol{X}_\mathrm{T}(i) \\
&\quad \textbf{if } \lambda(i) \neq 0 \textbf{ then} \\
&\quad\quad {}^i\boldsymbol{X}_0 = {}^i\boldsymbol{X}_{\lambda(i)}\, {}^{\lambda(i)}\boldsymbol{X}_0 \\
&\quad \textbf{end} \\
&\quad \boldsymbol{v}_i = {}^i\boldsymbol{X}_{\lambda(i)}\, \boldsymbol{v}_{\lambda(i)} + \boldsymbol{v}_\mathrm{J} \\
&\quad \boldsymbol{a}_i^{vp} = {}^i\boldsymbol{X}_{\lambda(i)}\, \boldsymbol{a}_{\lambda(i)}^{vp} + \boldsymbol{c}_\mathrm{J} + \boldsymbol{v}_i \times \boldsymbol{v}_\mathrm{J} \\
&\quad \boldsymbol{f}_i = \boldsymbol{I}_i\, \boldsymbol{a}_i^{vp} + \boldsymbol{v}_i \times^* \boldsymbol{I}_i\, \boldsymbol{v}_i - {}^i\boldsymbol{X}_0^*\, {}^0\boldsymbol{f}_i^x \\
&\textbf{end} \\
&\boldsymbol{f}_0 = \boldsymbol{I}_0\, \boldsymbol{a}_0^{vp} + \boldsymbol{v}_0 \times^* \boldsymbol{I}_0\, \boldsymbol{v}_0 - {}^0\boldsymbol{f}_0^x \\
&\textbf{for } i = N_B \textbf{ to } 1 \textbf{ do} \\
&\quad \boldsymbol{C}_i = \boldsymbol{S}_i^\mathrm{T} \boldsymbol{f}_i \\
&\quad \boldsymbol{f}_{\lambda(i)} = \boldsymbol{f}_{\lambda(i)} + {}^{\lambda(i)}\boldsymbol{X}_i^*\, \boldsymbol{f}_i \\
&\textbf{end} \\
&\boldsymbol{p}_0^c = \boldsymbol{f}_0
\end{aligned}
$$

Calculate $\boldsymbol{H}$, $\boldsymbol{F}$ and $\boldsymbol{I}_0^c$:

$$
\begin{aligned}
&\boldsymbol{H} = \boldsymbol{0} \\
&\textbf{for } i = 0 \textbf{ to } N_B \textbf{ do} \\
&\quad \boldsymbol{I}_i^c = \boldsymbol{I}_i \\
&\textbf{end} \\
&\textbf{for } i = N_B \textbf{ to } 1 \textbf{ do} \\
&\quad \boldsymbol{I}_{\lambda(i)}^c = \boldsymbol{I}_{\lambda(i)}^c + {}^{\lambda(i)}\boldsymbol{X}_i^*\, \boldsymbol{I}_i^c\, {}^i\boldsymbol{X}_{\lambda(i)} \\
&\quad \boldsymbol{F}_i = \boldsymbol{I}_i^c\, \boldsymbol{S}_i \\
&\quad \boldsymbol{H}_{ii} = \boldsymbol{S}_i^\mathrm{T} \boldsymbol{F}_i \\
&\quad j = i \\
&\quad \textbf{while } \lambda(j) \neq 0 \textbf{ do} \\
&\quad\quad \boldsymbol{F}_i = {}^{\lambda(j)}\boldsymbol{X}_j^*\, \boldsymbol{F}_i \\
&\quad\quad j = \lambda(j) \\
&\quad\quad \boldsymbol{H}_{ij} = \boldsymbol{F}_i^\mathrm{T} \boldsymbol{S}_j \\
&\quad\quad \boldsymbol{H}_{ji} = \boldsymbol{H}_{ij}^\mathrm{T} \\
&\quad \textbf{end} \\
&\quad \boldsymbol{F}_i = {}^0\boldsymbol{X}_j^*\, \boldsymbol{F}_i \\
&\textbf{end}
\end{aligned}
$$

```

/*!
 * Forward kinematics of all bodies.  Computes _Xup (from up the tree) and _Xa
 *(from absolute) Also computes _S (motion subspace), _v (spatial velocity in
 *link coordinates), and _c (coriolis acceleration in link coordinates)
 */
template <typename T>
void FloatingBaseModel<T>::forwardKinematics() {
  if (_kinematicsUpToDate) return;

  // calculate joint transformations
  _Xup[5] = createSXform(quaternionToRotationMatrix(_state.bodyOrientation),
                         _state.bodyPosition);
  _v[5] = _state.bodyVelocity;
  for (size_t i = 6; i < _nDof; i++) {
    // joint xform
    Mat6<T> XJ = jointXform(_jointTypes[i], _jointAxes[i], _state.q[i - 6]);
    _Xup[i] = XJ * _Xtree[i];
    _S[i] = jointMotionSubspace<T>(_jointTypes[i], _jointAxes[i]);
    SVec<T> vJ = _S[i] * _state.qd[i - 6];
    // total velocity of body i
    _v[i] = _Xup[i] * _v[_parents[i]] + vJ;
```

```
23
24      // Same for rotors
25      Mat6<T> XJrot = jointXform(_jointTypes[i], _jointAxes[i],
26                                 _state.q[i - 6] * _gearRatios[i]);
27      _Srot[i] = _S[i] * _gearRatios[i];
28      SVec<T> vJrot = _Srot[i] * _state.qd[i - 6];
29      _Xuprot[i] = XJrot * _Xrot[i];
30      _vrot[i] = _Xuprot[i] * _v[_parents[i]] + vJrot;
31
32      // Coriolis accelerations
33      _c[i] = motionCrossProduct(_v[i], vJ);
34      _crot[i] = motionCrossProduct(_vrot[i], vJrot);
35    }
36
37    // calculate from absolute transformations
38    for (size_t i = 5; i < _nDof; i++) {
39      if (_parents[i] == 0) {
40        _Xa[i] = _Xup[i];  // float base
41      } else {
42        _Xa[i] = _Xup[i] * _Xa[_parents[i]];
43      }
44    }
45
46    // ground contact points
47    //  // TODO : we end up inverting the same Xa a few times (like for the 8
48    //  points on the body). this isn't super efficient.
49    for (size_t j = 0; j < _nGroundContact; j++) {
50      if (!_compute_contact_info[j]) continue;
51      size_t i = _gcParent.at(j);
52      Mat6<T> Xai = invertSXform(_Xa[i]);  // from link to absolute
53      SVec<T> vSpatial = Xai * _v[i];
54
55      // foot position in world
56      _pGC.at(j) = sXFormPoint(Xai, _gcLocation.at(j));
57      _vGC.at(j) = spatialToLinearVelocity(vSpatial, _pGC.at(j));
58    }
59    _kinematicsUpToDate = true;
60  }
```

各任务的雅可比J的计算

雅可比矩阵3x18，其实应该是6×18，由于全身控制任务仅考虑对足底的平动，因此就舍掉了转动的影响

```
1     // Skip it if we don't care about it
```

```
    if (!_compute_contact_info[k]) continue;

    size_t i = _gcParent.at(k);

    // Rotation to absolute coords
    Mat3<T> Rai = _Xa[i].template block<3, 3>(0, 0).transpose();
    Mat6<T> Xc = createSXform(Rai, _gcLocation.at(k));

    // Bias acceleration
    SVec<T> ac = Xc * _avp[i];
    SVec<T> vc = Xc * _v[i];

    // Correct to classical
    _Jcdqd[k] = spatialToLinearAcceleration(ac, vc);

    // rows for linear velcoity in the world
    D3Mat<T> Xout = Xc.template bottomRows<3>();

    // from tips to base
    while (i > 5) {
      _Jc[k].col(i) = Xout * _S[i];
      Xout = Xout * _Xup[i];
      i = _parents[i];
    }
    _Jc[k].template leftCols<6>() = Xout;
  }
```

其中spatialToLinearAcceleration（）函数功能是将空间速度转变为经典的速度值：

公式为： $a = a_{line} + w \times v$,也即线加速度和角速度-速度产生的加速度两部分之和。

```
/*!
 * Compute the classical lienear accleeration of a frame given its spatial
 * acceleration and velocity
 */
template <typename T, typename T2>
auto spatialToLinearAcceleration(const Eigen::MatrixBase<T>& a,
                                 const Eigen::MatrixBase<T2>& v) {
  static_assert(T::ColsAtCompileTime == 1 && T::RowsAtCompileTime == 6,
                "Must have 6x1 vector");
  static_assert(T2::ColsAtCompileTime == 1 && T2::RowsAtCompileTime == 6,
                "Must have 6x1 vector");

  Vec3<typename T::Scalar> acc;
  // classical accleration = spatial linear acc + omega x v
```

```
15    acc = a.template tail<3>() + v.template head<3>().cross(v.template tail<3>
    ());
16    return acc;
17  }
```

计算质量矩阵

Calculate $\boldsymbol{C}$ and $\boldsymbol{p}_0^{\mathrm{c}}$:

$$
\begin{aligned}
&\boldsymbol{a}_0^{vp} = -{}^0\boldsymbol{a}_g \\
&\textbf{for } i = 1 \textbf{ to } N_B \textbf{ do} \\
&\quad [\boldsymbol{X}_{\mathrm{J}}, \boldsymbol{S}_i, \boldsymbol{v}_{\mathrm{J}}, \boldsymbol{c}_{\mathrm{J}}] = \mathrm{jcalc}(\mathrm{jtype}(i), \boldsymbol{q}_i, \dot{\boldsymbol{q}}_i) \\
&\quad {}^i\boldsymbol{X}_{\lambda(i)} = \boldsymbol{X}_{\mathrm{J}}\, \boldsymbol{X}_{\mathrm{T}}(i) \\
&\quad \textbf{if } \lambda(i) \neq 0 \textbf{ then} \\
&\quad\quad {}^i\boldsymbol{X}_0 = {}^i\boldsymbol{X}_{\lambda(i)}\, {}^{\lambda(i)}\boldsymbol{X}_0 \\
&\quad \textbf{end} \\
&\quad \boldsymbol{v}_i = {}^i\boldsymbol{X}_{\lambda(i)}\, \boldsymbol{v}_{\lambda(i)} + \boldsymbol{v}_{\mathrm{J}} \\
&\quad \boldsymbol{a}_i^{vp} = {}^i\boldsymbol{X}_{\lambda(i)}\, \boldsymbol{a}_{\lambda(i)}^{vp} + \boldsymbol{c}_{\mathrm{J}} + \boldsymbol{v}_i \times \boldsymbol{v}_{\mathrm{J}} \\
&\quad \boldsymbol{f}_i = \boldsymbol{I}_i\, \boldsymbol{a}_i^{vp} + \boldsymbol{v}_i \times^* \boldsymbol{I}_i\, \boldsymbol{v}_i - {}^i\boldsymbol{X}_0^*\, {}^0\boldsymbol{f}_i^x \\
&\textbf{end} \\
&\boldsymbol{f}_0 = \boldsymbol{I}_0\, \boldsymbol{a}_0^{vp} + \boldsymbol{v}_0 \times^* \boldsymbol{I}_0\, \boldsymbol{v}_0 - {}^0\boldsymbol{f}_0^x \\
&\textbf{for } i = N_B \textbf{ to } 1 \textbf{ do} \\
&\quad \boldsymbol{C}_i = \boldsymbol{S}_i^{\mathrm{T}} \boldsymbol{f}_i \\
&\quad \boldsymbol{f}_{\lambda(i)} = \boldsymbol{f}_{\lambda(i)} + {}^{\lambda(i)}\boldsymbol{X}_i^*\, \boldsymbol{f}_i \\
&\textbf{end} \\
&\boldsymbol{p}_0^{\mathrm{c}} = \boldsymbol{f}_0
\end{aligned}
$$

Calculate $\boldsymbol{H}$, $\boldsymbol{F}$ and $\boldsymbol{I}_0^{\mathrm{c}}$:

$$
\begin{aligned}
&\boldsymbol{H} = \boldsymbol{0} \\
&\textbf{for } i = 0 \textbf{ to } N_B \textbf{ do} \\
&\quad \boldsymbol{I}_i^{\mathrm{c}} = \boldsymbol{I}_i \\
&\textbf{end} \\
&\textbf{for } i = N_B \textbf{ to } 1 \textbf{ do} \\
&\quad \boldsymbol{I}_{\lambda(i)}^{\mathrm{c}} = \boldsymbol{I}_{\lambda(i)}^{\mathrm{c}} + {}^{\lambda(i)}\boldsymbol{X}_i^*\, \boldsymbol{I}_i^{\mathrm{c}}\, {}^i\boldsymbol{X}_{\lambda(i)} \\
&\quad \boldsymbol{F}_i = \boldsymbol{I}_i^{\mathrm{c}}\, \boldsymbol{S}_i \\
&\quad \boldsymbol{H}_{ii} = \boldsymbol{S}_i^{\mathrm{T}} \boldsymbol{F}_i \\
&\quad j = i \\
&\quad \textbf{while } \lambda(j) \neq 0 \textbf{ do} \\
&\quad\quad \boldsymbol{F}_i = {}^{\lambda(j)}\boldsymbol{X}_j^*\, \boldsymbol{F}_i \\
&\quad\quad j = \lambda(j) \\
&\quad\quad \boldsymbol{H}_{ij} = \boldsymbol{F}_i^{\mathrm{T}} \boldsymbol{S}_j \\
&\quad\quad \boldsymbol{H}_{ji} = \boldsymbol{H}_{ij}^{\mathrm{T}} \\
&\quad \textbf{end} \\
&\quad \boldsymbol{F}_i = {}^0\boldsymbol{X}_j^*\, \boldsymbol{F}_i \\
&\textbf{end}
\end{aligned}
$$

即浮动基公式中的H矩阵，通过这段代码，可以计算得到系统的质量矩阵，这对于进行逆动力学求解非常重要。

```
1  /*!
2   * Computes the Mass Matrix (H) in the inverse dynamics formulation
3   * @return H (_nDof x _nDof matrix)
4   */
5  template <typename T>
6  DMat<T> FloatingBaseModel<T>::massMatrix() {
7  //调用 compositeInertias 方法计算合成惯性矩阵。
8    compositeInertias();
9    _H.setZero();
10
11   // Top left corner is the locked inertia of the whole system
```

```
  //将整个系统的锁定惯性矩阵作为质量矩阵的左上角。这部分对应于机器人的基座。
  _H.template topLeftCorner<6, 6>() = _IC[5].getMatrix();

  for (size_t j = 6; j < _nDof; j++) {
    // f = spatial force required for a unit qdd_j
    SVec<T> f = _IC[j].getMatrix() * _S[j];
    SVec<T> frot = _Irot[j].getMatrix() * _Srot[j];

    _H(j, j) = _S[j].dot(f) + _Srot[j].dot(frot);

    // Propagate down the tree
    f = _Xup[j].transpose() * f + _Xuprot[j].transpose() * frot;
    size_t i = _parents[j];
    while (i > 5) {//从当前关节向上追溯到基座，计算每个关节和基座之间的力。
      // in here f is expressed in frame {i}
      //计算单位关节加速度作用下所需的空间力和旋转力。
      _H(i, j) = _S[i].dot(f);
      _H(j, i) = _H(i, j);

      // Propagate down the tree
      //将力传递到父节点，考虑了父节点的旋转。
      f = _Xup[i].transpose() * f;
      i = _parents[i];
    }

    // Force on floating base
    //将力赋值给质量矩阵中对应的位置，考虑了基座和关节之间的关系
    _H.template block<6, 1>(0, j) = f;
    _H.template block<1, 6>(j, 0) = f.adjoint();
  }
  //返回计算得到的质量矩阵。
  return _H;
}
```

计算广义重力矩阵G

Calculate $\boldsymbol{C}$ and $\boldsymbol{p}_0^{\mathrm{c}}$:

$$
\begin{aligned}
&\boldsymbol{a}_0^{vp} = -{}^0\boldsymbol{a}_g \\
&\textbf{for } i = 1 \textbf{ to } N_B \textbf{ do} \\
&\quad [\boldsymbol{X}_{\mathrm{J}}, \boldsymbol{S}_i, \boldsymbol{v}_{\mathrm{J}}, \boldsymbol{c}_{\mathrm{J}}] = \mathrm{jcalc}(\mathrm{jtype}(i), \boldsymbol{q}_i, \dot{\boldsymbol{q}}_i) \\
&\quad {}^i\boldsymbol{X}_{\lambda(i)} = \boldsymbol{X}_{\mathrm{J}}\, \boldsymbol{X}_{\mathrm{T}}(i) \\
&\quad \textbf{if } \lambda(i) \neq 0 \textbf{ then} \\
&\quad\quad {}^i\boldsymbol{X}_0 = {}^i\boldsymbol{X}_{\lambda(i)}\, {}^{\lambda(i)}\boldsymbol{X}_0 \\
&\quad \textbf{end} \\
&\quad \boldsymbol{v}_i = {}^i\boldsymbol{X}_{\lambda(i)}\, \boldsymbol{v}_{\lambda(i)} + \boldsymbol{v}_{\mathrm{J}} \\
&\quad \boldsymbol{a}_i^{vp} = {}^i\boldsymbol{X}_{\lambda(i)}\, \boldsymbol{a}_{\lambda(i)}^{vp} + \boldsymbol{c}_{\mathrm{J}} + \boldsymbol{v}_i \times \boldsymbol{v}_{\mathrm{J}} \\
&\quad \boldsymbol{f}_i = \boldsymbol{I}_i\, \boldsymbol{a}_i^{vp} + \boldsymbol{v}_i \times^* \boldsymbol{I}_i\, \boldsymbol{v}_i - {}^i\boldsymbol{X}_0^{*}\, {}^0\boldsymbol{f}_i^x \\
&\textbf{end} \\
&\boldsymbol{f}_0 = \boldsymbol{I}_0\, \boldsymbol{a}_0^{vp} + \boldsymbol{v}_0 \times^* \boldsymbol{I}_0\, \boldsymbol{v}_0 - {}^0\boldsymbol{f}_0^x \\
&\textbf{for } i = N_B \textbf{ to } 1 \textbf{ do} \\
&\quad \boldsymbol{C}_i = \boldsymbol{S}_i^{\mathrm{T}} \boldsymbol{f}_i \\
&\quad \boldsymbol{f}_{\lambda(i)} = \boldsymbol{f}_{\lambda(i)} + {}^{\lambda(i)}\boldsymbol{X}_i^{*}\, \boldsymbol{f}_i \\
&\textbf{end} \\
&\boldsymbol{p}_0^{\mathrm{c}} = \boldsymbol{f}_0
\end{aligned}
$$

Calculate $\boldsymbol{H}$, $\boldsymbol{F}$ and $\boldsymbol{I}_0^{\mathrm{c}}$:

$$
\begin{aligned}
&\boldsymbol{H} = \boldsymbol{0} \\
&\textbf{for } i = 0 \textbf{ to } N_B \textbf{ do} \\
&\quad \boldsymbol{I}_i^{\mathrm{c}} = \boldsymbol{I}_i \\
&\textbf{end} \\
&\textbf{for } i = N_B \textbf{ to } 1 \textbf{ do} \\
&\quad \boldsymbol{I}_{\lambda(i)}^{\mathrm{c}} = \boldsymbol{I}_{\lambda(i)}^{\mathrm{c}} + {}^{\lambda(i)}\boldsymbol{X}_i^{*}\, \boldsymbol{I}_i^{\mathrm{c}}\, {}^i\boldsymbol{X}_{\lambda(i)} \\
&\quad \boldsymbol{F}_i = \boldsymbol{I}_i^{\mathrm{c}}\, \boldsymbol{S}_i \\
&\quad \boldsymbol{H}_{ii} = \boldsymbol{S}_i^{\mathrm{T}} \boldsymbol{F}_i \\
&\quad j = i \\
&\quad \textbf{while } \lambda(j) \neq 0 \textbf{ do} \\
&\quad\quad \boldsymbol{F}_i = {}^{\lambda(j)}\boldsymbol{X}_j^{*}\, \boldsymbol{F}_i \\
&\quad\quad j = \lambda(j) \\
&\quad\quad \boldsymbol{H}_{ij} = \boldsymbol{F}_i^{\mathrm{T}} \boldsymbol{S}_j \\
&\quad\quad \boldsymbol{H}_{ji} = \boldsymbol{H}_{ij}^{\mathrm{T}} \\
&\quad \textbf{end} \\
&\quad \boldsymbol{F}_i = {}^0\boldsymbol{X}_j^{*}\, \boldsymbol{F}_i \\
&\textbf{end}
\end{aligned}
$$

注意由于只考虑重力影响，因此后面的离心力、科氏力可以不考虑，但是一般不会这么求，一般将重力矩阵G整合到C矩阵里面。也即浮动基方程由：

$$H(q)\ddot{q} + C(q,\dot{q} + G(q) = S\tau + J_c(q)^T F_c$$

改为：$H(q)\ddot{q} + C(q,\dot{q} = S\tau + J_c(q)^T F_c$

不过MIT min cheat还是进行分开求解。

```cpp
/*!
 * Computes the generalized gravitational force (G) in the inverse dynamics
 * @return G (_nDof x 1 vector)
 */
template <typename T>
DVec<T> FloatingBaseModel<T>::generalizedGravityForce() {
  compositeInertias();
//创建一个重力加速度的空间向量 aGravity，表示在世界坐标系下的重力加速度，其中 gravity 是表示重力矢量的数组。
  SVec<T> aGravity;
  aGravity << 0, 0, 0, _gravity[0], _gravity[1], _gravity[2];
  //将重力加速度转换到基座坐标系，并存储在 ag[5] 中。
  _ag[5] = _Xup[5] * aGravity;

  // Gravity comp force is the same as force required to accelerate
  // oppostite gravity
```

```
  //计算基座上的广义重力力，即应用于机器人基座的反向重力作用力。
  _G.template topRows<6>() = -_IC[5].getMatrix() * _ag[5];
  for (size_t i = 6; i < _nDof; i++) {
    _ag[i] = _Xup[i] * _ag[_parents[i]];
    _agrot[i] = _Xuprot[i] * _ag[_parents[i]];

    // body and rotor
    //计算每个关节上的广义重力力，即应用于每个关节的反向重力作用力。
    _G[i] = -_S[i].dot(_IC[i].getMatrix() * _ag[i]) -
            _Srot[i].dot(_Irot[i].getMatrix() * _agrot[i]);
  }
  //返回计算得到的广义重力力向量。
  return _G;
}
```

计算广义偏置力

广义偏置力由离心力、科氏力等组成

Calculate $\boldsymbol{C}$ and $\boldsymbol{p}_0^c$:

$$
\begin{aligned}
&\boldsymbol{a}_0^{vp} = -{}^0\boldsymbol{a}_g \\
&\textbf{for } i = 1 \textbf{ to } N_B \textbf{ do} \\
&\quad [\boldsymbol{X}_J, \boldsymbol{S}_i, \boldsymbol{v}_J, \boldsymbol{c}_J] = \text{jcalc}(\text{jtype}(i), \boldsymbol{q}_i, \dot{\boldsymbol{q}}_i) \\
&\quad {}^i\boldsymbol{X}_{\lambda(i)} = \boldsymbol{X}_J \, \boldsymbol{X}_T(i) \\
&\quad \textbf{if } \lambda(i) \neq 0 \textbf{ then} \\
&\quad\quad {}^i\boldsymbol{X}_0 = {}^i\boldsymbol{X}_{\lambda(i)} \, {}^{\lambda(i)}\boldsymbol{X}_0 \\
&\quad \textbf{end} \\
&\quad \boldsymbol{v}_i = {}^i\boldsymbol{X}_{\lambda(i)} \, \boldsymbol{v}_{\lambda(i)} + \boldsymbol{v}_J \\
&\quad \boldsymbol{a}_i^{vp} = {}^i\boldsymbol{X}_{\lambda(i)} \, \boldsymbol{a}_{\lambda(i)}^{vp} + \boldsymbol{c}_J + \boldsymbol{v}_i \times \boldsymbol{v}_J \\
&\quad \boldsymbol{f}_i = \boldsymbol{I}_i \, \boldsymbol{a}_i^{vp} + \boldsymbol{v}_i \times^* \boldsymbol{I}_i \, \boldsymbol{v}_i - {}^i\boldsymbol{X}_0^* \, {}^0\boldsymbol{f}_i^x \\
&\textbf{end} \\
&\boldsymbol{f}_0 = \boldsymbol{I}_0 \, \boldsymbol{a}_0^{vp} + \boldsymbol{v}_0 \times^* \boldsymbol{I}_0 \, \boldsymbol{v}_0 - {}^0\boldsymbol{f}_0^x \\
&\textbf{for } i = N_B \textbf{ to } 1 \textbf{ do} \\
&\quad \boldsymbol{C}_i = \boldsymbol{S}_i^T \boldsymbol{f}_i \\
&\quad \boldsymbol{f}_{\lambda(i)} = \boldsymbol{f}_{\lambda(i)} + {}^{\lambda(i)}\boldsymbol{X}_i^* \, \boldsymbol{f}_i \\
&\textbf{end} \\
&\boldsymbol{p}_0^c = \boldsymbol{f}_0
\end{aligned}
$$

Calculate $\boldsymbol{H}$, $\boldsymbol{F}$ and $\boldsymbol{I}_0^c$:

$$
\begin{aligned}
&\boldsymbol{H} = \boldsymbol{0} \\
&\textbf{for } i = 0 \textbf{ to } N_B \textbf{ do} \\
&\quad \boldsymbol{I}_i^c = \boldsymbol{I}_i \\
&\textbf{end} \\
&\textbf{for } i = N_B \textbf{ to } 1 \textbf{ do} \\
&\quad \boldsymbol{I}_{\lambda(i)}^c = \boldsymbol{I}_{\lambda(i)}^c + {}^{\lambda(i)}\boldsymbol{X}_i^* \, \boldsymbol{I}_i^c \, {}^i\boldsymbol{X}_{\lambda(i)} \\
&\quad \boldsymbol{F}_i = \boldsymbol{I}_i^c \, \boldsymbol{S}_i \\
&\quad \boldsymbol{H}_{ii} = \boldsymbol{S}_i^T \boldsymbol{F}_i \\
&\quad j = i \\
&\quad \textbf{while } \lambda(j) \neq 0 \textbf{ do} \\
&\quad\quad \boldsymbol{F}_i = {}^{\lambda(j)}\boldsymbol{X}_j^* \, \boldsymbol{F}_i \\
&\quad\quad j = \lambda(j) \\
&\quad\quad \boldsymbol{H}_{ij} = \boldsymbol{F}_i^T \boldsymbol{S}_j \\
&\quad\quad \boldsymbol{H}_{ji} = \boldsymbol{H}_{ij}^T \\
&\quad \textbf{end} \\
&\quad \boldsymbol{F}_i = {}^0\boldsymbol{X}_j^* \, \boldsymbol{F}_i \\
&\textbf{end}
\end{aligned}
$$

```
/*!
 * Computes the generalized coriolis forces (Cqd) in the inverse dynamics
```

```
 * @return Cqd (_nDof x 1 vector)
 */
template <typename T>
DVec<T> FloatingBaseModel<T>::generalizedCoriolisForce() {
  biasAccelerations();

  // Floating base force
  Mat6<T> Ifb = _Ibody[5].getMatrix();
  //计算基座上的角动量。
  SVec<T> hfb = Ifb * _v[5];
  //计算基座上的速度乘积加速度，并加上角动量产生的力。
  _fvp[5] = Ifb * _avp[5] + forceCrossProduct(_v[5], hfb);

  for (size_t i = 6; i < _nDof; i++) {
    // Force on body i
    //获取链接 i 的惯性矩阵。
    Mat6<T> Ii = _Ibody[i].getMatrix();
    //计算链接 i 上的角动量。
    SVec<T> hi = Ii * _v[i];
    //计算链接 i 上的速度乘积加速度，并加上角动量产生的力。
    _fvp[i] = Ii * _avp[i] + forceCrossProduct(_v[i], hi);

    // Force on rotor i
    //获取旋转部件 i 的惯性矩阵
    Mat6<T> Ir = _Irot[i].getMatrix();
    //计算旋转部件 i 上的角动量。
    SVec<T> hr = Ir * _vrot[i];
    //计算旋转部件 i 上的速度乘积加速度，并加上角动量产生的力。
    _fvprot[i] = Ir * _avprot[i] + forceCrossProduct(_vrot[i], hr);
  }
//从最后一个关节开始往基座遍历，计算广义科里奥利力的传递。
  for (size_t i = _nDof - 1; i > 5; i--) {
    // Extract force along the joints
    //计算关节 i 上的广义科里奥利力，其等于速度乘积加速度在关节力方向上的投影。
    _Cqd[i] = _S[i].dot(_fvp[i]) + _Srot[i].dot(_fvprot[i]);

    // Propage force down the tree
    //将计算得到的广义科里奥利力传递到父关节。
    _fvp[_parents[i]] += _Xup[i].transpose() * _fvp[i];
    //同样将计算得到的广义科里奥利力传递到父关节。
    _fvp[_parents[i]] += _Xuprot[i].transpose() * _fvprot[i];
  }

  // Force on floating base
  //将基座上的广义科里奥利力赋值给 _Cqd 的前六行，以保证基座部分的力被正确计算。
  _Cqd.template topRows<6>() = _fvp[5];
  //返回计算得到的广义科里奥利力
```

```
50    return _Cqd;
51 }
```

任务更新_ContactTaskUpdate

对应代码

```
1  void LocomotionCtrl<T>::_ContactTaskUpdate(void* input, ControlFSMData<T>
   & data){
2    _input_data = static_cast<LocomotionCtrlData<T>* >(input);
3
4    _ParameterSetup(data.userParameters);
5
6    // Wash out the previous setup
7    _CleanUp();
8  //将身体的期望欧拉角转换为四元数，并将其赋值给 _quat_des。
9    _quat_des = ori::rpyToQuat(_input_data->pBody_RPY_des);
10
11   Vec3<T> zero_vec3; zero_vec3.setZero();
12   //更新身体姿态任务（orientation task），传入期望的姿态四元数、期望的身体角速度和零
   加速度。
13   _body_ori_task->UpdateTask(&_quat_des, _input_data->vBody_Ori_des,
   zero_vec3);
14   //更新身体位置任务（position task），传入期望的身体位置、速度和加速度。
15   _body_pos_task->UpdateTask(
16       &(_input_data->pBody_des),
17       _input_data->vBody_des,
18       _input_data->aBody_des);
19 //将更新后的身体姿态任务和位置任务添加到控制器的任务列表中。
20   WBCtrl::_task_list.push_back(_body_ori_task);
21   WBCtrl::_task_list.push_back(_body_pos_task);
22
23   for(size_t leg(0); leg<4; ++leg){
24   //如果当前腿处于接触状态，则执行接触任务的更新；否则执行摆动任务的更新。
25     if(_input_data->contact_state[leg] > 0.){ // Contact
26     //设置当前腿的期望接触力。
27       _foot_contact[leg]->setRFDesired((DVec<T>)(_input_data->Fr_des[leg]));
28       //更新当前腿的接触规范
29       _foot_contact[leg]->UpdateContactSpec();
30       //将更新后的脚部接触信息添加到控制器的接触列表中。
31       WBCtrl::_contact_list.push_back(_foot_contact[leg]);
32
33     }else{ // No Contact (swing)
34     //更新当前腿的脚部任务，传入期望的脚部位置、速度和加速度。
35       _foot_task[leg]->UpdateTask(
```

```
            &(_input_data->pFoot_des[leg]),
            _input_data->vFoot_des[leg],
            _input_data->aFoot_des[leg]);
            //zero_vec3);
            //将更新后的脚部任务添加到控制器的任务列表中。
        WBCtrl::_task_list.push_back(_foot_task[leg]);
      }
    }
}
```

一共有4个任务，按照各个任务的重要性优先级划分为：

0级：足端接触力

1级：机器狗的姿态（不能翻）

2级：机器狗的位置（可以释放浮动）

3级：摆动退的轨迹（级别最低）

WBC解算

函数为_ComputeWBC()：

```
template <typename T>
void WBC_Ctrl<T>::_ComputeWBC() {
  // TEST
  //带有冗余优先级控制的WBC
  _kin_wbc->FindConfiguration(_full_config, _task_list, _contact_list,
                              _des_jpos, _des_jvel);

  // WBIC
  //阻抗WBC
  _wbic->UpdateSetting(_A, _Ainv, _coriolis, _grav);
  //输出力矩
  _wbic->MakeTorque(_tau_ff, _wbic_data);
}
```

其中_kin_wbc为了进行WBC的计算，得到冗余自由度下的角度，角速度，角加速度。

$$\Delta \mathbf{q}_i = \Delta \mathbf{q}_{i-1} + \boldsymbol{J}_{i|pre}{}^{\dagger} \left(\mathbf{e}_i - \boldsymbol{J}_i \Delta \mathbf{q}_{i-1} \right), \tag{16}$$

$$\dot{\mathbf{q}}_i^{\mathrm{cmd}} = \dot{\mathbf{q}}_{i-1}^{\mathrm{cmd}} + \boldsymbol{J}_{i|pre}{}^{\dagger} \left(\dot{\mathbf{x}}_i^{\mathrm{des}} - \boldsymbol{J}_i \dot{\mathbf{q}}_{i-1}^{\mathrm{cmd}} \right), \tag{17}$$

$$\ddot{\mathbf{q}}_i^{\mathrm{cmd}} = \ddot{\mathbf{q}}_{i-1}^{\mathrm{cmd}} + \overline{\boldsymbol{J}_{i|pre}^{\mathrm{dyn}}} \left(\ddot{\mathbf{x}}_i^{\mathrm{cmd}} - \dot{\boldsymbol{J}}_i \dot{\mathbf{q}} - \boldsymbol{J}_i \ddot{\mathbf{q}}_{i-1}^{\mathrm{cmd}} \right), \tag{18}$$

where

$$\begin{aligned} \boldsymbol{J}_{i|pre} &= \boldsymbol{J}_i \boldsymbol{N}_{i-1}, \\ \boldsymbol{N}_{i-1} &= \boldsymbol{N}_0 \boldsymbol{N}_{1|0} \cdots \boldsymbol{N}_{i-1|i-2}, \end{aligned} \tag{19}$$

$$\begin{aligned} \boldsymbol{N}_0 &= \boldsymbol{I} - \boldsymbol{J}_c^{\dagger} \boldsymbol{J}_c, \\ \boldsymbol{N}_{i|i-1} &= \boldsymbol{I} - \boldsymbol{J}_{i|i-1}{}^{\dagger} \boldsymbol{J}_{i|i-1}. \end{aligned} \tag{20}$$

Here, $i \geq 1$, and

$$\begin{aligned} \Delta \mathbf{q}_0, \ \dot{\mathbf{q}}_0^{\mathrm{cmd}} &= \mathbf{0}, \\ \ddot{\mathbf{q}}_0^{\mathrm{cmd}} &= \overline{\boldsymbol{J}_c^{\mathrm{dyn}}} (-\boldsymbol{J}_c \dot{\mathbf{q}}). \end{aligned} \tag{21}$$

$\mathbf{e}_i$ is the position error defined by $\mathbf{x}_i^{\mathrm{des}} - \mathbf{x}_i$ and $\ddot{\mathbf{x}}_i^{\mathrm{cmd}}$ is the acceleration command of i-th task defined by

$$\ddot{\mathbf{x}}_i^{\mathrm{cmd}} = \ddot{\mathbf{x}}^{\mathrm{des}} + \boldsymbol{K}_p \left(\mathbf{x}_i^{\mathrm{des}} - \mathbf{x}_i \right) + \boldsymbol{K}_d \left(\dot{\mathbf{x}}^{\mathrm{des}} - \dot{\mathbf{x}} \right), \tag{22}$$

_wbic为了松弛优化，得到最终的_tau_ff。

其中_kin_wbc计算得到关节角加速度，理论有了角加速度和单刚体动力学计算得到的足底力 就可以根据浮动基公式计算得到关节反馈力 ，但是理论计算的结果无法满足等式，需要加上松弛变量:

$$\begin{cases} \ddot{\boldsymbol{q}} = \ddot{\boldsymbol{q}}^{cmd} + \begin{bmatrix} \boldsymbol{\delta}_{\ddot{q}} \\ \mathbf{0} \end{bmatrix} \\ {}^{\mathcal{O}}\boldsymbol{f}_c = {}^{\mathcal{O}}\boldsymbol{f}^{MPC} + \boldsymbol{\delta}_f \end{cases}$$

_kin_WBC对应代码

```
template <typename T>
bool KinWBC<T>::FindConfiguration(
    const DVec<T>& curr_config, const std::vector<Task<T>*>& task_list,
    const std::vector<ContactSpec<T>*>& contact_list, DVec<T>& jpos_cmd,
    DVec<T>& jvel_cmd) {

  // Contact Jacobian Setup
  DMat<T> Nc(num_qdot_, num_qdot_); Nc.setIdentity();
```

```
  if(contact_list.size() > 0){
    DMat<T> Jc, Jc_i;
    //获取第一个接触雅可比矩阵 Jc。
    contact_list[0]->getContactJacobian(Jc);
    size_t num_rows = Jc.rows();
    //遍历接触规范列表，将所有接触雅可比矩阵拼接成一个大的雅可比矩阵 Jc。
    for (size_t i(1); i < contact_list.size(); ++i) {
      contact_list[i]->getContactJacobian(Jc_i);
      size_t num_new_rows = Jc_i.rows();
      Jc.conservativeResize(num_rows + num_new_rows, num_qdot_);
      Jc.block(num_rows, 0, num_new_rows, num_qdot_) = Jc_i;
      num_rows += num_new_rows;
    }

    // Projection Matrix
    //根据接触雅可比矩阵构建投影矩阵 Nc。
    _BuildProjectionMatrix(Jc, Nc);
  }

  // First Task
  DVec<T> delta_q, qdot;
  DMat<T> Jt, JtPre, JtPre_pinv, N_nx, N_pre;
   //获取任务列表中的第一个任务。
  Task<T>* task = task_list[0];
  //获取第一个任务的任务雅可比矩阵 Jt
  task->getTaskJacobian(Jt);
  //计算任务雅可比矩阵在接触约束下的投影 JtPre。
  JtPre = Jt * Nc;
  //计算投影雅可比矩阵的伪逆 JtPre_pinv。
  _PseudoInverse(JtPre, JtPre_pinv);
  //计算关节位移和速度命令。
  delta_q = JtPre_pinv * (task->getPosError());
  qdot = JtPre_pinv * (task->getDesVel());
  //保存前一个任务的关节位移和速度命令。
  DVec<T> prev_delta_q = delta_q;
  DVec<T> prev_qdot = qdot;

  _BuildProjectionMatrix(JtPre, N_nx);
  N_pre = Nc * N_nx;
   //遍历剩余的任务列表。
  for (size_t i(1); i < task_list.size(); ++i) {
    task = task_list[i];
    //获取当前任务的任务雅可比矩阵 Jt
    task->getTaskJacobian(Jt);
    //计算当前任务雅可比矩阵在之前任务的投影下的投影 JtPre
    JtPre = Jt * N_pre;

```

```
    _PseudoInverse(JtPre, JtPre_pinv);
    //更新关节位移和速度命令。
    delta_q =
        prev_delta_q + JtPre_pinv * (task->getPosError() - Jt *
prev_delta_q);
    qdot = prev_qdot + JtPre_pinv * (task->getDesVel() - Jt * prev_qdot);

    // For the next task
    //根据当前任务雅可比矩阵构建新的投影矩阵 N_nx。
    _BuildProjectionMatrix(JtPre, N_nx);
    //更新整体投影矩阵 N_pre。
    N_pre *= N_nx;
    prev_delta_q = delta_q;
    prev_qdot = qdot;
  }
  //保存当前任务的关节位移和速度命令，以备下一次迭代使用。
  for (size_t i(0); i < num_act_joint_; ++i) {
    jpos_cmd[i] = curr_config[i + 6] + delta_q[i + 6];
    jvel_cmd[i] = qdot[i + 6];
  }
  return true;
}
```

WBIC代码

```
template <typename T>
void WBIC<T>::MakeTorque(DVec<T>& cmd, void* extra_input) {
  if (!WB::b_updatesetting_) {
    printf("[Wanning] WBIC setting is not done\n");
  }
  //如果提供了额外的输入，则将其转换为 WBIC_ExtraData 类型并存储在 _data 中。
  if (extra_input) _data = static_cast<WBIC_ExtraData<T>*>(extra_input);

  // resize G, g0, CE, ce0, CI, ci0
  //设置优化问题的大小，调整相关矩阵和向量的维度。
  _SetOptimizationSize();
  //设置优化问题的成本函数。
  _SetCost();

  DVec<T> qddot_pre;
  DMat<T> JcBar;
  DMat<T> Npre;
  //如果存在接触力约束。
  if (_dim_rf > 0) {
    // Contact Setting
```

```
    //构建接触力约束。
    _ContactBuilding();

    // Set inequality constraints
    //设置不等式约束。
    _SetInEqualityConstraint();
    //使用加权逆求解接触力雅可比矩阵的伪逆。
    WB::_WeightedInverse(_Jc, WB::Ainv_, JcBar);
    //根据接触力导数和加权逆求解先验关节加速度。
    qddot_pre = JcBar * (-_JcDotQdot);
    //根据接触力雅可比矩阵和加权逆计算投影矩阵。
    Npre = _eye - JcBar * _Jc;

  } else {
    qddot_pre = DVec<T>::Zero(WB::num_qdot_);
    Npre = _eye;
  }

  // Task
  Task<T>* task;
  DMat<T> Jt, JtBar, JtPre;
  DVec<T> JtDotQdot, xddot;

  for (size_t i(0); i < (*_task_list).size(); ++i) {
    task = (*_task_list)[i];

    task->getTaskJacobian(Jt);
    task->getTaskJacobianDotQdot(JtDotQdot);
    task->getCommand(xddot);
    //根据投影矩阵计算任务的修正雅可比矩阵。
    JtPre = Jt * Npre;
    //使用加权逆求解任务的修正雅可比矩阵的伪逆。
    WB::_WeightedInverse(JtPre, WB::Ainv_, JtBar);
    //根据任务的期望加速度和修正雅可比矩阵计算先验关节加速度。
    qddot_pre += JtBar * (xddot - JtDotQdot - Jt * qddot_pre);
    //更新投影矩阵。
    Npre = Npre * (_eye - JtBar * JtPre);

  }

  // Set equality constraints
  //设置相等约束
  _SetEqualityConstraint(qddot_pre);
    //调用求解二次规划的函数，求解优化问题并返回目标函数值。
  T f = solve_quadprog(G, g0, CE, ce0, CI, ci0, z);
  // std::cout<<"\n wbic old time: "<<timer.getMs()<<std::endl;
  (void)f;
```

```
68
69    // pretty_print(qddot_pre, std::cout, "qddot_cmd");
70    //将优化问题的解添加到先验关节加速度中。
71    for (size_t i(0); i < _dim_floating; ++i) qddot_pre[i] += z[i];
72    //根据优化问题的解生成最终的关节力矩命令。
73    _GetSolution(qddot_pre, cmd);
74      //存储优化问题的解。
75    _data->_opt_result = DVec<T>(_dim_opt);
76    //将优化问题的解存储在 _data->_opt_result 中。
77    for (size_t i(0); i < _dim_opt; ++i) {
78      _data->_opt_result[i] = z[i];
79    }
80
81  }
```

松弛优化：在文件WBIC中，用上一步得到的加速度和MPC中得到的反作用力计算最终的反馈力。参照的是QP求解器优化算法。

$$min_{\delta_{F_r},\delta_{F_r}} \quad \delta_{f_r}^T Q_1 \delta_{f_r} + \delta_f^T Q_2 \delta_f \tag{1}$$

s.t.

$$S_f(A\ddot{q} + b + g) = S_f J_c^T f_r,$$
$$\ddot{q} = \ddot{q}^{cmd} + \begin{bmatrix} \delta_f \\ 0_{n_j} \end{bmatrix},$$
$$f_r = f_r^{MPC} + \delta_{f_r},$$
$$Wf_r \geq 0$$

对应代码

```
1  template <typename T>
2  void WBIC<T>::MakeTorque(DVec<T>& cmd, void* extra_input) {
3    if (!WB::b_updatesetting_) {
4      printf("[Wanning] WBIC setting is not done\n");
5    }
6    if (extra_input) _data = static_cast<WBIC_ExtraData<T>*>(extra_input);
7
8    // resize G, g0, CE, ce0, CI, ci0
9    _SetOptimizationSize();
10   _SetCost();
11
12   DVec<T> qddot_pre;
13   DMat<T> JcBar;
14   DMat<T> Npre;
```

```
15
16    if (_dim_rf > 0) {
17      // Contact Setting
18      _ContactBuilding();
19
20      // Set inequality constraints
21      _SetInEqualityConstraint();
22      WB::_WeightedInverse(_Jc, WB::Ainv_, JcBar);
23      qddot_pre = JcBar * (-_JcDotQdot);
24      Npre = _eye - JcBar * _Jc;
25      // pretty_print(JcBar, std::cout, "JcBar");
26      // pretty_print(_JcDotQdot, std::cout, "JcDotQdot");
27      // pretty_print(qddot_pre, std::cout, "qddot 1");
28    } else {
29      qddot_pre = DVec<T>::Zero(WB::num_qdot_);
30      Npre = _eye;
31    }
32
33    // Task
34    Task<T>* task;
35    DMat<T> Jt, JtBar, JtPre;
36    DVec<T> JtDotQdot, xddot;
37
38    for (size_t i(0); i < (*_task_list).size(); ++i) {
39      task = (*_task_list)[i];
40
41      task->getTaskJacobian(Jt);
42      task->getTaskJacobianDotQdot(JtDotQdot);
43      task->getCommand(xddot);
44
45      JtPre = Jt * Npre;
46      WB::_WeightedInverse(JtPre, WB::Ainv_, JtBar);
47
48      qddot_pre += JtBar * (xddot - JtDotQdot - Jt * qddot_pre);
49      Npre = Npre * (_eye - JtBar * JtPre);
50
51      // pretty_print(xddot, std::cout, "xddot");
52      // pretty_print(JtDotQdot, std::cout, "JtDotQdot");
53      // pretty_print(qddot_pre, std::cout, "qddot 2");
54      // pretty_print(Jt, std::cout, "Jt");
55      // pretty_print(JtPre, std::cout, "JtPre");
56      // pretty_print(JtBar, std::cout, "JtBar");
57    }
58
59    // Set equality constraints
60    _SetEqualityConstraint(qddot_pre);
61
```

```
62    // printf("G:\n");
63    // std::cout<<G<<std::endl;
64    // printf("g0:\n");
65    // std::cout<<g0<<std::endl;
66
67    // Optimization
68    // Timer timer;
69    T f = solve_quadprog(G, g0, CE, ce0, CI, ci0, z);
70    // std::cout<<"\n wbic old time: "<<timer.getMs()<<std::endl;
71    (void)f;
72
73    // pretty_print(qddot_pre, std::cout, "qddot_cmd");
74    for (size_t i(0); i < _dim_floating; ++i) qddot_pre[i] += z[i];
75    _GetSolution(qddot_pre, cmd);
76
77    _data->_opt_result = DVec<T>(_dim_opt);
78    for (size_t i(0); i < _dim_opt; ++i) {
79      _data->_opt_result[i] = z[i];
80    }
81
82    // std::cout << "f: " << f << std::endl;
83    //std::cout << "x: " << z << std::endl;
84
85    // DVec<T> check_eq = _dyn_CE * _data->_opt_result + _dyn_ce0;
86    // pretty_print(check_eq, std::cout, "equality constr");
87    // std::cout << "cmd: "<<cmd<<std::endl;
88    // pretty_print(qddot_pre, std::cout, "qddot_pre");
89    // pretty_print(JcN, std::cout, "JcN");
90    // pretty_print(Nci_, std::cout, "Nci");
91    // DVec<T> eq_check = dyn_CE * data_->opt_result_;
92    // pretty_print(dyn_ce0, std::cout, "dyn ce0");
93    // pretty_print(eq_check, std::cout, "eq_check");
94
95    // pretty_print(Jt, std::cout, "Jt");
96    // pretty_print(JtDotQdot, std::cout, "Jtdotqdot");
97    // pretty_print(xddot, std::cout, "xddot");
98
99    // printf("CE:\n");
100   // std::cout<<CE<<std::endl;
101   // printf("ce0:\n");
102   // std::cout<<ce0<<std::endl;
103
104   // printf("CI:\n");
105   // std::cout<<CI<<std::endl;
106   // printf("ci0:\n");
107   // std::cout<<ci0<<std::endl;
108 }
```

优化问题

优化有两个东西需要解决，第一个就是找到代价函数，第二个就是权重（调参）。

这里列出LQR和MPC的代价函数：

LQR:

$$J_{min} = \int_0^{\infty} (x^T Q x + u^T R u) dt$$

权重：Q和R

MPC:

$$J_{min} = \Sigma_k^{j-1} (e^T Q e + u_k^T R u_k + e_N^T F e_N)$$

其中 e_N 表示误差的终值，也是衡量优劣的一种标准。

权重：Q、R和F

从上面的方程可以发现：

LQR是对整个时间的一个最优评估，而MPC是当前时刻对过去的一个最优评估。

因此当你对LQR的代价方程 J_{min} 求u的梯度，即 $\triangle u = 0$，会得到一个形如 $u = kx$ 的表达式，这也就导致了LQR存在一个闭环控制的形式。而MPC的代价方程就会麻烦点，根据状态方程对代价方程进行整理后可得：

$$J(U_k) = U_k^T (\Psi^T Q \Psi + R) U_k + (2 e^T Q \Psi) U_K + e^T Q e$$

上式 $e^T Q e$ 是常数项，对结果没影响，可以舍去。

也就是对这个方程进行最优化求解，进一步进行简化整理可以得到形如：

$$H = 2(\Psi^T Q \Psi + R)$$
$$f^T = 2 e^T q \Psi$$

则代价函数变为：

$$J(U_k)_{min} = 1/2 U_k^T H U_k + f^T x$$

刚好它就满足具有线性约束的二次目标函数的求解，可以用现成的求解器Quadprog对其进行求解，并且可以对状态变量或者输入进行约束。

从MPC的求解过程可以看出，优化是开环的，从而也就导致了我可以将一些约束加入进去，反正是对未来的优化，我加入一些约束也不会对现在造成什么影响，而LQR则不同，他没法加入不等式约束，它是一个闭环的，他会存在一个最优解，如果这个最优解刚好在你的不等式约束之外，那就会导致无法求出，因此LQR的约束就只有那个状态方程，而MPC则可以加不等式约束。

特殊符号