1 Introduction

The objective of this project is the writing of a crocoddyl class in C ++ in order to use it in a simulation of the quadruped. It is based on a simplified dynamical model of the quadruped and used to solve a MPC problem. The development in c++ would allow to reach the 0.02s of execution time needed.

The new quadruped action model class is mainly based on the unicycle example class which derives from on the ActionModelAbstract class.

The results are taken from the following repository : https ://gitlab.laas.fr/loco-3d/quadruped-walkgen/-/tree/devel1.

1.1 Unicycle example

This section is focused on the unicycle benchmark in c++ to get an order of magnitude. The ddp solver run with only 1 iteration. There are 200 nodes, so 200 called are done to calc in the function ShootingProblem.calc for example. The the results are obtained with 5000 trials.

Function called	Mean [ms]	Min - Max [ms]
DDP.solve [ms]	0.399744	0.390704 - 0.984936
ShootingProblem.calc [ms]	0.0139808	0.013108 - 0.333873
ShootingProblem.calcDiff [ms]	0.0176839	0.016154 - 0.083063

1.2 C++ Benchmark for quadruped action model

The next results are obtained with 5000 trials. There are 16 nodes and the maximum iteration is set to 5. The acceptable limit is 20ms to solve the MPC problem.

Function called	Mean [ms]	Min - Max [ms]
DDP.solve [ms]	1.49355	1.44367 - 4.63013
ShootingProblem.calc [ms]	0.00558205	0.005353 - 0.019979
ShootingProblem.calcDiff [ms]	0.00942601	0.008905 - 0.026563
UpdateModel [ms]	0.00591416	0.005256 - 0.050072

1.2.1 Calc function

This part this part gives the detail of the code of the calc function.

Listing 1 – C++ part of calc function

```
1  template <typename Scalar>
2  void ActionModelQuadrupedTpl<Scalar>::calc(...) {
3      // ...test on the arguments ...
4
5    ActionDataQuadrupedTpl<Scalar>* d = static_cast<...;
6
7    // Discrete dynamic, operation 1
8    d->xnext << A.diagonal().cwiseProduct(x) + B*u + g;
9
10   // Residual cost on the state and force norm
11   // Operation 2
12   d->r.template head<12>() =  state_weights_.asDiagonal() * (x - xref_);
13   // Operation 3
14   d->r.template tail<12>() =  force_weights_.asDiagonal() * u;
15
16   // Cost relative to the friction cone, operation 4 :
17   for (int i=0; i<4; i=i+1){
18     Fa_x_u.segment(5*i,5) << u(3*i) - mu*u(3*i+2) , -u(3*i) - mu*u(3*i+2),
```

```
                                    u(3*i+1) - mu*u(3*i+2)  , -u(3*i+1) - mu*u(3*i+2),
                                    u(3*i+2) ;
  }
  rlb_min_ = (Fa_x_u - lb).array().min(0.);
  rub_max_ = (Fa_x_u - ub).array().max(0.);

  // Cost computation, operation 5 :
  d->cost = 0.5 * d->r.transpose() * d->r
  + friction_weight_ * (Scalar(0.5) * rlb_min_.matrix().squaredNorm() +
                        Scalar(0.5) * rub_max_.matrix().squaredNorm()) ;

}
```

1.2.2 CalcDiff function

This part this part gives the detail of the code of the calcDiff function. The operations that involves large matrix have been with block operations.

Listing 2 – C++ part of calcDiff function

```
template <typename Scalar>
void ActionModelQuadrupedTpl<Scalar>::calcDiff(...) {
//... Test the argument size ...

  ActionDataQuadrupedTpl<Scalar>* d = static_cast<... ;

  // Matrix friction cone hessian, Operation 1
  rlb_ = Fa_x_u - lb ;
  rub_ = Fa_x_u - ub ;
  Arr.diagonal() =
        ((rlb_.array() <= 0.) +
         (rub_.array() >= 0.)     ).matrix().template cast<Scalar>() ;

  // Cost derivatives, Operation 2
  d->Lx = (state_weights_.array()* d->r.template head<12>().array()).matrix() ;

  // Operation 3
  Scalar r1 = friction_weight_*(rlb_min_(0) + rub_max_(0)) ;
  Scalar r2 = friction_weight_*(rlb_min_(1) + rub_max_(1)) ;
  Scalar r3 = friction_weight_*(rlb_min_(2) + rub_max_(2)) ;
  Scalar r4 = friction_weight_*(rlb_min_(3) + rub_max_(3)) ;
  Scalar r5 = friction_weight_*(rlb_min_(4) + rub_max_(4)) ;
  d->Lu.block(0,0,3,1) << r1 - r2 , r3 - r4 , -mu*(r1 + r2 + r3 + r4 ) + r5 ;
  for (int i=1; i<4; i=i+1){
    r1 = friction_weight_*(rlb_min_(5*i) + rub_max_(5*i)) ;
    r2 = friction_weight_*(rlb_min_(5*i+1) + rub_max_(5*i+1)) ;
    r3 = friction_weight_*(rlb_min_(5*i+2) + rub_max_(5*i+2)) ;
    r4 = friction_weight_*(rlb_min_(5*i+3) + rub_max_(5*i+3)) ;
    r5 = friction_weight_*(rlb_min_(5*i+4) + rub_max_(5*i+4)) ;
    d->Lu.block(i*3,0,3,1) << r1 - r2, r3 - r4, -mu*(r1 + r2 + r3 + r4)+ r5 ;
  }
  d->Lu += (force_weights_.array()*d->r.template tail<12>().array()).matrix() ;

  // Operation 4
  d->Lxx.diagonal() = (state_weights_.array() * state_weights_.array()).matrix() ;

  // Operation 5
```

```
38    d->Luu = friction_weight_ * ( Fa.transpose()*Arr*Fa) ;
39
40    // Operation 6
41    for (int i=0; i<4; i=i+1){
42         r1 = friction_weight_*Arr(5*i,5*i) ;
43         r2 = friction_weight_*Arr(5*i+1,5*i+1) ;
44         r3 = friction_weight_*Arr(5*i+2,5*i+2) ;
45         r4 = friction_weight_*Arr(5*i+3,5*i+3) ;
46         r5 = friction_weight_*Arr(5*i+4,5*i+4) ;
47         d->Luu.block(3*i,3*i,3,3) << r1 + r2 ,     0.0     , mu*(r2 - r1 ),
48                                      0.0     ,  r3 + r4  , mu*(r4 - r3 ),
49                      mu*(r2 - r1 ), mu*(r4 - r3), mu*mu*(r1 + r2 + r3 + r4)+ r5
    ;
50    }
51
52    // Dynamic derivatives, Operation 7
53    d->Fx << A;
54    d->Fu << B;
55
56
57 }
```

1.3 Friction cone cost, explanation

The command vector is $u^T = \begin{bmatrix} f_{x1} & f_{y1} & f_{z1} \dots & f_{xn} & f_{yn} & f_{zn} \end{bmatrix}$ where f_{x1} is the ground reaction force among the x-axis in the local frame of the first foot. The following constraints need to be respected :

$$|f_x| < \mu f_z \qquad |f_y| < \mu f_z \qquad f_z > 0 \tag{1}$$

The friction cone is a discrete approximation of the cone friction with 4 facets. For each foot the following residual vector is computed :

$$r = Au_i = \begin{bmatrix} 1 & 0 & -\mu \\ -1 & 0 & -\mu \\ 0 & 1 & -\mu \\ 0 & -1 & -\mu \\ 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} f_{xi} \\ f_{yi} \\ f_{zi} \end{bmatrix} = \begin{bmatrix} r_1 & r_2 & r_3 & r_4 & r_5 \end{bmatrix}$$

It corresponds to the operation 4 in calc Function. The A matrix is actually of size 20*12 (4 feet), and to avoid unnecessary null operation, the calculation is done by block.

Then, the cost relative to one friction cone $a_i(r)$, is :

$$a_i(r) = \frac{1}{2}||r_1^+||^2 + \dots + \frac{1}{2}||r_5^+||^2 = \frac{1}{2}||r^+||^2 \tag{2}$$

where $y^+ = y$ if y > 0 and 0 if otherwise.

The cost function relative to one friction cone is $a_i(r)$. The partial derivative of this cost can be written as : - operation 3 -

$$L_{u,i} = A^T \begin{bmatrix} r_1^+ \\ r_2^+ \\ r_3^+ \\ r_4^+ \\ r_5^+ \end{bmatrix} = \begin{bmatrix} r_1^+ - r_2^+ \\ r_3^+ - r_4^+ \\ -\mu(r_1^+ + r_2^+ + r_3^+ + r_4^+) - r_5^+ \end{bmatrix} \quad => \quad L_u = \begin{bmatrix} L_{u1} & \dots & L_{u5} \end{bmatrix}$$

and the Hessian, the calculation can be made by block too : - operation 6 -

$$L_{uu,i} = A^T A_{rr,i} A \quad => \quad L_{uu} = \begin{bmatrix} L_{uu,1} & & \\ & \ddots & \\ & & L_{uu,5} \end{bmatrix}$$

1.4 Python Benchmark for quadruped action model

The next results are obtained with 5000 trials. There are 16 nodes and the maximum iteration is set to 5. The acceptable limit is 20ms to solve the MPC problem.

Function called	Mean [ms]	Min - Max [ms]
DDP.solve [ms]	1.58440	1.53899 - 2.92778
ShootingProblem.calc [ms]	0.02034	0.01883 - 0.30875
ShootingProblem.calcDiff [ms]	0.02561	0.02384 - 0.30612
UpdateModel [ms]	0.37035	0.35381 - 7.50422