Improving the Solution Speed of the Continuum Stewart-Gough Inverse Kinematics Simulation

1 Setting up the Benchmark

Eventually we want to use our inverse kinematics simulation to control a robot. We need to ensure we are solving the problem fast enough that the robot feels responsive. In this first part, we will setup a benchmark to time how quickly the model solves so that we can measure the effects of changes to the code.

The benchmark we use is from my paper "Efficient Computation of Multiple Coupled Cosserat Rod Models for Real-time Simulation and Control of Parallel Continuum Manipulators". The simulation starts with the end effector at $\boldsymbol{p}_E = \begin{bmatrix}0 & 0.02 & 0.48\end{bmatrix}^T$. Over the course of 100 numerical solutions the robot moves to $\boldsymbol{p}_E = \begin{bmatrix}0 & 0.12 & 0.58\end{bmatrix}^T$, and then returns to the initial position over the course of 100 more solves. The motion is repeated as long as the simulation runs.

The end effector position variable "pE" was previously declared as const, so our first change is to make it mutable (capable of changing state) by declaring it as static instead of const:

```
static Vector3d pE = 0.4*Vector3d::UnitZ();
```

For this scenario the end effector orientation remains flat, so we change the declaration:

```
const Matrix3d RE = Matrix3d::Identity();
```

We also remove the code for plotting and outputting, and the associated variables "p", "px", and "pz" since this is not part of the benchmark. The end effector mass and the density of the rods are neglected. To match the robot from the paper, we also set the Young's modulus to $E = 207\text{GPa}$, set G using a Poisson's ratio of 0.305, set the radius as $rad = 0.0013/2\text{mm}$, and set the number of integration points to $N = 40$.

We will use std::clock_t to measure the time elapsed during the simulation, so we include dependencies for "stdio.h" and "ctime". Then we update the main function to run the benchmark:

```
pE = Vector3d(0, 0.02, 0.48);
guess = solveLevenbergMarquardt<shootingFunction>(guess, 1e-6, 500, 1e-4, 0.5);

//Benchmark speed test
std::clock_t start = std::clock();
const int M = 100;
for(int i = 0; i < M; i++){
        if(i%200 < 100){
                pE(1) += 0.001;
                pE(2) += 0.001;
        }else{
                pE(1) -= 0.001;
                pE(2) -= 0.001;
        }
        guess=solveLevenbergMarquardt<shootingFunction>(guess,1e-6,500,1e-4,0.5);
}
double duration = ( std::clock() - start ) / double(CLOCKS_PER_SEC);
std::cout << "Time elasped: " << duration << "s" << std::endl;
std::cout << "Solution rate: " << M/duration << "Hz" << std::endl;
```

We specify the total number of solutions as $M = 100$. The number of solutions is low right now since the code is slow, but we'll increase it as we go. The guess is updated with each solution since we get better behavior from the shooting method when the initial guess is close to the solution. We have specified additional solver options in the Levenberg-Marquardt algorithm. The second argument is the solver tolerance, which is the acceptable sum-of-squares of the objective function. The third argument is the maximum iterations to use before aborting and throwing an error. The fourth argument is the initial damping of the Levenberg-Marquardt algorithm, and the fifth argument is the adjustment in damping- in this case the damping is halved after a successful step.

Finally we run the code and get the initial benchmark results. On my desktop the above code solves at 50 Hz. This is definitely on the low end of the computational efficiency we would want to teleoperate the robot.

2 Code Optimizations

In this section we'll change some of the functions and calculations to speed up the code while maintaining fundamentally the same mathematical approach.

Although the diagonal stiffness matrix data structures probably have an efficient inverse calculation, we can still improve the solution speed by taking the inverse in the setup phase:

```
const DiagonalMatrix<double, 3> Kse_inv =
        DiagonalMatrix<double, 3>(G*A,G*A,E*A).inverse();
const DiagonalMatrix<double, 3> Kbt_inv =
        DiagonalMatrix<double, 3>(E*I,E*I,G*J).inverse();
```

```
Vector3d v = Kse_inv*R.transpose()*n + Vector3d::UnitZ();
Vector3d u = Kbt_inv*R.transpose()*m;
```

The calculation $\boldsymbol{R\hat{u}}$ contains several unnecessary multiplications by zero. There is a routine to calculate this more efficiently in "commonmath.h" so that we can write:

```
Matrix3d R_s = hat_postmultiply(R,u);
```

Similarly transposing then multiplying wastes effort, so there is a function to do both:

```
Vector3d v = Kse_inv*transposeMultiply(R,n) + Vector3d::UnitZ();
Vector3d u = Kbt_inv*transposeMultiply(R,m);
```

Also on this line, calling "Vector3d::UnitZ()" to increment one index is overkill, so we write:

```
Vector3d v = Kse_inv*transposeMultiply(R,n); v(2) += 1;
```

If we run the program, we can see a noticeable improvement in the benchmark speed. However, we haven't addressed what is probably the slowest part of the ODE function- its function signature:

```
VectorXd cosseratRodOde(VectorXd y)
```

This has us allocating two dynamically sized vectors with each call of the ODE function. In this case it's worth using an output argument. That means we'll change the function signature to

```
void cosseratRodOde(VectorXd& y_s_out, VectorXd& y)
```

Output arguments are often frowned upon, and for good reason since they make code harder to read and reason about, but there is enough of an improvement in the numerical integration "hot loop" to justify the use of one here.

Our method of packing the state vector derivative is also slow, and it is faster to use maps:

```
//Refer to the state vector derivative by its components
Map<Vector3d> p_s (&y_s_out[0]);
Map<Matrix3d> R_s (&y_s_out[3]);
Map<Vector3d> n_s (&y_s_out[12]);
Map<Vector3d> m_s (&y_s_out[15]);

//ODEs
p_s = R*v;
R_s = hat_postmultiply(R,u);
n_s = Vector3d::Zero();
m_s = -p_s.cross(n);
```

Assignments made to the map objects will automatically make changes to y_s_out. Similarly we use maps to unpack the state vector:

```
//Unpack state vector
Matrix3d R = Map<Matrix3d>(&y[3]);
Map<Vector3d> n(&y[12]);
Map<Vector3d> m(&y[15]);
```

It turns out to be preferably to convert $\boldsymbol{R}$ to a matrix, presumably because it is referenced several times.

Now we have specialized some of the calculations to avoid unnecessary work, and we have made some reasonable changes to the program structure to favor speed over readability while keeping the same number of lines of code. On my desktop this revised code solves at 315 Hz.

3 Reducing Number of Numerical Integration Calls

One of the most computationally intensive tasks in the control program is numerically integrating the Cosserat ODEs. It turns out we are doing unnecessary integrations, and we can avoid unnecessary numerical integration calls by writing our own function to calculate the Jacobian.

The Levenberg-Marquardt solver calculates a *Jacobian matrix*, which is defined as

$$\boldsymbol{J} := \frac{\partial \boldsymbol{f}(\boldsymbol{y})}{\partial \boldsymbol{y}},$$

where $\boldsymbol{y}$ is the guess and $\boldsymbol{f}(\boldsymbol{y})$ is the objective function. The solver approximates this partial derivative by a first-order finite difference, that is

$$\boldsymbol{J}_{\text{col i}} \approx \frac{\boldsymbol{f}(\boldsymbol{y} + \boldsymbol{e}_i \Delta) - \boldsymbol{f}(\boldsymbol{y})}{\Delta}$$

for some small increment value Δ. We hold on to the value of $\boldsymbol{f}(\boldsymbol{y})$ after calling the objective function, so to calculate the Jacobian we are calling the objective function 36 times for the 36 columns of the Jacobian. Calling the objective function $\boldsymbol{f}$ requires 6 rod integrations so that a total of 216 numerical integrations are required to find the Jacobian once.

However, the distal variables in rod 1 aren't actually effected when we increment one of the proximal variables of rod 2. Each element of the guess only effects one rod integration, so with a Jacobian calculation specific to our problem, we could use only 36 rod integrations to obtain the Jacobian. Furthermore, the partial derivatives of the distal variables with respect to the arc lengths are already given by the Cosserat ODEs, so we only need 30 rod integrations. More detail is provided in "Efficient Computation of Multiple Coupled Cosserat Rod Models for Real-time Simulation and Control of Parallel Continuum Manipulators".

Although it is slightly labor intensive, we can provide our own function to calculate the Jacobian which is more efficient than brute force finite differencing. The first change we make is to store the distal rod variables globally instead of locally inside the objective function:

```
//Shooting method objective function
static Vector3d pL_shot[6]; //Now a global variable
static Matrix3d RL_shot[6];
static Vector3d nL[6];
static Vector3d mL[6];
const int N = 40;
VectorXd shootingFunction(VectorXd guess){
    VectorXd residual(36);

    Vector3d EF = Vector3d::Zero();
    Vector3d EM = Vector3d::Zero();

    for(int i = 0; i < 6; i++){
        Vector3d n0 = guess.segment<3>(5*i);
        Vector2d m0xy = guess.segment<2>(5*i+3);
        double L = guess(30+i);

        VectorXd y0(18);
        y0 << p0[i], 1, 0, 0, 0, 1, 0, 0, 0, 1, n0, m0xy, 0;

        //Numerically integrate the Cosserat rod equations
        MatrixXd Y = ode4<cosseratRodOde,N>(y0, L);

        pL_shot[i] = Y.block<3,1>(0, N-1);
        RL_shot[i] = Map<Matrix3d>(Y.block<9,1>(3, N-1).data());
        nL[i] = Y.block<3,1>(12, N-1);
        mL[i] = Y.block<3,1>(15, N-1);

        residual.segment<3>(5*i) = pL_shot[i] - (pE + RE*r[i]);
        residual.segment<2>(5*i+3) =
                linear_rotation_error(RL_shot[i],RE).segment<2>(0);

        EF -= nL[i];
        EM -= (mL[i] + (RE*r[i]).cross(nL[i]));
    }
```

We declare static, global variables for the distal values of $\boldsymbol{p}_i(L)$, $\boldsymbol{R}_i(L)$, $\boldsymbol{n}_i(L)$, and $\boldsymbol{m}_i(L)$, so we can access them throughout the "main.cpp" file. These values are stored in arrays so that we can hold the distal states of all the rods.

Now we can write the a function to calculate the Jacobian:

```
const double incr = 1e-8;
void jacobianFunction(MatrixXd& J_out, VectorXd& guess, VectorXd&){
    for(int i = 0; i < 6; i++){
        double L = guess(30+i);
        for(int j = 0; j < 5; j++){
            //A guessed variable is incremented for the finite difference
            double temp = guess(5*i+j);
            guess(5*i+j) += incr;
            Vector3d n0 = Map<Vector3d>(&guess[5*i]);
            Vector2d m0xy = Map<Vector2d>(&guess[5*i+3]);
            guess(5*i+j) = temp;
            VectorXd y0(18);
            y0 << p0[i], 1, 0, 0, 0, 1, 0, 0, 0, 1, n0, m0xy, 0;

            //Numerically integrate the Cosserat rod equations
            VectorXd yf = ode4_endpoint<cosseratRodOde,N>(y0, L);
            Vector3d pL_incr = yf.segment<3>(0);
            Matrix3d RL_incr = Map<Matrix3d>(&yf(3));
            Vector3d nL_incr = yf.segment<3>(12);
            Vector3d mL_incr = yf.segment<3>(15);

            //Jacobian Blocks are partial derivatives of obj. func. equations
            J_out.block<3,1>(5*i, 5*i+j) = (pL_incr - pL_shot[i]) / incr;
            J_out.block<2,1>(5*i+3, 5*i+j) =
              linear_rotation_error((RL_incr-RL_shot[i])/incr, RE).segment<2>(0);
            Vector3d nL_partial = (nL_incr - nL[i])/incr;
            J_out.block<3,1>(30, 5*i+j) = -nL_partial;
            J_out.block<3,1>(33, 5*i+j) = -( (mL_incr - mL[i]) / incr
                                        +(RE*r[i]).cross(nL_partial) );
        }

        //Partial derivatives w.r.t. arc length do not require integration
        VectorXd y_s(18), yL(18);
        yL << pL_shot[i], Map<VectorXd>(RL_shot[i].data(), 9), nL[i], mL[i];
        cosseratRodOde(y_s, yL);

        J_out.block<3,1>(5*i, 30+i) = y_s.segment<3>(0);
        J_out.block<2,1>(5*i+3, 30+i) =
          linear_rotation_error( Map<Matrix3d>(&y_s[3]), RE).segment<2>(0);

        J_out.block<3,1>(30, 30+i) = -y_s.segment<3>(12);
        J_out.block<3,1>(33, 30+i) = -( y_s.segment<3>(15)
                                    +(RE*r[i]).cross(y_s.segment<3>(12)) );
    }
}
```

The structure is similar to the objective function, but we find the partial derivatives of the error for each guessed variable. The solver routine with a user-supplied Jacobian function defined in "convexoptimization.h" is written so that the Jacobian is initialized to zeroes, so we only need to assign the non-zero elements. Confirming the partial derivative equations and structure of the Jacobian is left as an exercise. We update the solver invocation to include the Jacobian function:

```
guess=solveLevenbergMarquardt<shootingFunction,jacobianFunction>
                            (guess, 1e-6, 500, 1e-4, 0.5);
```

We've gone from 216 numerical integration calls required to calculate the Jacobian to only 30 calls. There is one further step we can take to reduce the number of rod integrations. At the start of every solver call, we use the old solution as the initial guess. The result of the

numerical integration does not change between solver calls, so we don't need to integrate on the first objective function evaluation. We add a variable to tell us if we should integrate:

```
//Shooting method objective function
static bool integrate = true;
```

We only integrate if this variable is true, and we set it to true at the end of the objective function:

```
for(int i = 0; i < 6; i++){
        if(integrate){
            Vector3d n0 = guess.segment<3>(5*i);
            Vector2d m0xy = guess.segment<2>(5*i+3);
            double L = guess(30+i);

            VectorXd y0(18);
            y0 << p0[i], 1, 0, 0, 0, 1, 0, 0, 0, 1, n0, m0xy, 0;

            //Numerically integrate the Cosserat rod equations
            MatrixXd Y = ode4<cosseratRodOde,N>(y0, L);

            pL_shot[i] = Y.block<3,1>(0, N-1);
            RL_shot[i] = Map<Matrix3d>(Y.block<9,1>(3, N-1).data());
            nL[i] = Y.block<3,1>(12, N-1);
            mL[i] = Y.block<3,1>(15, N-1);
        }

        ...
}
integrate = true;
```

Finally each pass through the benchmark loop we skip an integration

```
for(int i = 0; i < M; i++){
        integrate = false;
```

This results in another significant speedup. Now the inverse kinematics benchmark solves at about 1930Hz on my desktop. However, we have added 52 to lines to the program, and rather complicated lines at that.

4 Multithreading

I should start this section with a disclaimer that we only achieve a 2x speedup with a significantly more complicated implementation, and the previous performance is likely adequate already. I've opted to keep this section since it may be educational, but it certainly isn't necessary.

Integrating the rods is an *embarrassingly parallel* problem, so we stand to gain from using multiple threads in parallel. We include dependencies on "thread" and "atomic" so that we can implement concurrency. We add a parameter for the number of threads to use:

```
//Independent Parameters
const int num_threads = 6;
```

In addition to the main thread of execution, we will have worker threads that follow the orders of the main thread. The number of workers will be the number of threads minus one. The computational effort of numerically integrating is low compared to the effort of starting up a thread, so we need to maintain an active pool of threads that run continuously by receiving commands. To facilitate giving these orders, we create an enum and an array:

```
enum WORKER_STATE{
    CALCULATING_OBJFUNC,
    CALCULATING_JACOBIAN,
    RESTING,
    TERMINATED
};
static std::atomic<WORKER_STATE> worker_states[num_threads-1];
```

The array of worker states will indicate the current status of each worker, and also allow the main thread to tell the workers to perform a calculation.

Next we create a couple of short functions for the master thread to use:

```
static void waitOnWorkers(){
    for(int i = 0; i < num_threads-1; i++)
        while( worker_states[i]!=RESTING ){ /* Execution is blocked */ }
}

static void setWorkerState(WORKER_STATE command){
    for(int i = 0; i < num_threads-1; i++) worker_states[i] = command;
}
```

The first function allows the main thread to wait until workers are finished with their calculations, at which point they will change their state to "RESTING". The second function allows the main thread to send commands, for example by setting the workers state to "CALCULATING_OBJFUNC". Next we make the guess and the Jacobian global variables:

```
static MatrixXd J_global;
static VectorXd guess_global;
```

This allows us to easily refactor the rod integration into a separate function from the objective function:

```
static void integrateRod(int i){
    Vector3d n0 = Map<Vector3d>(&guess_global[5*i]);
    Vector2d m0xy = Map<Vector2d>(&guess_global[5*i+3]);
    double L = guess_global(30+i);

    VectorXd y0(18);
    y0 << p0[i], 1, 0, 0, 0, 1, 0, 0, 0, 1, n0, m0xy, 0;

    //Numerically integrate the Cosserat rod equations
    MatrixXd Y = ode4<cosseratRodOde,N>(y0, L);

    pL_shot[i] = Y.block<3,1>(0, N-1);
    RL_shot[i] = Map<Matrix3d>(Y.block<9,1>(3, N-1).data());
    nL[i] = Y.block<3,1>(12, N-1);
    mL[i] = Y.block<3,1>(15, N-1);
}
```

Now we can call the "integrateRod" function in parallel from the various threads.

With the new rod integration function, we rewrite the objective function to run in the main thread and outsource the integration calls to workers:

```
static VectorXd shootingFunction(VectorXd guess){
    guess_global = guess;
    VectorXd residual(36);

    if(integrate){
        setWorkerState(CALCULATING_OBJFUNC);
        switch(num_threads){
            case 1: //Do all the work
                for(int i = 0; i < 6; i++) integrateRod(i);
                break;
            case 2: //Do half the work
                for(int i = 3; i < 6; i++) integrateRod(i);
                break;
            case 3: //Do a third of the work
                for(int i = 4; i < 6; i++) integrateRod(i);
                break;
            case 6: //Do a sixth of the work
                integrateRod(5);
                break;
            default:
                std::cout<<"Invalid num_threads: must be divisor of 6"<<std::endl;
                throw(1);
        }
        waitOnWorkers();
    }
    integrate = true;

    Vector3d EF = Vector3d::Zero();
    Vector3d EM = Vector3d::Zero();
    for(int i = 0; i < 6; i++){
        residual.segment<3>(5*i) = pL_shot[i] - (pE + RE*r[i]);
        residual.segment<2>(5*i+3) =
           linear_rotation_error(RL_shot[i],RE).segment<2>(0);
        EF -= nL[i];
        EM -= (mL[i] + (RE*r[i]).cross(nL[i]));
    }
    residual.segment<3>(30) = EF;
    residual.segment<3>(33) = EM;

    return residual;
}
```

The main thread commands the workers to start integrating, then performs its share along with the workers, then waits for all the integrations to finish before calculating the residual from the distal variables.

Now the workers can help to evaluate the objective function, but the Jacobian calculation is still single threaded. We write a function to set the portions the Jacobian corresponding to each leg:

```
static void setJacobianOfLeg(int i){
    for(int j = 0; j < 5; j++){
        double temp = guess_global(5*i+j);
        guess_global(5*i+j) += incr;
        Vector3d n0 = Map<Vector3d>(&guess_global[5*i]);
        Vector2d m0xy = Map<Vector2d>(&guess_global[5*i+3]);
        guess_global(5*i+j) = temp;
        double L = guess_global(30+i);
        VectorXd y0(18);
        y0 << p0[i], 1, 0, 0, 0, 1, 0, 0, 0, 1, n0, m0xy, 0;
        //Numerically integrate the Cosserat rod equations
        VectorXd yf = ode4_endpoint<cosseratRodOde,N>(y0, L);
        Vector3d pL_incr = yf.segment<3>(0);
        Matrix3d RL_incr = Map<Matrix3d>(&yf(3));
        Vector3d nL_incr = yf.segment<3>(12);
        Vector3d mL_incr = yf.segment<3>(15);

        J_global.block<3,1>(5*i, 5*i+j) = (pL_incr - pL_shot[i]) / incr;
        J_global.block<2,1>(5*i+3, 5*i+j) =
           linear_rotation_error( (RL_incr - RL_shot[i])/incr, RE).segment<2>(0);
        Vector3d nL_partial = (nL_incr - nL[i])/incr;
        J_global.block<3,1>(30, 5*i+j) = -nL_partial;
        J_global.block<3,1>(33, 5*i+j) =
           -( (mL_incr - mL[i]) / incr  + (RE*r[i]).cross(nL_partial) );
    }

    VectorXd y_s(18), yL(18);
    yL << pL_shot[i], Map<VectorXd>(RL_shot[i].data(), 9), nL[i], mL[i];
    cosseratRodOde(y_s, yL);

    J_global.block<3,1>(5*i, 30+i) = y_s.segment<3>(0);
    J_global.block<2,1>(5*i+3, 30+i) =
       linear_rotation_error( Map<Matrix3d>(&y_s[3]), RE).segment<2>(0);

    J_global.block<3,1>(30, 30+i) = -y_s.segment<3>(12);
    J_global.block<3,1>(33, 30+i) = -( y_s.segment<3>(15)
                                      +(RE*r[i]).cross(y_s.segment<3>(12)));
}
```

The main thread doles out work in the Jacobian calculation, exactly like the objective function:

```
static void jacobianFunction(MatrixXd& J_out, VectorXd&, VectorXd&){
    setWorkerState(CALCULATING_JACOBIAN);
    switch(num_threads){
        case 1:
            for(int i = 0; i < 6; i++) setJacobianOfLeg(i);
            break;
        case 2:
            for(int i = 3; i < 6; i++) setJacobianOfLeg(i);
            break;
        case 3:
            for(int i = 4; i < 6; i++) setJacobianOfLeg(i);
            break;
        case 6:
            setJacobianOfLeg(5);
    }
    waitOnWorkers();
    J_out = J_global;
}
```

Finally we write the function which the workers will run to respond to commands:

```
static void workerFunction(int id){
    while( worker_states[id]!=TERMINATED ){
        if( worker_states[id]==CALCULATING_OBJFUNC ){
            switch(num_threads){
                case 2:
                    integrateRod(0);
                    integrateRod(1);
                    integrateRod(2);
                    break;
                case 3:
                    integrateRod(2*id);
                    integrateRod(2*id+1);
                    break;
                case 6:
                    integrateRod(id);
                    break;
            }
            worker_states[id] = RESTING;
        }

        if( worker_states[id]==CALCULATING_JACOBIAN ){
            switch(num_threads){
                case 2:
                    setJacobianOfLeg(0);
                    setJacobianOfLeg(1);
                    setJacobianOfLeg(2);
                    break;
                case 3:
                    setJacobianOfLeg(2*id);
                    setJacobianOfLeg(2*id+1);
                    break;
                case 6:
                    setJacobianOfLeg(id);
                    break;
            }
            worker_states[id] = RESTING;
        }
    }
}
```

It's just a continuous loop of checking for work, performing any tasks and reporting completion. Finally in the main script we can start our worker pool:

```
int main(int, char**){
    std::thread workers[num_threads-1];
    setWorkerState(RESTING);
    for(int i = 0; i < num_threads-1; i++)
        workers[i] = std::thread(workerFunction, i);
```

At the end of the main method, we shutdown the worker pool to make sure that the program terminates gracefully:

```
setWorkerState(TERMINATED);
for(int i = 0; i < num_threads-1; i++) workers[i].join();

return 0;
```

With two threads the model solves at around 3500 Hz, with three around 4600 Hz, and six is slightly slower than three. Of course by the time you reach six threads, you'll probably struggle to have enough cores to run them in parallel on a single PC.

5 Over-optimized Custom Integration Routine

The single-threaded integration code is probably good enough in most circumstances. However, I think it's interesting to see how much more performance we can gain if we're willing to over-optimize the numerical integration code. In the fifth example code folder, we write a function which integrates the Cosserat rod ODEs using a hard-coded fourth-order Runge-Kutta method, with calculations written on an element by element basis.

This code is an unmaintainable mess, but it achieves a performance of about 4200Hz on the benchmark. One interesting aspect is that the tradeoff between readability and performance is not a fundamental issue; the optimizing compiler helps us to ease the burden of this tradeoff, and an optimizing compiler could potentially achieve the performance we see here starting from readable code. However, the C++ standard is fairly conservative in terms of what code transformations result in equivalent code, and the compiler probably isn't allowed to make all the changes we've made. But that's a tangent from robotics.

6 Conclusion

Now we can solve the continuum Stewart-Gough inverse kinematics at rates that are easily fast enough to control the robot. Some simple changes to the code in section 2 resulted in roughly a 6x speedup, and a user-supplied Jacobian calculation in section 3 resulted in another 6x speedup. The multithreading and over-optimized integration code in sections 4 and 5 resulted in roughly a 2x speedup each, which probably doesn't justify the added complexity, but hopefully it is useful to see these modifications.