3.2 Line Search

The topic of approximately optimizing with respect to a single variable in the context of a higher dimensional problem falls under the topic of *line search*. The "line" in the name refers to the possibility of optimizing along any line in the parameter space, not just parallel to the coordinate axes as we do in Cyclic Coordinate Descent.

We generally set up line search with a current guess $\vec{\theta}$ and a *search direction* vector $\vec{p}$ (which in Cyclic Coordinate Descent will just be one of the axes: $\vec{p} = (1, 0, \ldots, 0)$ for example), then approximately solve for a *step length* α:

$$\min_{\alpha} f(\vec{\theta} + \alpha\vec{p}). \tag{28}$$

Phrasing it this way will let us re-use line search for other optimization algorithms that use non-axis-aligned search directions.

Our general goal is to find a step length α that is reasonably close to the optimal step length, and in particular will advance us towards the final solution. This is by no means a simple property to guarantee without more knowledge of, or restrictions on, the objective function f. Many line search methods make use of the derivative of f, for example, to figure out what a sufficiently good reduction in $f(x)$ is to guarantee eventual convergence.

We'll look at a simple and reasonably effective line search method which doesn't use the derivative, with the caveat that for some badly-behaved objective functions it may well fail to give adequate improvement. In fact, even with a perfect line search method, Cyclic Coordinate Descent is known to fail on certain classes of objective functions; we'll rely on the IK problems we encounter to be well-behaved enough that this isn't an issue.

Our simplest strategy is *backtracking*. We will start off with an initial step size α_0. It might be a sensible default derived from the nature of the problem, such as $30°$ for an angle or half the length of a limb for a translation; it could also be the last successful step size for this variable from a previous line search.

If $f(\vec{\theta} + \alpha_0 \vec{p} \geq f(\vec{\theta})$, we conclude the step is too large, and scale it back by some constant τ such as $1/2$: $\alpha_1 = \tau\alpha_0$, and try again until we find a step size that reduces the value of f—or give up after a certain number of iterations because the step size has become small enough that we can conclude we are already at a minimum.

On the other hand, if the first step already provides some reduction, $f(\vec{\theta} + \alpha_0 \vec{p} < f(\vec{\theta})$, it's worth exploring if a larger step size is possible (to get us to the minimum faster). We thus try scaling up the step size by a similar constant such as 2: $\alpha_1 = \alpha_0/\tau$. As long as we continue to find even smaller values of f we keep going; as soon as we find a step with a larger f than the previous step, we stop and use the previous step.

This must work as long as $\vec{p}$ is a "descent direction", i.e. that the derivative of f along $\vec{p}$ at θ is negative, so locally f is decresing in this direction. In particular, this means that once the step size has been scaled down small enough, we have to get some reduction in f. For Cyclic Coordinate Descent, however, we don't know this: it could be that f is increasing along $\vec{p}$. In that case we should interleave the search along $\vec{p}$ with a search along the opposite direction $-\vec{p}$ (or equivalently with negative step sizes) until we do find a reduction in f (or give up after a fixed number of iterations, concluding we are already at a local minimum).

There are many ways this can be potentially improved. For example, the backtracking search can be extended to arrive at a "bracketing" of a minimum, finding three step sizes where the value of f is smallest at the middle of the three. Assuming f is smooth and these steps are close together, f should be accurately approximated by fitting a quadratic through those three points; finding where the minimum of that quadratic lies gives hopefully an even better step size. (Indeed, if f happens to be a quadratic polynomial, this lets us jump to the exact answer.)

3.3 When to Stop

How many iterations of Cyclic Coordinate Descent, or any other method, are enough?

From a practical perspective, we probably should always have a maximum number of iterations allowed. If convergence is particularly slow for some unknown reason, it's generally better to stop early and alert the user to the problem rather than hang for some arbitrary long time, or even indefinitely. This might indicate the problem is ill-posed, for example, or that the IK solve is jammed in some degenerate configuration. The user or a higher level routine could then "nudge" the solve, perhaps, by adding a random perturbation to the initial guess and trying again.

Ideally, however, we will arrive at or close enough to a minimum much faster than that, and should be able to quit early. For Cyclic Coordinate Descent, if all the line searches in a sweep report no progress was possible, we clearly are at an impasse and should stop; also if all the final step sizes in a sweep are below some threshold (e.g. the point where there is no visual change in the solution) we are probably as far as we can get.

Stopping at a point where the algorithm can no longer progress doesn't guarantee we are at a minimum unfortunately. For example the function $f(x,y) = xy + \frac{1}{4}(x^4 + y^4)$ is zero at the origin $(x,y) = (0,0)$, and only increases along the x and y axes from the origin. However, the actual minimum of $-\frac{1}{2}$ occurs at $(1,-1)$ and $(-1,1)$. Cyclic Coordinate Descent would be stuck at the origin, which is a *stationary point* (the first derivatives are zero) but not a minimum—though a random perturbation would get it unstuck again.

Even if we do converge to a *local minimum*, where f does not decrease in any direction, it is not guaranteed to be a *global minimum*, where f takes on its actual minimum value.

Having said that, guaranteeing arrival at a global minimum, and not getting stuck at a non-minimum stationary point, is in general tough and almost certainly requires more information about f such as it being "convex". For the purposes of IK, we will be content with some combination of testing out different objective functions (penalties and regularizations), occasional random perturbation, and user intervention in particularly difficult scenarios.

3.4 Least Squares and Gauss-Newton

While Cyclic Coordinate Descent with backtracking is a simple and practical method for many small problems, it can be slow to converge. Consider this problem, for example:

$$\min_{\theta_1,\theta_2} 5(\theta_1+\theta_2-1)^2+(\theta_1-\theta_2-3)^2. \tag{29}$$

Here the main "valley" of f is in the direction $(1,1)$, at a 45° angle from the coordinate axes. Cyclic Coordinate Descent steadily zigzags along the axes towards the minimum, in this case, instead of going along the main downhill direction.

In this case, the objective is just a quadratic, which means its gradient is a linear function:

$$\begin{aligned} \frac{\partial}{\partial\theta_1}: &\qquad 10(\theta_1+\theta_2-1)+2(\theta_1-\theta_2-3), \\ \frac{\partial}{\partial\theta_2}: &\qquad 10(\theta_1+\theta_2-1)-2(\theta_1-\theta_2-3). \end{aligned} \tag{30}$$

Finding where the gradient is zero amounts to solving a simple linear system of equations:

$$\begin{aligned} 12\theta_1+8\theta_2&=16, \\ 8\theta_1+12\theta_2&=4. \end{aligned} \tag{31}$$

The exact solution is just $(2,-1)$. If we had code which could "see" the coefficients of the quadratic, it could jump straight to the minimum without iterating at all.

Obviously some of the functions we want to optimize aren't just quadratics, so we can't generally provide coefficients for the code to work with. However, for the sake of generating a good guess at the minimum we can **approximate** f with a quadratic "model function", and minimize the model instead of f. This is the essence of Newton's method, probably the single most important algorithm in optimization.

Newton's method itself requires knowing both the derivative and second derivatives of f to form a quadratic Taylor series to approximate the function near a guess, which is then minimized to find the next guess. In our IK examples, we can get away with only needing first derivatives by using the least-squares structure of our problems. This leads to what is known as the *Gauss-Newton* method.

Recall our least-squares form is

$$\min_{\vec{\theta}} \|\vec{r}(\vec{\theta})\|^2, \tag{32}$$

where $\vec{r}(\vec{\theta})$ is the "residual" function. We're going to use the derivative of $\vec{r}$, called the *Jacobian*. Suppose $\vec{r}$ is an m-dimensional vector function, and θ is an n-dimensional vector. The Jacobian J is then an $m \times n$ matrix-valued function:

$$J(\vec{\theta}) = \begin{pmatrix} \frac{\partial r_1}{\partial \theta_1} & \frac{\partial r_1}{\partial \theta_2} & \cdots & \frac{\partial r_1}{\partial \theta_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial r_m}{\partial \theta_1} & \frac{\partial r_m}{\partial \theta_2} & \cdots & \frac{\partial r_m}{\partial \theta_n} \end{pmatrix}. \tag{33}$$

Although it's harder to visualize in multiple dimensions, the Jacobian is just a generalization of the "slope" of a function. Just like we can approximate a smooth function around a point with a line tangent to the function—i.e. a line with slope equal to the derivative—we can approximate $\vec{r}(\theta)$ near a point using a linear function with the Jacobian:

$$\vec{r}(\vec{\theta} + d\vec{\theta}) \approx \vec{r}(\vec{\theta}) + J(\vec{\theta})d\vec{\theta}. \tag{34}$$

Taylor's theorem tells us this is accurate to within $O(\|d\vec{\theta}\|^2)$.

If we're at guess θ, we can compute the Jacobian and approximate $\vec{r}$ with this. Plugging this approximation into the minimization gives a least-squares problem again:

$$\min_{d\vec{\theta}} \left\| \vec{r}(\vec{\theta}) + J(\vec{\theta}) d\vec{\theta} \right\|^2 . \tag{35}$$

The crucial improvement is that here θ is fixed—it doesn't enter into the minimization. The new approximate residual is linear in the variable $d\vec{\theta}$, so this is a linear least-squares problem: the objective is indeed just a quadratic, so we can solve it exactly. Our new guess is

$$\vec{\theta}_{\text{new}} = \theta + d\vec{\theta}. \tag{36}$$

We can continue with the next iteration from this point, recomputing the Jacobian etc., and stopping when the iterates no longer change significantly.

How exactly do we solve this easier subproblem? There are several approaches of varying sophistication and complexity, but the simplest is called the *normal equations* approach. For a column vector x the norm squared $\|x\|^2$ is just $x^T x$ using the transpose, so

$$\begin{aligned} \left\| \vec{r}(\vec{\theta}) + J(\vec{\theta}) d\vec{\theta} \right\|^2 &= \left(\vec{r}(\vec{\theta}) + J(\vec{\theta}) d\vec{\theta} \right)^T \left(\vec{r}(\vec{\theta}) + J(\vec{\theta}) d\vec{\theta} \right) \\ &= \vec{r}(\vec{\theta})^T \vec{r}(\vec{\theta}) + 2 d\vec{\theta}^T J(\vec{\theta})^T \vec{r}(\vec{\theta}) + d\vec{\theta}^T J(\vec{\theta})^T J(\vec{\theta}) d\vec{\theta}. \end{aligned} \tag{37}$$

Simplifying this, with $\vec{r}$ instead of $\vec{r}(\vec{\theta})$ and J instead of $J(\vec{\theta})$, finally leads to:

$$\|\vec{r}\|^2 + 2 d\vec{\theta}^T J^T \vec{r} + d\vec{\theta}^T J^T J d\vec{\theta}. \tag{38}$$

To find where this is at a minimum, we can take the gradient with respect to $d\vec{\theta}$:

$$2 J^T \vec{r} + 2 J^T J d\vec{\theta}. \tag{39}$$

(If you're not used to working with vector calculus and linear algebra, this step might be a bit hard to take in one go, but you can verify it by expanding out the matrix and vector products in terms of sums over indices, then differentiating with respect to each entry individually.) Once we have the gradient, we can solve for when it is zero:

$$(J^T J) d\vec{\theta} = -J^T \vec{r}. \tag{40}$$

The square matrix $J^T J$ is naturally symmetric, and can be efficiently solved with the Cholesky factorization for example (see a numerical linear algebra reference such as Golub and van Loan's book [GVL96], and standard software libraries such as LAPACK [ABB$^+$99]).

Basic Gauss-Newton can be shown to converge rapidly to the solution if the initial guess is close enough, but it can run into troubles if the initial guess is too far away. Therefore it's a good idea to combine Gauss-Newton with a line search routine. If we choose the search direction to be the Gauss-Newton step itself, $\vec{p} = d\vec{\theta}$, then it's not difficult to prove it must be a descent direction; the natural first guess at a step size should be $\alpha = 1$ (which is perfect if the problem is actually linear least-squares). The combination of Gauss-Newton and line search, applied to problems that have been appropriately regularized as discussed earlier, is extremely robust and generally works very well.

3.5 Computing Derivatives

The main difficulty for Gauss-Newton tends to be the calculation of the derivatives in the Jacobian matrix. In principle this is a simple affair, of course, but writing the code that can do it for a FK hierarchy constructed interactively by a user is a bit daunting at first glance. Let's look at it in detail.

Consider a point on some bone in a FK hierarchy. Suppose its coordinates in the bone's local coordinate system are $\vec{p}$. The FK tree traversal gives us the world space coordinates $\vec{x}$ as a product of transformations applied to $\vec{p}$:

$$\vec{x} = T_0(\theta_0)T_1(\theta_1)\dots T_n(\theta_n)\vec{p}. \tag{41}$$

Here I have separated out multi-parameter joints, and the transformation of the FK root with respect to world space, into products of single parameter transformations. All joint parameters are labeled with θ, both rotations and translations along various axes. For simplicity of notation I've also ignored non-parameterized transformations (like fixed rotations or translations that define where a joint is on a bone, and should never be changed for regular motion) as transformations with a single parameter—you could also view them as parameterized transformations in the list with a dummy parameter which we ignore in subsequent processing. Obviously the full differentiation code will need to multiply them in as appropriate.

In this form, the different partial derivatives of the Jacobian are refreshingly simple:

$$\frac{\partial \vec{x}}{\partial \theta_i} = T_0(\theta_0) \cdots T_{i-1}(\theta_{i-1}) \frac{\partial T_i(\theta_i)}{\partial \theta_i} T_{i+1}(\theta_{i+1}) \ldots T_n(\theta_n)\vec{p}. \tag{42}$$

This means that for column i of the Jacobian, $i = 1, \ldots, n$, we simply use the regular FK formula but with the i'th transformation $T_i(\theta_i)$ replaced by its partial derivative with respect to θ_i.

The derivative of a single transformation matrix is generally pretty easy. For example, here's a matrix which translates along the x-axis:

$$T(\theta) = \begin{pmatrix} 1 & 0 & 0 & \theta \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{43}$$

Its derivative is just:

$$\frac{\partial T(\theta)}{\partial \theta} = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}. \tag{44}$$

Here's a matrix which rotates around the z-axis:

$$R(\theta) = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{45}$$

Its derivative is just:

$$\frac{\partial T(\theta)}{\partial \theta} = \begin{pmatrix} -\sin(\theta) & -\cos(\theta) & 0 & 0 \\ \cos(\theta) & -\sin(\theta) & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}. \tag{46}$$

Note that in both cases, the derivative of the transformation matrix has a lot of zeros, including along the bottom row: these aren't regular affine transformations any more. When you apply them to a point, which normally has a fourth (homogeneous) coordinate of one, you get a *vector*, with fourth coordinate equal to zero. You shouldn't normalize the result—simply drop the zero fourth coordinate to get the derivative vector in 3D coordinates. This makes sense since the derivative of a point is a direction it moves in when a parameter is adjusted, not an actual location or point itself.

With this in mind, we could easily compute each column of the Jacobian independently. However, it's worth realizing that any two columns actually share a lot of factors: columns i and j only differ at T_i and T_j. It is inefficient to recompute all those products of transforms over and over. A useful mental challenge is to write a more efficient algorithm for evaluating the Jacobian that shares the computation of intermediate products between all columns (hint: in the first pass over all columns, compute only the $T_{i+1} \cdots T_n \vec{p}$ final part of the answer for column i; in a second pass multiply in the derivatives for all columns; in a third pass complete the calculation of all the columns).