Introduction to Direct collocation

1 Optimization in motor learning

Let's begin by framing the issue of optimization in a biomechanical/motor control framework. Consider the task of baseball throwing (a personal favourite of mine and one that has grabbed some spotlight recently [1]). Apparently humans are really good at this relative to our nearest phylogenetic neighbours. To do this task best, the nervous system needs to figure out how to turn on your arm muscles in the right way to accelerate the hand (and ball) to maximal speed, while releasing the ball and striking a target within some range of acceptible variation. As scientists we might like to model this motion to gain insight into our own functional morphology, perhaps allowing us to predict what a movement would look like if it were optimizing some cost.

We might phrase this problem by stating that our movement has

- an objective, or cost function which we will call f
- constraints that represent physiology and the desired behaviour, g

Older ways of solving these problems would involve simulating the behaviour. We could solve many initial value problems where an ordinary differential equation solver (like ode45) allows us to simulate a movement given some initial value of the body, and some input control signals. To grade how well that simulation did, one might use the simulation data to extract how well the objective was met (how fast the ball was thrown), and dock the simulation points if it violated any constraints in any way.

Today the approach will be to avoid running a simulation and also integrate our constraints in a useful way. To understand the direct collocation approach we need to refresh or introduce only 4 main ideas and put them together:

- Newtons method for solving equations.
- Unconstrained Optimization.

- Lagrange multipliers.
- Collocation.

2 Newton's method for solving equations

We begin by talking about Newton's method for finding the solution to a system of equations. Example: The roots (value of x where $f(x) = 0$) of the equation

$$x_1^2 - 4 = 0 \tag{1}$$

are just

$$x_1 = \pm 2 \tag{2}$$

We can also solve a system of equations:

$$x_1 + x_2^2 = 0 \tag{3}$$

$$2x_2 + 4 = 0 \tag{4}$$

which is, in Matlab:

```
% the variables:
x1=sym('x1','real');
x2=sym('x2','real');
% the equations:
eq1 = sym('x1+x2^2=0');
eq2 = sym('2*x2+4=0');
%%
[x1sol,x2sol]=solve(eq1,eq2)
```

which produces the output $x1sol = -4$ and $x2sol = 2$.

More generally, finding the roots of complex functions can be difficult or impossible analytically. In those cases we avoid doing the algebra to solve for x, and we use local approximations of a function to arrive at solutions to equations over several steps. Newton's method is such a method, and is nicely visualized on wikipedia for the one-dimensional case. The idea is to keep make simple approximations of the equations, find the solution to those approximations, until we are as close as we want to a solution to the function. In words/pseudocode, this is:

```
Begin with a random or guess $x$, and compute $f(x)$;
while $f(x) \neq 0$ do
    approximate the equations near $f(x)$ by a line;
    set $x$ to be the root of the line;
end
```

We can implement this to solve our example system above. To do so, we will introduce some symbols that will be helpful throughout the document. We are going to use a simple convention that $\boldsymbol{b}$old font denotes vectors (i.e. $\boldsymbol{f}$ is n by 1) and $\boldsymbol{B}$old capitals refers to a matrix (i.e. $\boldsymbol{J}$ is a matrix). In our toy example we had two equations that equal 0. We now denote the set of functions as a vector, and denote it $\boldsymbol{f}(\boldsymbol{x})$.

Newton's method means that we linearly approximate the equations $\boldsymbol{f}$. To do so we use the Jacobian $\boldsymbol{J}$, which just computes the partial derivatives of each equation in our system of equations. We make a linear approximation (also called a first-order Taylor series approximation) of the function f near $\boldsymbol{x_c}$:

$$\boldsymbol{f}(\boldsymbol{x}) \approx \boldsymbol{f}(\boldsymbol{x_c}) + \boldsymbol{J}(\boldsymbol{x_c})\boldsymbol{\Delta x} \tag{5}$$

$$\tag{6}$$

by convention, the Jacobian is organized the following way:

$$\boldsymbol{J} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \tag{7}$$

i.e. the ith row contains the partial derivatives of the ith equation.

Since we want to solve where this approximation $\boldsymbol{f}(\boldsymbol{x})$ is zero, we solve equation 5 for $\boldsymbol{\Delta x}$:

$$\Delta x_c = -\boldsymbol{J}(\boldsymbol{x_c})^{-1}\boldsymbol{f}(\boldsymbol{x}) \tag{8}$$

By adding Δx_c to x, we have the root of equation 5, i.e. the linear approximation to our original function. To proceed to our true solution we follow our algorithm by updating x and repeating until we are close enough:

$$x_{i+1} = x_i + \Delta x_i = x_i - \boldsymbol{J(x_i)}^{-1} f_{x_i} \tag{9}$$

We will now take a look at an example root finding exercise.

```
% the variables:
x1=sym('x1','real');
x2=sym('x2','real');
% the equations:
eq1 = sym('x1+x2^2');
eq2 = sym('-2*x2-4*x1 -5');
f=sym(zeros(2,1));
f(1,1)=eq1;
f(2,1)=eq2;
a=ezplot(eq1==0);hold on;
b = ezplot(eq2==0);
% %
set(a,'color','b');
set(b,'color','r');
legend({char(eq1),char(eq2)});
title('');
%%
[solx1,solx2]=solve(eq1==0,eq2==0)
%%
J=sym(zeros(2,2));
J(1,1)=diff(eq1,x1);
J(1,2) = diff(eq1,x2);
J(2,1)=diff(eq2,x1);
J(2,2) = diff(eq2,x2);
invJ = inv(J);
%%
xt=sym(zeros(2,1));
xt(1) = -2 + 4*rand(1);
xt(2) = -2 + 4*rand(1);
dx=sym([1;1]);
subs(dx,1);
spots=[];
```

```
counter = 1;

spots(:,counter) = xt;

thresh=1e-6;

while double(norm(dx))>thresh

    dx=-subs(invJ,{x1,x2},{xt(1),xt(2)})*subs(f,{x1,x2},{xt(1),xt(2)});

    hold on;

    text(double(xt(1)),double(xt(2)),num2str(counter));

    xt = xt+dx;

    counter=counter+1;

end;
```

3 Unconstrained optimization

A value of $\boldsymbol{x}$ that minimizes a continous function f (note f is not bold, meaning we are in this case concerning ourselves with a scalar multivariate function f) occurs where the derivative of f is zero. Thus, finding an optimal solution of a multivariate function f is equivalent to finding roots of the *gradient* of f. We write this f as

$$\nabla f(\boldsymbol{x}) = \mathbf{0} \tag{10}$$

Where ∇f is used to denote the *gradient* of f: a vector pointing in the direction of greatest increase in the value of f. The output of ∇ is n by 1, where n is the number of variables. Thus you notice that we do not bold this symbol only because it is redundant (this is a choice and it's maybe not clear that this is the best syntax). In the below figure, The gradient vectors for the function $f(\boldsymbol{x}) = x_1^2 + x_2^2$ are plotted in red arrows.

Closely related to the concept of gradient is the *level set*. In black lines are plotted the level sets of f. A level set denotes contour lines of f, i.e. places where values of f are equal. Importantly, the gradient vector at a particular point is perpendicular to the level set at that point.

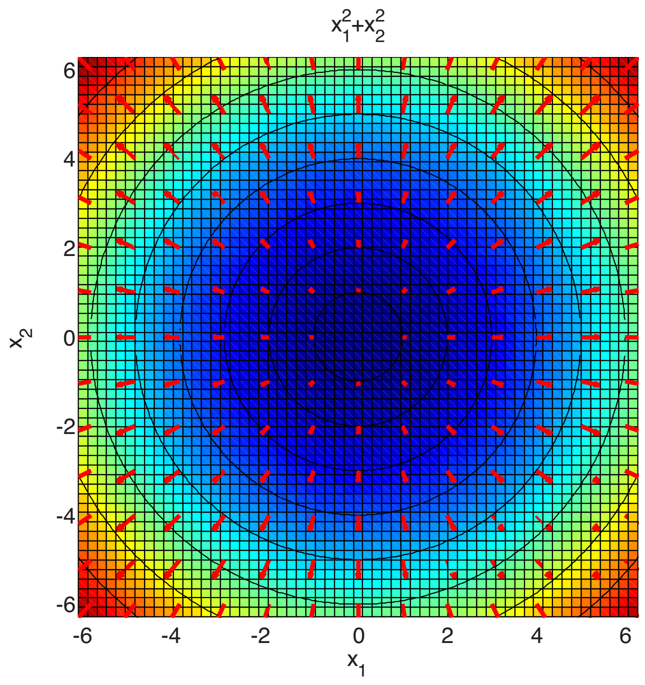

Figure 1. Plot of $x_1^2 + x_2^2$. Gradient vectors ∇f are potted in red arrows; level sets of f are plotted in black lines.

In the case where again an algebraic solution is difficult or impossible, we use the iterative approach to finding the value of $\boldsymbol{x}$ that is a root of ∇f. Here to approximate ∇f at a current $\boldsymbol{x_c}$:

$$\nabla f(x_c) \approx \nabla f(x_c) + \boldsymbol{H}\boldsymbol{\Delta x} \tag{11}$$

$\boldsymbol{H}$ is called the Hessian and is the partial derivatives of ∇f, or equivalently the second partial derivatives of $\boldsymbol{f}$:

$$\boldsymbol{H} = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \cdots & \frac{\partial^2 f}{\partial {x_n}^2} \end{bmatrix} \tag{12}$$

We can now iteratively find $\boldsymbol{x}$ in equation 11 by continuing to adjust by Δx until we have a solution of ∇f:

$$\boldsymbol{\Delta x} = -\boldsymbol{H}^{-1}(\boldsymbol{x})\nabla f(\boldsymbol{x_c}) \tag{13}$$

so that on each iteration,

$$x_{i+1} = x_i + \Delta x_i = x_i - H(x_i)^{-1}\nabla f(x_i) \tag{14}$$

This method allows us to find critical points (maxima, minima, inflection points) of f, but does not differentiate between them. From single-variable calculus we might recall that the critical point was a minimum if the second derivative was positive. Similarly, we know that the we have found a minimum of our multivariate function if

$$\boldsymbol{v^T H v} > 0 \tag{15}$$

This occurs when the eigenvalues of $\boldsymbol{H}$ are all positive. In this case, the matrix $\boldsymbol{H}$ is positive definite.

We'll now take a look at the very simple example of minimizing $x_1^2 + x_2^2$.

```
% the variables:
% x1=sym('x1','real');
% x2=sym('x2','real');
clear all;close all;
syms x1 x2
% the equations:
eqF = sym('x1^2+x2^2');
% eqF = sym('2*x1^2 ? 3*x1 + .25 *x2^2 + x2');
ezsurf(eqF);

% set(S,'facealpha',.5);

%%
gradF = sym(zeros(2,1));
gradF(1) = diff(eqF,x1);
gradF(2) = diff(eqF,x2);
%%
H=sym(zeros(2,2));
H(1,1)=diff(gradF(1),x1);
H(1,2) = diff(gradF(1),x2);
H(2,1)=diff(gradF(2),x1);
H(2,2) = diff(gradF(2),x2);
invH = inv(H);



%%
xt=sym(zeros(2,1));
xt(1) = -5
xt(2) = 5
dx=sym([1;1]);
subs(dx,1);
```

```
spots=[];
counter = 1;
spots(:,counter) = xt;
thresh=1e-6;
%% Newton's method
while double(norm(dx))>thresh
    dx=-subs(invH,{x1,x2},{xt(1),xt(2)})*subs(gradF,{x1,x2},{xt(1),xt(2)});
    hold on;
    text(double(xt(1)),double(xt(2)),double(subs(eqF,{x1,x2},{xt(1),xt(2)})),num2str(counter-1),'fontsize',20);
    xt = xt+dx;
    counter=counter+1;
end;
view(2);
```

4 Lagrange multipliers

We first reviewed finding roots of equations using Newton's method to solve equations, and then applied this approach to finding an optimal minimum value of a multivariate function. We now consider optimizing a function where we also have some set of constraints. We are therefore concerned with finding

$$\min f(x) \tag{16}$$

where

$$g(x) = 0 \tag{17}$$

We'll consider a visualizable example and stick with the above quadratic equation we used in the above section. Consider a 2-dimensional vector x, $[x_1, x_2]$. We will minimize

$$norm(x) \tag{18}$$

where

$$x_1x_2 - 3 = 0 \tag{19}$$

We can visualize this in the following plot. Again, the color reflects the value of $x_1^2 + x_2^2$ (warmer colors are larger) and here the black lines denote the constraint equation. Of course we can see right away the solution in the unconstrained case that the solution would be $[0, 0]$; but while we can see visually that the solution to the constrained case is near $\pm[1.5, 1.5]$), it's hard to say exactly what it is.

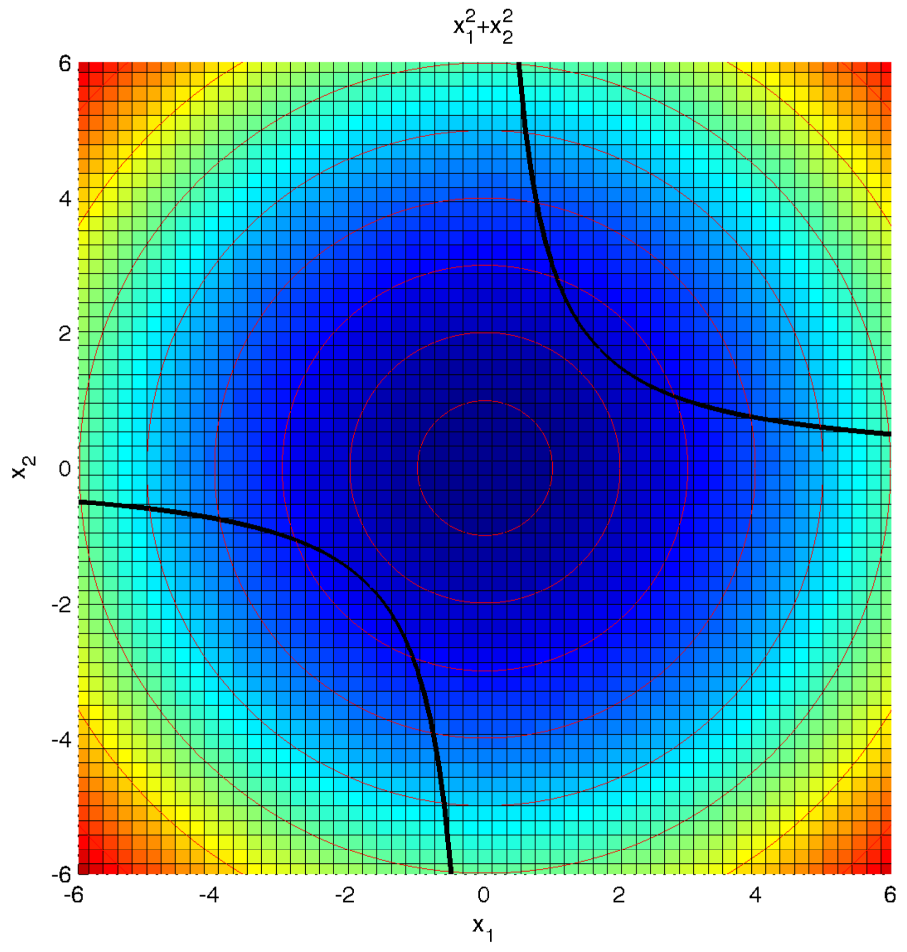

Figure 2. Plot of $x_1^2 + x_2^2$, and the constraint equation $x_1 x_2 - 3 = 0$

Let's consider the constrained minimum $\boldsymbol{x_{cm}}$. First, we know that since this is a constrained minimum, if we move along the constraint (i.e. along the black line), the derivative of f must be zero. This means that the gradient of f, ∇f must be perpendicular to the constraint equation $g((x)) = 0$ at x_{cm}.

If we step back and think about $g(\boldsymbol{x}) = 0$, we realize that we can think about this equality constraint $g(\boldsymbol{x}) = 0$ as just a level set of the multivariate function $g(\boldsymbol{x})$. The gradient of g, ∇g is by definition perpendicular to the level set of g. Therefore, the gradient of f, and the gradient of g, are parallel to each other at the constrained minimum. This means

$$\nabla f = \nabla g \lambda \tag{20}$$

Where λ is in this case a scalar (1 x 1) and reflects the fact that ∇f and ∇g are parallel but not (in general) of equal length. In our above example where we are optimizing for f given g, equation 20 gives us two pieces of information and thus two more equations (because our gradients in this case have two components). We need 3 equations to solve this system because we have three unknowns —$(x_1, x_2, \lambda$ —and so we need a third equation, which is the constraint equation $g(x) = 0$. Thus knowing our observed fact about the gradient of f and g and the constraint equation is enough information to solve the system.

We can write these facts that are all we need to solve a constrained optimzation in a compact way. We define the Lagrangian, $L(x, \lambda)$, as

$$L(\boldsymbol{x}, \boldsymbol{\lambda}) = f(\boldsymbol{x}) - \lambda^T \boldsymbol{g}(\boldsymbol{x}) \tag{21}$$

and necessary conditions for the variables (x, λ) to define a constrained minimum are that the gradient w/r/t x and λ are equal to 0.

$$\nabla_x L = \mathbf{0}, \tag{22}$$

$$\nabla_\lambda L = \mathbf{0}. \tag{23}$$

Note that this is the first time we are introducing subscritps into our syntax. These denote that the gradient vector should include terms only with respect to the variable denoted in the subscript. Then computing the gradient of L with x gives us:

$$\nabla_x L = \nabla f - \nabla g \lambda \tag{24}$$

which we write slightly differently to be more general and handle the case where we have more than a single constraint (unlike our current example):

$$\nabla_x L = \nabla f - \boldsymbol{J_g^T} \lambda \tag{25}$$

where $\boldsymbol{J_g}$ is the Jacobian of the constraint equations. You may wonder where the transpose comes from. If you think about the way that ∇f is defined to be arranged, as an $n \times 1$ column vector where each row is the partial derivative with respect to a variable, then it is clear that the Jacobian needs to be rotated to match. Normally in the Jacobian, each row represents a different equation, and the jth row in each column represents the partial derivative of that equation with respect to variable j.

and the gradient of L with *lambda*:

$$\nabla_\lambda L = \boldsymbol{g}(\boldsymbol{x}) \tag{26}$$

As before in section 3, this method does not tell us which of the three types of critical points we have found. We might guess that some kind of second-derivative test will be useful and that intuition is correct: the *Hessian of the Lagrangian* is defined as

$$\boldsymbol{H_L} = \nabla^2_{xx} L = \nabla^2_{xx} F - \sum_{i=1}^{m} \lambda_i \nabla^2 g_i \tag{27}$$

$$\boldsymbol{H_L} = \boldsymbol{H_f} - \lambda \boldsymbol{H_g} \tag{28}$$

(what you can see from this definition is that we do not care about the second derivative information of the multipliers) and the sufficient condition for a minima is that

$$\boldsymbol{v^T} \boldsymbol{H_L} \boldsymbol{v} > 0 \tag{29}$$

for $\boldsymbol{v}$ in the constraint tangent space. This condition for finding minima is thus just a bit different than the above condition for unconstrained optimization, since in 29 the curvature must only be positive in the constraint space.

5 Lagrange multipliers with Newton's method

We can apply Newton's method now to iteratively finding x and λ. We proceed from equation 23:

$$\nabla_x L = \nabla f - \boldsymbol{J_g^T}\lambda \tag{30}$$

$$\nabla_x L \approx \nabla f(\boldsymbol{x_c}) + \boldsymbol{H_L}\Delta\boldsymbol{x} - (\boldsymbol{J_g^T} + \boldsymbol{J_g^T}\Delta\boldsymbol{\lambda}) \tag{31}$$

and we note that since $\Delta\lambda = \lambda_{new} - \lambda$,

$$\tag{32}$$

$$\nabla_x L = \nabla f(\boldsymbol{x_c}) + \boldsymbol{H_L}\Delta\boldsymbol{x} - \boldsymbol{J_g}^T\boldsymbol{\lambda_{new}} \tag{33}$$

The second part of our Lagrangian is just the gradient of L with λ

$$\nabla_\lambda L = -\boldsymbol{g} - \boldsymbol{J_g}\boldsymbol{\Delta x} \tag{34}$$

$$\tag{35}$$

Since the conditions for constrained optimality are $\nabla_x L = 0$ and $\nabla_\lambda L = 0$, we can re-arrange and solve for both Δx and λ_{new}. The following matrix formulation is referred to as a KKT system (Karush-Kuhn-Tucker), the generalized form for writing constrained optimization problems.

$$\begin{bmatrix} \boldsymbol{H}_L & -\boldsymbol{J_g}^T \\ \boldsymbol{J_g} & \boldsymbol{0} \end{bmatrix} \begin{bmatrix} \Delta\boldsymbol{x} \\ \boldsymbol{\lambda}_{new} \end{bmatrix} = \begin{bmatrix} -\nabla\boldsymbol{f} \\ -\boldsymbol{g} \end{bmatrix} \tag{36}$$

We now consider this example in matlab:

```
%% minimize f(x1,x2)=x1^2 + x2^2

%% given g(x1,x2) = x1*x2-3=0

%% first some plotting of the problem

clear all

figure(1)

xv=-4:.4:4;

yv=-4:.4:4;

for i=1:length(xv)

   f(:,i)= xv(i)^2 + yv.^2;

end

S=surf(xv,yv,f);,hold on

set(S,'facealpha',.5);

plot([-2 2],[0 0],'k')

plot([0 0],[-2 2],'k')

contour(xv,yv,f,[0.5 2 3 5 7 9 11 13])

xlabel('x_1','fontsize',20,'fontname','times')

ylabel('x_2','fontsize',20,'fontname','times')

zlabel('f(\bfx\rm)','fontsize',20,'fontname','times')


x=[-4:0.01:3/-4 0 3/4:.01:4];

y=3./x;

plot(x,y,'k','linewidth',2,'linesmoothing','on')


figure(2)

contour(xv,yv,f,[0.5 1.5 2.5 4 6 9 11 13]), hold on

plot(x,y,'k','linewidth',2,'linesmoothing','on')

plot([-5 5],[0 0],'k')

plot([0 0],[-5 5],'k')

axis equal


% get rid of all plotting variables...
```

```
clear

%% newton's method for constrained optimization: KKT system
maxiter=100;
x=[-4 -3]';
lambda=0;
for i=1:maxiter
%    mapping:
% betts me
% g    nablaf
% c    g [GG temporarily]
% b    -c and -g ; Jac is -Jac in the first equation.
% Hc   Hg
%
    i
    figure(2)
    p1=plot(x(1),x(2),'ro','markersize',6,'markerfacecolor','r');
    f= x(1)^2+x(2)^2; %cost function
    g=x(1)*x(2)-3;    %constraint function
    L=f-lambda*g;     %Lagrangian
    nablaf=[2*x(1);2*x(2)];          %gradient of f
    Jg=[x(2) x(1)];    %Jacobian of g
    Hf=[2 0;0 2];          %Hessian of f
    Hg=[0 1; 1 0];          %Hessian of g
    Hl=Hf-lambda*Hg;  %The Hessian of the Lagrangian
    Null=zeros(1,1);        % a null matrix
    %%%Now form KKT system:
    A=[Hl -Jg';Jg Null];
    b=[-nablaf;-g];           %these are the knowns
    dX=A\b;           %these are the unknown variables and Lagrange multipliers
    dx=dX(1:length(x)); %change in x
    lambda=dX(length(x)+1:end)%new estimated lambda
```

```
   x=x+dx              %new estimated value for x
%     pause
   set(p1,'markerfacecolor','k')
   axis equal
   if norm(dx)<.01 %this means we stop if dx is smaller than a predefined value
      break
   end
end
%until here
```

6 Direct collocation

We have thus far considered objective and constraint functions that were abstract and visualizable. We will now consider applying this task to a dynamical system for which the variables are not abstract, and the number of them makes the solution no longer immediately visualizable in the same way. We will consider a toy system, a driven pendulum, as the system that we would like to control. We will try to find a driving torque $u(t)$ to the pendulum to bring it from a starting position x_0 to x_{end}. We will try to find the minimum control signal, $\sum_{i=0}^{t_{end}} u^2(t)$.

The first thing we need to understand is: how do we we deal with time? The second question we might have is: what is x, in this case? In other words, what are the variables for which we are solving?

It turns out that the answer to these two questions is related. In direct collocation we divide up time into discrete units, Δt. We choose the size of these discrete units so that we can make good-enough approximations of the differential terms in our equation. We might use .001 s as a reasonable unit in time for human movements. Thus if we simulate a movement lasting 2 seconds, we will have 2000 slices of time.

The variables x we might guess to be only our optimization parameters, namely our input torques u. Instead, x is a composite vector composed of:

- the positions of our system at every moment in discrete time (as it turns out, we do not need a minimal set to use many implementations of collocation)
- the inputs of our system

While it might seem strange that we are finding both the generalized coordinates and input torques that solve our optimization problem, there is a good reason for it: it allows us to take advantage of our equations of motion, and use them as constraint equations. Here is the equation of motion for this single degree-of-freedom system using Lagrangian dynamics (a different idea than Lagrange multipliers which we do not discuss here) in an inertial reference frame:

$$j\ddot{x} = mg\sin(x) + u(t) \tag{37}$$

Note that j, m and g are just constants. As noted above, the cost/objective function that we want to minimize is the total input torque u:

$$\min_{t=0\ldots end} u(t)^2 \tag{38}$$

$$\tag{39}$$

For inertia j, mass m, grativational constant g, and external torque u. We also assume some constraints for the start and end of the pendulum's position: i.e. $x0$ and x_{end} are some constant values, and the velocity of the pendulum is zero at beginning and end. These will thus add 4 additional constraint equations.

We make use of our discretized-time design to replace our differential term, angular acceleration, with an approximation by finite differences. The time derivative of x is approximated as

$$\dot{x} = \frac{x_{i+1} - x_i}{\Delta t} \tag{40}$$

and the second time derivative is

$$\ddot{x} = \frac{x_{i+1} - 2x + x_{i-1}}{\Delta t^2} \tag{41}$$

We then re-write our equation of motion

$$g(x) = x_{i+1} - 2x_i + x_{i-1} - \frac{\Delta t^2 mg\sin(x_i) + u_i}{j} = 0 \tag{42}$$

for which we can simply compute the partial derivatives (with respect to our variables x_{i-1}, x_i, x_{i+1} and u_i) analytically:

$$\nabla g = \begin{bmatrix} \frac{\partial g}{\partial x_{i-1}} \\ \frac{\partial g}{\partial x_i} \\ \frac{\partial g}{\partial x_{i+1}} \\ \frac{\partial g}{\partial u_i} \end{bmatrix} = \begin{bmatrix} 1 \\ -2 - \frac{\Delta t^2 mg\cos(x_i)}{j} \\ 1 \\ -\frac{1}{j} \end{bmatrix} \tag{43}$$

Thus we have written the jacobian of the constraint equations with respect to the states and input torques. How does our resulting sub-matrix of constraint equations then look? Our constraint terms in our KKT system are combination of two sets. First, our equations-of-motion constraints. Since the above equation is a constraint at every time point, i.e. for all i of our n discretized timepoints, our constraint equations have a stack of n equations where this must be true at each time point. The second portion of our constraint equations is our constraints on the movement itself, namely the constraints on the states of the system (position and velocity) at beginning and end. In general this second set will have far fewer equations (one or two for the beginning and end each).

In the following exercise the organization of direct collocation is done for a simple pendulum example. Pay particular attention to how the constraint equations are organized. They will follow the layout of the KKT system 36. The exercise is organized in two matlab files, a function file and a short calling script. If you want to test yourself, delete the functions that compute the constraint jacobian, and write them!

```
function out = DC_first_3(t,x_1,x_end,param)
% function out = DC_first_3(t,x_1,x_end)
% performs direct collocation on a single pendulum.
% t are the timesteps of the system (should be x X 1 in size)
% x is the only state of the system, the angular position of the bob.
% the additional constraints are that
% x((t(1))=x_1 and x(t(end)) = x_end;
% dx(end) = dx(1) = 0
% NOTES: 2014/10/22:
% I use the notation 'C' to denote the constraints in this function.
%

%physical system structure
% param=struct;
% param.g=9.81;
% param.r=1;
% param.m=1;

%\Delta t
param.dt=t(2)-t(1); %well, not so 'physical' maybe...

%matrix structure keeping track of the matrix dimensions.
ma=struct;
ma.nStates=size(t,1)*2;%since we need position and torque states.
ma.nX=size(t,1);
ma.nEConstraints=4; %4 extra constraints for position and velocity at end.
ma.nCon=ma.nX + ma.nEConstraints; % total number of constraint equations.
ma.iX=1:size(t,1); %indices of position vector (# of time slices)
ma.iU=size(t,1)+1:ma.nStates; %indices of driving torque.


x=zeros(size(t,1),1);% initial guesses of the position.
%the torques on the bob, u;
u=zeros(size(t,1),1);% initial guesses for the torque.
```

```
% tolerances for the KKT iteration
ctol = 1e-4;          % constraint tolerance
ftol = 1e-4;          % cost function tolerance
xtol = 1e-4;          % solution tolerance
maxiter = 100;    % maximum number of iterations
F = 1e10;
F_prev = 1e8;
dX=1;
C=1;
% /tolerances

% begin KKT loop
iter=1;
while (TOL_MET(max(abs(C)),ctol,abs(F-F_prev),ftol,mean(abs(dX)),xtol,iter,maxiter)==0)
    clear dX;clear C;
    F_prev=F;
    [F,grad_F,H_F]=Lagrangian_Objective_Pendulum(x,u,param,ma);
    [C,grad_C,H_C]=Lagrangian_Constraint_Pendulum(x,u,x_1,x_end,param,ma);
    b=[-grad_F;-C];
    K=[H_F+H_C grad_C'; grad_C zeros(ma.nCon,ma.nCon)];
    dZ=K\b;
    dX=dZ([ma.iX,ma.iU]);
    X=[x;u];
    Xbar=X+dX;
    x=Xbar(ma.iX);
    u=Xbar(ma.iU);
    iter = iter +1;
    disp(['C=',num2str(max(abs(C))),' F-F_prev=',num2str(abs(F-F_prev)),'
        meanabsdX=',num2str(mean(abs(dX)))]);
end;
% /end KKT loop
out.x=x;
out.u=u;
out.t=t;
out.iter=iter;
end

%%%%CONSTRAINT FUNCTION
function [C,grad_C,H_C]=Lagrangian_Constraint_Pendulum(x,u,x_1,x_end,param,ma)
g=param.g;
r=param.r;
dt=param.dt;
m=param.m;
C=zeros(size(x,1)+2,1);
grad_C=spalloc(ma.nX+ma.nEConstraints,ma.nStates,4*ma.nStates); % sparce matrices! just for
    speed, not conceptually different.
xdd=getdt2(x,param);
H_C=spalloc(ma.nStates,ma.nStates,4*ma.nStates);

%iterate through timesteps and calculate C,grad_C, and H_C for the
%columns of x - namely the position states.
for i=1:size(x,1)
    C(i,1)=xdd(i)+(g*m*r*sin(x(i)) - u(i))*dt^2/(m*r^2); %Constraint along main diagonal
    H_C(i,i)= -(g*m*r/m/r^2)*sin(x(i))*dt^2; %Hessian along main diagonal

    %for calculation of the jacobian, we have to know what kind of
    %derivative approximations are used. in this case we are using the
    %3-point approximations
    k=g*m*r/(m*r^2);
    if i==1 %if we are on t(1)
        grad_C(i,i)=1 + k*cos(x(i))*dt^2;
        grad_C(i,i+1)=-2;
        grad_C(i,i+2)=1;
    elseif i==size(x,1) %if t(end)
```

```
        grad_C(i,i)=1 + k*cos(x(i))*dt^2;
        grad_C(i,i-1)=-2;
        grad_C(i,i-2)=1;
    else %otherwise
        grad_C(i,i)=-2 + k*cos(x(i))*dt^2;
        grad_C(i,i+1)=1;
        grad_C(i,i-1)=1;
    end;
end;
%iterate through timesteps to calculate C, grad_C and H_C for the
%columns of u - the torque states.
for i = 1:size(ma.iU,2)
    grad_C(i,ma.iU(i))=-1/(m*r^2)*dt^2;
end;
%four (!) extra constraint functions
s_X=ma.nX;
C(s_X+1,1)=x(1)-x_1;    %static start and
C(s_X+2,1)=x(end)-x_end; %end constraints
grad_C(s_X+1,ma.iX(1))=1;
grad_C(s_X+2,ma.iX(end))=1;
C(s_X+3,1)=(-3*x(1)+4*x(2)-x(3));
C(s_X+4,1)=(x(end-2)-4*x(end-1)+3*x(end));
%as a result of the three point first derivative, there are three
%partial for each additional velocity as well.
grad_C(s_X+3,ma.iX(1))=-3;
grad_C(s_X+3,ma.iX(2))=4;
grad_C(s_X+3,ma.iX(3))=-1;
grad_C(s_X+4,ma.iX(end-2))=1;
grad_C(s_X+4,ma.iX(end-1))=-4;
grad_C(s_X+4,ma.iX(end))=3;
end
%%%%//END Constraint function
```

```
%%%% Objective Function
function [F,grad_F,H_F]=Lagrangian_Objective_Pendulum(x,u,param,ma)
%Here dinant scales his torques by the timeslice. i guess here he's
%taking the integral so it makes sense to do so...

grad_F=zeros(ma.nStates,1);
H_F=spalloc(ma.nStates,ma.nStates,ma.nStates);
F=param.dt*sum(u.^2);
grad_F(ma.iU)=2*param.dt*u;
for i=1:size(ma.iU,2)
    H_F(ma.iU(i),ma.iU(i))=2*param.dt*1;
end;
end
%%%%/ END Objective Function

% compute acceleration approx using 3-point finite difference.
function out = getdt2(x,param)
out=zeros(size(x,1),1);
dt=param.dt;
for i=1:size(x,1)
    if i==1
        out(i)=(x(1)-2*(x(2))+x(3));
    elseif i==size(x,1)
        out(i)=(x(i-2)-2*(x(i-1))+x(i));
    else
        out(i)=(x(i-1)-2*(x(i))+x(i+1));
    end;
end;
end

%end conditions for iteration terr < tcval, titer < tmaxiter
```

```
%returns 0 if end conditions reached (time to stop!).
function out = TOL_MET(c,cthresh,f,fthresh,x,xthresh,iter,maxiter)
out = 0;

if iter>(maxiter-1)
    out=1;
end;
if (c<cthresh && f<fthresh && x<xthresh)
    out=1;
end;
end
```

```
t=(0:0.001:1)';
param=struct;
param.g=9.81;
param.dt=t(2)-t(1);
param.r=1;
param.m=1;
param.x0=0;
xend=pi/2;
solution=DC_first_3(t,param.x0(1),xend,param);
```

7 Further reading

There are several topics for further reading that might be of interest. These include

- what can go wrong in general

  determining that each Newton iteration is actually improving

  the importance of initial good guesses of the solution ensuring that your equations are continuous in first and second derivatives

implicit methods for numerical integration, which appear to be crucial for solving problems in biomechanics

- speeding things up with approximations to derivatives and sparse matrices
- where collocation stands in regard to other non-linear optimization approaches

It is important to note that for people beginning to apply this tool to the biomechanics field, it is not as much a science as it is an art. Numerical problems can potentially paralyze a novice attempting to begin using this technique. For example, some of the people who first published this approach in biomechanics literally took months resolving the things that can go wrong, particularly with finding initial conditions that converged to a solution, and discovering the importance of implicit formulations of the equations of motion (listed above).

Direct collocation is a powerful technique. It has generated solutions to problems that we previously could not really solve. In this case, "solve" means "find kinematics, forces, and energetics of movements that are approximately human-like", and not "find the global minimum solution", since the only problems where we have provably found the global minimum are for convex problems, which biomechanics control problems are not. The technique is also useful because it produces these solutions quickly, relative to other approaches. It has broad use, and a handful of interesting uses include (in the motor control/biomechanics/robotics field) Emo Todorov [2], Manoj Srinivasan and Andy Ruina [3], Marko Ackermann and Antonie van den Bogert [4–6], David Remy [7], and Dinant Kistemaker [8].

For biomechanists, all studies including those mentioned above use direct collocation in one of two ways: either merely to just generate movements that meet some standard of optimality (so that we might compare human behaviour to it), or as a proposed computational method or algorithm that the brain implements to solve control problems of its own (Todorov is the only one in the above list to occasionally claim this). In both use cases, direct collocation is not the only game in town. Many computer scientists in the field of animation prefer to use covariance matrix adaptation [9]. Deep reinforcement learning (a combination of deep neural networks [10] and reinforcement learning [11] to allow learning through experience) makes no use of trajectory optimization either, and has achieved the most revolutionary results in control, at least in the domain of decision making but also achieving video game mastery for many ATARI games [12, 13].

In the broader field of optimization, direct collocation is often referred to as "spacetime methods" and "trajectory optimization." Technical references are Gill Murray and Saunders [14], Witkin and Kass [15], and notably Betts [16], the textbook on which this introduction is based. I would like to thank Dinant Kistemaker, from whom I have used significant portions of his introductory lecture on Direct Collocation at Vrije Universiteit.

References

1. Roach NT, Venkadesan M, Rainbow MJ, Lieberman DE (2013) Elastic energy storage in the shoulder and the evolution of high-speed throwing in Homo. Nature 498: 483–486.

2. Todorov E, Jordan MI (2002) Optimal feedback control as a theory of motor coordination. Nature Neuroscience 5: 1226–1235.

3. Srinivasan M, Ruina A (2006) Computer optimization of a minimal biped model discovers walking and running. Nature 439: 72–75.

4. Ackermann M, van den Bogert AJ (2010) Optimality principles for model-based prediction of human gait. Journal of biomechanics 43: 1055–1060.

5. van den Bogert AJ, Blana D, Heinrich D (2011) Implicit methods for efficient musculoskeletal simulation and optimal control. Procedia IUTAM 2: 297–316.

6. Ackermann M, van den Bogert AJ (2012) Predictive simulation of gait at low gravity reveals skipping as the preferred locomotion strategy. Journal of biomechanics 45: 1293–1298.

7. Remy C (2011) Optimal Exploitation of Natural Dynamics in Legged Locomotion. PhD Dissertation .

8. Kistemaker DA, Wong JD, Gribble PL (2014) The cost of moving optimally: kinematic path selection. Journal of Neurophysiology 112: 1815–1824.

9. Hansen N (2006) The CMA Evolution Strategy: A Comparing Review. In: Towards a New Evolutionary Computation, Berlin, Heidelberg: Springer Berlin Heidelberg. pp. 75–102.

10. Hinton GE, Salakhutdinov RR (2006) Reducing the Dimensionality of Data with Neural Networks. Science (New York, NY) 313: 504–507.

11. Barto AG, Sutton RS (1998) Reinforcement learning: an introduction. Cambridge: MIT Press.

12. Mnih V, Badia A, Mirza M, Graves A (2016) Asynchronous methods for deep reinforcement learning. arXiv preprint arXiv: … .

13. Gu S, Lillicrap T, Sutskever I, Levine S (2016) Continuous Deep Q-Learning with Model-based Acceleration. arXiv preprint arXiv:160300748 .

14. Gill P, Murray W, Saunders M (2002) Snopt - an SQP algorithm for nonlinear optimization. Siam Journal of Optimization 12: 979–1006.

15. Witkin A, Kass M (1988) Spacetime constraints. Computer Graphics 22: 159–168.

16. Betts J (2001) Practical Methods for Optimal Control Using Nonlinear Programming. SIAM .