# Algorithmic Control of Walking

A. James Stewart
James F. Cremer
Computer Science Department
Cornell University

## Abstract

We suggest that there are two components required in most control schemes: an algorithmic component which takes a high-level goal and generates joint trajectories, and a dynamics component which takes these joint trajectories and generates the required joint torques. The joint torques are then sent to the mechanism, which (hopefully) achieves the high-level goal.

We present an approach that simplifies the programming of the algorithmic component for high degree of freedom objects. Instead of supplying a complete set of joint trajectories as a function of time, the algorithm controls other, more intuitive, degrees of freedom. These degrees of freedom are automatically converted to the more conventional joint trajectories. The algorithm can underconstrain the object, in which case constrained optimization is used in converting to joint trajectories.

This approach is applied to generate joint trajectories for a walking biped. The walking algorithm is presented along with the results from testing with the *Newton* simulation system.

## 1 Introduction

There are two components required in most control schemes: an algorithmic component and a dynamics component. The algorithmic component generates joint trajectories from high-level goals, and the dynamics component generates actuator torques from the joint trajectories.

This paper describes a general approach that simplifies the algorithmic component of the control scheme. The approach advocates the selection of a small set of intuitive variables which are used by the algorithm in controlling the object. These degrees of freedom are automatically converted into the more conventional joint variables. In the event that the algorithm underconstrains the motion of the object, constrained optimization techniques are used to choose a motion that optimizes some criterion while satisfying the constraints imposed by the algorithm.

This paper does not consider the dynamics component of the control scheme. However, inverse dynamics is used in the algorithmic component, and joint torques are generated as a side effect; these are the torques required to control the ideal, simulated, object. We do not expect these torques to have much similarity with those required to control a real object, and hence treat the dynamics component as a separate, unsolved, problem.

We demonstrate the algorithmic approach by programming a sixteen degree of freedom anthropomorphic biped. The resulting algorithm generates realistic joint trajectories and has been tested on the *Newton* simulation system with favorable results.

## 2 Motivation

Various approaches have been taken to reduce the complexity of high-level control programs. The computed torque method for robot arms (see [1]) can, in our view, be thought of as simplifying control by splitting the problem into two components: an algorithmic component and a dynamics component. The algorithmic component can ignore the dynamics of the robot arm, only concerning itself with the position of the end effector as a function of time. The dynamics component can ignore the high-level goal, only concerning itself with generating torques to achieve the specified end effector trajectory.

In building his one-legged hopping machine, Raibert [8] partitioned control along three intuitive degrees of freedom: hopping, forward speed and body posture. This resulted in surprisingly simple control programs for the hopping robot. For multi-legged machines, Raibert introduced the idea of a "virtual leg" which was defined in terms of the robot's physical legs. This again led to simplified control programs.

Both the computed torque method and Raibert's virtual leg demonstrate that a proper choice of control variables can lead to simplified control. We believe that by partitioning the control problem into *algorithm* and *dynamics*, and by choosing an intuitive set of control variables, relatively complex objects can be controlled with greater ease.

Different methods have been used to deal with high degree of freedom objects. Some control programs for redundant manipulators use the extra degrees of freedom to

```
procedure initialize

  begin
  add-equation( a_cm = (1/M) Σ m_i a_i )
  end

procedure controller( time )

  begin
  a_cm := f( time )
  end
```

Figure 1: The Format of an Algorithm

giving the algorithm access to these force and torque quantities[1]. The algorithm could, for example, specify the amount of force required between two objects while at the same time specifying the acceleration of some other object. A concrete example is a humanoid robot hurling a ball. The algorithm could specify fifty pounds of force between the ball and the hand (in some direction), and could simulataneously require that the center of mass of the robot remain at rest.

Figure 1 shows the format of a control algorithm. For the sake of clarity the algorithms will be described in a Pascal-like notation. Two procedures are always present: one to initialize the algorithm (called `initialize`) and one to be executed repeatedly over the course of the task (called `controller`). The `controller` procedure has access to the complete state of the system. The algorithm of Figure 1 trivially defines and controls $a_{cm}$, the acceleration of the center of mass of an object (the function $f$ must be defined elsewhere).

Defining and controlling a vectorial quantity like the acceleration of the center of mass has the effect of adding three constraint equations to *Newton's* system of simultaneous linear equations that describe the instantaneous motion of the object. By considering joint torques as *unknowns* in this augmented system of equations, the system can be solved to produce values for these torques – and for the unknown forces and object accelerations – that satisfy the additional constraint equations. This is a simple application of inverse dynamics.

In controlling a real object, the object accelerations would be used to directly compute the joint accelerations. These joint accelerations would be given to the *dynamics* component, which would calculate the torques required to achieve these accelerations. In our *Newton* simulation, however, the object accelerations are simply integrated over time to produce the simulated motion.

For an object with $n$ degrees of freedom the control algorithm can define and control up to $n$ independent scalar quantities[2]. If fewer than $n$ equations are added, the system of motion equations is underdetermined and many different solutions could satisfy the constraints of the control algorithm. In this case the algorithm must guide the selection of a solution by providing a cost function which is quadratic in the unknowns. A standard numerical optimization technique is used to compute a solution that minimizes the cost function while obeying the algorithm's constraints.

In summary, the programmer designs an algorithm in a high-level computer language to control intuitive degrees of freedom of the object. These degrees of freedom are defined as linear combinations of the unknowns in the object's equations of motion. An augmented linear system of equations describes the instantaneous behavior of the object; this system can be solved to produce the joint accelerations required to achieve the desired motion. If the system is underdetermined, the algorithm can provide a cost function to guide the choice of a solution.

In the remaining sections we describe the application of this approach to the design of a simple walking algorithm.

# 5 Trajectory Generation

In designing algorithms with *Newton*, we found ourselves frequently using PD controllers and curve-fitting controllers to generate the trajectory of many of the defined quantities. Unlike PD controllers used with real objects, our "controllers" simply generated the second derivative of a quantity as a function of time. To avoid confusion we will call them "trajectory generators". The programmer can make use of trajectory generators to specify the trajectories of the intuitive control variables.

In the biped algorithm, for example, a quintic curve generator is used for the trajectory of the heel, and a PD generator is used to orient the foot before it strikes the ground. The PD generator defines the foot's angular acceleration as a function of time until the foot strikes the ground. A small library of these trajectory generators is used in the biped algorithm, and will be described here.

PD generators are used in the biped algorithm to control orientation, position and joint angle. Each generator adds an equation to the system of motion equations that defines the second derivative of the quantity in terms of the first derivative and the quantity itself. The procedure in Figure 2 produces accelerations to move an object to within 1% of a position `x-desired` within a given time `delta-time`. The quantities $x$, $v$ and $a$ are data structures representing state variables of the controlled object. These data structures are used by the `add-named-equation` function to create the appropriate equation.

Execution of the procedure in Figure 2 causes a named equation to be added to the system of motion equations. This equation will continue to affect the motion of the object until it is explicitly removed by the control algorithm.

The biped algorithm uses three similar trajectory generators: `orient-with-PD`, `set-angle-with-PD` and

[1] A simpler approach might be to let the algorithm specify *positional* constraints which would be automatically differentiated by *Newton* to obtain the second derivatives.

[2] The additional definitional equations could make the system of motion equations inconsistent. This would be an error on the part of the control algorithm.

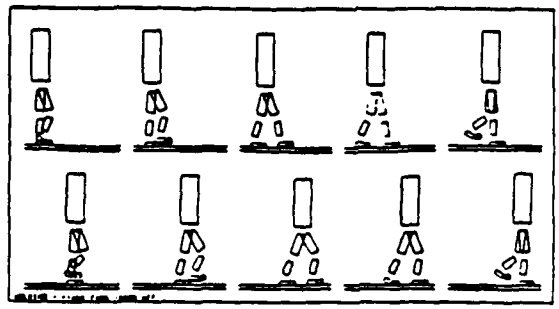

Figure 4: Walking Cycle

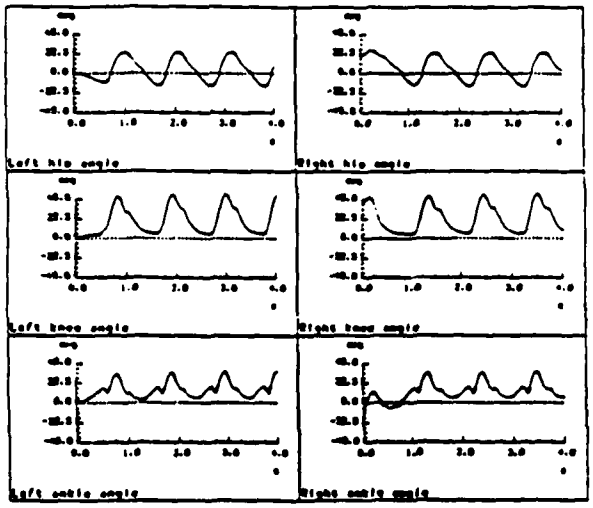

Figure 5: *Newton* Statistical Output

biped completes a full cycle of the six phases described above. The full simulation consisted of twenty seconds of straight-line walking at varying speeds on a flat surface and generated the statistics shown in Figure 5 (only a portion of the simulation is shown).

Due to the simplicity of our current biped model, this algorithm is forced to use many constraints to achieve the desired motion. In particular, the trajectory of the heel must be specified. It almost seems that the algorithmic approach to walking is at least as complicated as any other approach. However, if the biped model were extended to include elastic tendons the number of constraints might be reduced. In this case, the swing phase would not have to specify a trajectory for the heel. Instead, no torque would be applied in the swing leg; it would be pulled forward by the stored energy of the stretched tendons. This might approximate the "ballistic walking" described by McMahon[7].

We feel that a high-level algorithm should *greatly underdetermine* the motion of the controlled object. Our philosophy is to incorporate in the model many "passive elements" – such as springs, dampers and joint limits – which reduce the number of constraints needed by the control algorithm. The algorithm then has the job of guiding, rather than forcing, the motion of the object.

# 8 Summary

There are two components required in most control schemes: an algorithmic component and a dynamics component. We have presented a general approach that simplifies the programming of the algorithmic component. This approach allows the programmer to choose intuitive degrees of freedom by which to control an object; if the motion of the object is not fully determined the programmer can define a quadratic cost function which is minimized subject to the constraints of the control algorithm. We believe that the algorithm should greatly underconstrain the motion by relying on "passive elements" of the model.

We have presented an algorithm to generate joint trajectories for a biped, along with results from the algorithm's execution on the *Newton* simulation system. We do not expect our algorithm to work on a physical biped. Rather, we are using this example to demonstrate the usefulness of a high-level algorithmic approach in controlling an object. An algorithm embodies the *ideas* and the *structure* of a solution to a given problem. We believe that controlling complex physical objects requires a structured, algorithmic approach similar to that presented in this paper.

# Acknowledgements

We would like to thank the reviewers for their helpful comments. This work was supported in part by NSF grant DMC 86-17355, ONR grant N0014-86K-0281 and DARPA grant N0014-88K-0591. Support for James Stewart is provided in part by U.S. Army Mathematical Sciences Institute grant U03-8300 and NASA training grant NGT-50327. The *Newton* system is being developed in Common Lisp on Symbolics Lisp Machines and can be used on other machines supporting Common Lisp.

# References

[1] J. J. Craig. *Introduction to Robotics: Mechanics and Control.* Addison Wesley, 1986.

[2] J. F. Cremer. PhD thesis, Cornell University, May 1989.

[3] J. F. Cremer and A. J. Stewart. The architecture of *Newton*, a general purpose dynamics simulator. In *IEEE ICRA '89*.

[4] R. Featherstone. The dynamics of rigid body systems with multiple concurrent contacts. In *Robotics Research: The Third International Symposium*, O. D. Faugeras and G. Giralt, editors, pages 191–198. The MIT Press, 1985.

[5] C. M. Hoffmann and J. E. Hopcroft. Simulation of physical systems from geometric models. *IEEE Journal of Robotics and Automation*, RA-3(3):194–206, June 1987.

[6] P. Hsu, J. Hauser, and S. Sastry. Dynamic control of redundant manipulators. In *IEEE ICRA '88*, pages 183–187.

[7] T. A. McMahon. Mechanics of locomotion. *Intl. J. Robotics Research*, 3(2):4–28, 1984.

[8] M. H. Raibert. *Legged Robots That Balance.* The MIT Press, 1986.

[9] D. Zeltzer. Motion control techniques for figure animation. *IEEE Comp. Graphics and Applic.*, 2(9):53–59, 1982.

avoid obstacles or to stay away from singularities. Work presented in [6] uses the one-dimensional null space of the Jacobian of a three link planar arm to move the arm into a configuration of equal joint angles, thereby avoiding singularities where possible.

Our decision to allow control programs to underconstrain the controlled object – requiring the use of constrained optimization techniques – is based on the belief that researchers will soon be attempting to control objects much more complex than robot arms. These objects will have many more degrees of freedom than are necessary to their control programs. A robot modeled after the human figure may have more than fifty degrees of freedom [9], while the control program for such a robot would only require twenty or thirty degrees of freedom to accomplish its task. In programming our anthropomorphic robot we needed at most eleven of its sixteen degrees of freedom at any given instant.

In summary, with the algorithmic approach the algorithm constrains a set of intuitive degrees of freedom that are defined in terms of the conventional degrees of freedom of the controlled object. The algorithm is allowed to underconstrain the motion of the object, in which case a motion is chosen which minimizes a cost function while obeying the constraints.

# 3 Overview of Newton

The algorithms described in this paper have been designed and tested using the *Newton* simulation system, part of a large research effort in modeling and simulation at Cornell University[5,2]. Using *Newton*, a designer can define complex physical objects and can represent object characteristics from a wide range of domains. In particular, a dynamic simulation of the object can be performed. A more detailed description of Newton can be found in [3], in these proceedings.

An object is modeled as a collection of rigid bodies related by constraints. Newton-Euler equations of motion are associated with each rigid body. At the time an object is created the equations are of the form

$$m\dot{v} = mg$$
$$J\dot{\omega} + \omega \times J\omega = 0.$$

Constrained relationships between objects are normally represented by data structures called "hinges." A specification that two objects are to be connected with a spherical joint is met by the addition of one vectorial constraint equation and the addition of some terms to the motion equations of the constrained objects. Thus, the equations above become

$$\begin{aligned} m_1\dot{v}_1 &= m_1 g + F_{hinge} \\ J_1\dot{\omega}_1 + \omega_1 \times J_1\omega_1 &= c_1 \times F_{hinge} \\ m_2\dot{v}_2 &= m_2 g - F_{hinge} \\ J_2\dot{\omega}_2 + \omega_2 \times J_2\omega_2 &= c_2 \times -F_{hinge} \\ \dot{v}_1 + \dot{\omega}_1 \times c_1 + \omega_1 \times (\omega_1 \times c_1) &= \dot{v}_2 + \dot{\omega}_2 \times c_2 + \omega_2 \times (\omega_2 \times c_2) \end{aligned}$$

where $c_i$ is the vector from object $i$'s center of mass to the location of the hinge and $F_{hinge}$ is the constraint force that keeps the objects together. Other kinds of hinges commonly used in *Newton* include revolute or pin joints, prismatic joints, springs and dampers, and rolling contacts.

The motion equations are collected from each rigid body and assembled into a system of equations which can be solved to determine instantaneous accelerations and hinge constraint forces. *Newton* integrates the derived accelerations to produce the motion of the simulated object.

*Newton*, unlike many other simulation systems (though see [4]), can automatically and incrementally reformulate the motion equations as exceptional events occur during simulations. Two examples of exceptional events are impact and changing kinematic relationships between objects, such as a block sliding off a table. These changing contact relationships are detected automatically by *Newton* and the system of motion equations and the related constraint equations are automatically maintained to reflect these changing relationships. This is explained in more detail in [3].

User-directed influence of object motion is supported in part through control forces and control torques. If a programmer wants to associate an actuator with a hinge between two objects, the system associates a control torque quantity with the hinge. As part of the creation of this quantity, the system modifies the motion equations of the two constrained objects, adding torque terms to the relevant motion equations.

# 4 The Algorithmic Approach

In *Newton's* automatically-generated equations of motion certain quantities are considered to be *unknowns*. A system of simultaneous linear equations is solved at each time step to produce values for the unknowns. These values are integrated over time to produce the simulated motion. Typically, the unknowns consist of accelerations and joint constraint forces, while positions, velocities and joint control torques are *knowns*.

In the algorithmic approach, the programmer controls "intuitive" quantities defined as linear combinations of the unknowns. The programmer might, for example, want to control the acceleration of the center of mass of a biped without explicitly controlling each component of the biped. To do this, the algorithm must define the acceleration of the center of mass in terms of the accelerations of the centers of mass of the primitive components of the object. Over the course of execution, the algorithm must supply the desired acceleration of the center of mass.

It might appear needlessly complicated to require the algorithm to control the second derivative of a variable (e.g. acceleration) rather than the variable itself (e.g. position). Doing it in this fashion, however, allows the second derivative quantities to be related to the force and torque quantities in *Newton's* motion equations, thus

```
procedure position-with-PD( constraint-name, object,
                            x-desired, delta-time )

  var x, v, a:  quantity
      τ:  real

  begin
  x := get-position-quantity( object )
  v := get-velocity-quantity( object )
  a := get-acceleration-quantity( object )

  τ := - delta-time / log( .01 )

  add-named-equation( constraint-name,
                      a + 2/τ v + 1/τ²(x - x-desired) = 0 )
  end
```

Figure 2: PD Trajectory Generator

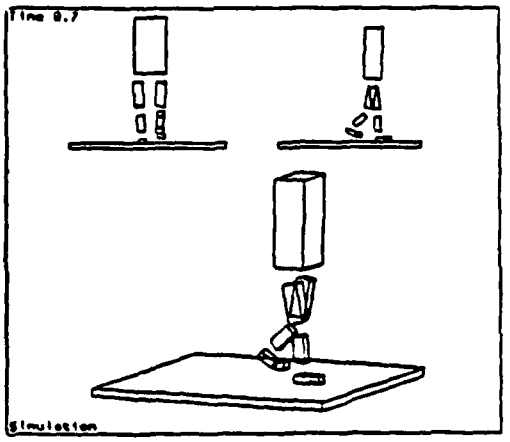


Figure 3: Simulated Biped Model

`position-point-with-quintic`. This last generator plots a quintic trajectory for a particular point on an object (in our case, the heel on the foot).

# 6 The Biped Model

The simulated biped is composed of a torso, two legs with knee joints and two feet with toe joints. This model was adapted from a description in [7] and is shown in Figure 3. The hips and ankles are three degree of freedom spherical joints, while the knees and toes are one degree of freedom revolute joints, making a total of sixteen degrees of freedom. The biped is about six feet tall with moments approximating those of a human being.

We hope to improve this model by incorporating joint limits and elastic tendons. McMahon suggests that, during walking, energy is stored in stretched tendons and is released when the stretched leg swings forward [7]. This idea might be used to simplify the walking algorithm described in the next section.

*Newton's* impact handling capabilities have not yet been extended to accurately model the impact of the feet upon the ground. Instead, impact is simulated by adding an external force and torque to the feet that holds them level with the ground until they are released with an explicit command from the control algorithm. This is as though the biped was walking with magnetic shoes on a steel plate. Very shortly we expect to adapt the algorithm to incorporate realistic impact.

| Constraint Name | dof | Constrained Item |
|---|---|---|
| TORSO-CONSTRAINT | 3 | torso orientation |
| L-KNEE-ANGLE | 1 | knee joint angle |
| R-HEEL-TRAJ | 3 | heel acceleration |
| R-FOOT-ORIENTATION | 3 | foot orientation |
| R-TOE-ANGLE | 1 | toe joint angle |

Table 1: Swing Phase Constraints

# 7 The Walking Algorithm

An abbreviated version of the walking algorithm is shown in Figure 6 at the end of this paper. The algorithm cycles through a set of six states: swing the right leg, land the right foot, lift the left foot, swing the left leg, land the left foot, lift the right foot and then repeat the cycle. In the *swing* phase, a quintic trajectory is plotted for the swing foot with `move-heel-to-target`, while the stance leg is stiffened with `set-angle-with-PD` and the foot is oriented for landing with `orient-with-PD` (shown under `START` in Figure 6). In the *landing* phase, the leading leg is stiffened as the foot nears the ground. Following this, the *takeoff* phase flexes the trailing leg, causing the trailing foot to lift from the ground. Once the trailing toe is bent to 10 degrees the flexing constraint is removed and the *swing* phase begins for the trailing leg.

The largest number of constraints is applied during the *swing* phase, as shown in Table 1. Since the biped has sixteen degrees of freedom (dof) it remains underconstrained at all times. A quadratic cost function is therefore defined (in `initialize` of Figure 6) in order to fully determine the motion of the biped. The cost function is a weighted sum of the translational and angular accelerations, and of the difference between the torso translational acceleration and an acceleration defined by a function F that tries to keep the torso mid-way between the two feet.

We found that a cost function that minimizes translational and rotational acceleration usually produces smooth motion. In the case of the simulated biped, the cost function causes the constrained heel acceleration to be achieved by a linear combination of small accelerations of many components of the body, rather than a few large accelerations of those components that are near the heel. We have observed that the combination of many small accelerations yields more stable motion than large, local accelerations.

The walking algorithm was tested with the *Newton* simulation system. Figure 4 shows ten frames in which the

```
const   time-in-air              = 0.5 s
        stride                   = 0.5 m
        direction                = (1 0 0)
        inside-step-fraction     = 20 %
        heel-Y-strike-speed      = -0.05 m/s
        heel-X-strike-speed      = 0.02 m/s
        foot-strike-orientation  = 10° about (0 0 1)
        torso-orientation        = -10° about (0 0 1)


var     phase:  ( start r-swing r-land l-lift l-takeoff l-swing l-land r-lift r-takeoff )


procedure move-heel-to-target( constraint-name, foot, other-foot, hip, other-hip )

  var target-r, target-v, hip-to-hip:  vector

  begin
  hip-to-hip := get-position( TORSO, hip ) - get-position( TORSO, other-hip )

  target-r := get-position( other-foot, HEEL ) + stride x direction
              + inside-step-fraction x hip-to-hip

  target-v := heel-Y-strike-speed x (0 1 0) + heel-X-strike-speed x direction

  position-point-with-quintic( constraint-name, foot, HEEL, target-r, target-v, time-in-air )
  end


procedure initialize

  let F = Kp((1/2)(r_l-foot + r_r-foot) - r_torso) + Kv((1/2)(ṙ_l-foot + ṙ_r-foot) - ṙ_torso)

  begin
  quadratic-cost := Σ ω² + Σ r̈² + 20 (r̈_torso - F)²
  phase := START
  end


procedure controller( time )

  begin
  case phase of

    START:
      phase := R-SWING
      orient-with-PD( TORSO-CONSTRAINT, TORSO, torso-orientation, 2.0 s )
      move-heel-to-target( R-HEEL-TRAJ, R-HEEL, L-HEEL, R-HIP, L-HIP )
      set-angle-with-PD( L-KNEE-ANGLE, L-KNEE, 175°, 0.1 s )
      orient-with-PD( R-FOOT-ORIENTATION, R-FOOT, foot-strike-orientation, time-in-air )
      set-angle-with-PD( R-TOE-ANGLE, R-TOE-JOINT, 0°, time-in-air )

    R-SWING:
      if distance-to-target( R-FOOT ) < 0.01 m then

        phase := R-LANDING
        remove-constraint( R-HEEL-TRAJ )
        set-angle-with-PD( R-KNEE-ANGLE, R-KNEE, 175°, 0.05 s )

    R-LANDING:
      if heel-has-touched( R-FOOT ) then

        phase := L-TAKEOFF
        remove-constraints( R-FOOT-ORIENTATION, R-TOE-ANGLE, L-KNEE-ANGLE )
        set-angle-with-PD( L-KNEE-ANGLE, L-KNEE, 160°, 0.1 s )

    L-TAKEOFF:
      if joint-angle( L-TOE-JOINT ) > 10° then

        phase := L-SWING
        remove-constraint( L-KNEE-ANGLE )
        move-heel-to-target( L-HEEL-TRAJ, L-HEEL, R-HEEL, L-HIP, R-HIP )
        orient-with-PD( L-FOOT-ORIENTATION, L-FOOT, foot-strike-orientation, time-in-air )
        set-angle-with-PD( L-TOE-ANGLE, bL-TOE-JOINT, 180°, time-in-air )

    Cases L-SWING, L-LANDING, and R-TAKEOFF
    are analogous to the preceding three cases.

    end
  end
```

Figure 6: Abbreviated Walking Algorithm