Tools for Integrated Modeling, Design, and Nonlinear Control

G.L. Blankenship, R. Ghanadan, H.G. Kwatny, C. LaVigna, and V. Polyakov

Increasingly, design engineers are identifying new opportunities for innovation by incorporating active microprocessor control into mechanical systems of all types; e.g., spacecraft and aircraft, ground vehicles, robots, and machine tools. These mechanical systems are often complex, multibody dynamical systems with rigid and elastic substructures. Their behavior is often inherently nonlinear over their operational range. Effective design of such systems and their controls relies on computer analysis for composing and screening alternative design concepts before constructing expensive prototypes. As a consequence, there has been a considerable amount of work on computational tools to support the development of models for systems with embedded control elements (see, e.g., the examples and references in [12] and [15]). To achieve optimal performance, it is critical to *integrate* the design of the system structures and the embedded control architecture and laws.

Our research is intended to contribute to the development of tools to support integrated design. The goal is a system as suggested in Fig. 1. We have not yet achieved all the elements suggested in the figure. Two key missing elements are the graphical definition of systems (or a definition from a requirements document) and the integrated optimization system. This article is a "progress report" on the status of our efforts, focusing on two components: (i) *Dynamics*, a package for generating models of multibody dynamics; and (ii) *Controls*, a package for design of nonlinear control systems. In effect, we have thus far achieved the system shown in Fig. 2. This article describes that system and some of its applications.

Our technical approach combines symbolic and numerical computing with graphics pre- and post-processing. Computer algebra and mathematical symbolic manipulation systems have matured substantially in recent years. Advances in this field provide an opportunity for a new approach to the assembly of models for integrated design. Model-building software based on computer algebra need not constrain systems to be composed of rigidly defined sets of components. Such an approach can greatly expand the design engineer's ability to devise and experiment with new types of elements and configurations. Equally important is that access to analytical tools for nonlinear (control) design and (bifurcation) analysis is natural in this setting. Some early work on this idea is reported in [7]. Our work can be compared to efforts such as the CAMeL system described in the paper [29], which is an open environment for CACSD of mechatronic systems. CAMeL includes facilities for use of parallel computing. We have not addressed this issue in our work.

In the sections that follow we describe an integrated set of tools developed using a symbolic computer algebra system (*Mathematica*) for the generation of models and the design of control laws for certain classes of nonlinear systems. The tools include: (i) *Dynamics*: a toolbox for automatic generation of explicit models for multibody dynamical systems composed of rigid and flexible bodies interconnected by simple and compound joints; and (ii) *Controls*: a toolbox for synthesis of nonlinear and adaptive control laws based on dynamic inversion methods. Both toolboxes include functions for generation of simulation models

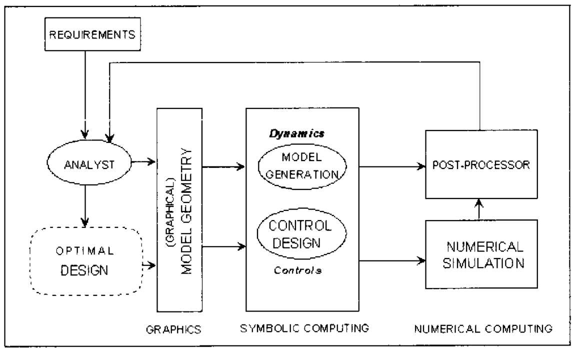


Fig. 1. Integrated system for modeling and design of nonlinear systems.

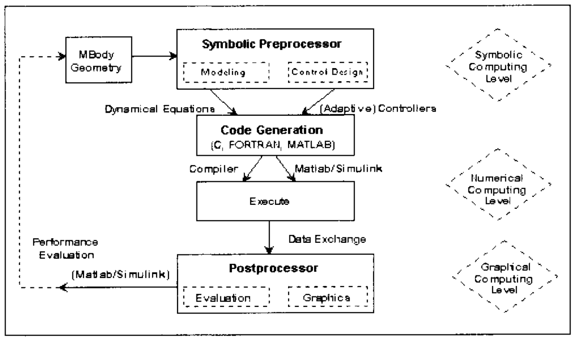


Fig. 2. Integrated symbolic-numeric modeling and design system.

A version of this paper was presented at the IEEE/IFAC Joint Symposium on Computer-Aided Control System Design, Tucson, March 1994. Blankenship, Ghanadan, and Polyakov are with the Department of Electrical Engineering and Institute for Systems Research, University of Maryland. Ghanadan's work was done in part while visiting the Department of Elect. Comp. Eng., University of California, Santa Barbara. Kwatny is with the Department of Mechanical Engineering, Drexel University, Philadelphia, PA. LaVigna is with Techno-Sciences, Lanham, MD.

in MATLAB/Simulink or C. These programs provide functions for the manipulation of dynamical models into standard formats and for the basic mathematical operations commonly encountered in analysis of nonlinear systems. They implement algorithms for adaptive and approximate nonlinear control and provide flexible numerical simulation of the closed-loop systems via automatic C or MATLAB code generation. We have used the packages to generate models for complex systems including a tracked, multiwheel vehicle (13 degrees of freedom) and to design adaptive, stabilizing controllers for several systems, including a conical magnetic bearing (18 states, one uncertain parameter). These case studies are summarized to illustrate the capabilities of the design system.

Modeling Multibody Systems

In this section we describe functions for modeling complex multibody structures. The software is organized into a package, called *Dynamics*, written in *Mathematica.* It generates models for certain classes of multibody systems interconnected by "joints." Both rigid and elastic bodies may be included in the system. Unlike many available programs (e.g., ADAMS, DYMAC, DADS, [12]) that focus on the assembly of pure simulation models, our tools generate *explicit* nonlinear equations of motion in the form needed for *control system design* and other analytical purposes. The models can be passed to other programs, including MATLAB/Simulink, for execution and analysis.

Computer derivation of the equations of motion for multibody systems has been previously considered by other investigators, including Leu and Hemati [24], Cetinkunt and Ittoop [6], and Sreenath [3]. Our approach extends that work in two important aspects. First, we admit a more general class of joint models in which the joint parameterization and all relevant joint kinematic relations are derived directly from the specific joint definition_as opposed to prescribing them beforehand. Second, we use Poincare's form of Lagrange's equations, which allows maximum freedom of choice for velocity coordinates [2]. That can substantially simplify the final model equations.

Our joint characterization distinguishes between "simple" and "compound" joints. Most joints with multiple degrees of freedom are realized physically as a sequence of joints each of which has one degree of freedom_a subclass that we call *compound joints.* When joints are modeled in this way, Poincare's equations are Lagrange's equations. Any compound joint is kinematically equivalent to a simple joint, and in our formulation both joint descriptions induce the same parameterization. When a compound joint is represented by its equivalent simple joint, the resulting Poincare's equations are much less complex than Lagrange's equations.

The *Dynamics* package creates fully nonlinear, explicit symbolic models of the form[1]

$$\dot{q} = V(q)p$$

$$M(q)\dot{p} + C(p,q)p + F(q,p) = Qp$$

where q is a vector of configuration coordinates, p is a vector of quasi-velocities, $M(q)$ is the system inertia matrix, and Q_p is the vector of generalized forces acting on the system in the p-coordinate frame. The models may be subjected to further symbolic processing for nonlinear model reduction, nonlinear control system design, linearization, etc.

[1]By "symbolic" we mean that the parameters and variables in the model may be left in symbolic form, in the sense used in computer algebra languages. There is no need to assign numerical values in the model construction phase.

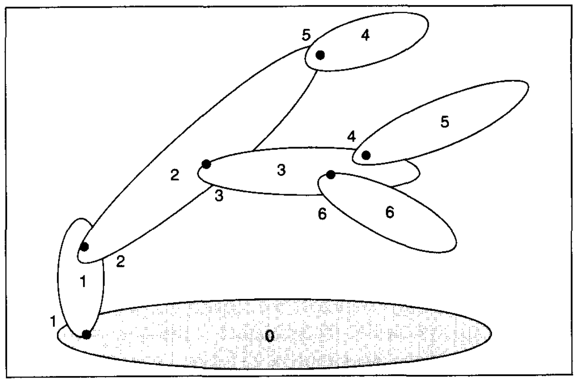


Fig. 3. A multibody system with tree topology. Joint numbers are in italic. The inertial reference frame is designated as body 0.

To use the *Dynamics* package, a user supplies defining data for individual joints and bodies, and the system structure. With this data it can compute the kinetic energy function and inertia matrix as well as the gravitational potential energy function. It can also compute the strain potential energy and dissipation functions associated with deformations of flexible bodies. Various kinematic quantities can be obtained as well, e.g., end-effector configuration or velocity as a function of joint and deformation parameters. To complete a dynamic analysis, the user must supply the remaining parts of the potential energy function. Generalized forces, including control signals, can be generated using other available functions in the package.

Data Structures

The following paragraphs describe the data structures used to create mathematical models and simulations for multibody systems.

Chains and Trees

The *Dynamics* package builds models for mechanical systems which have a *tree topology.* Chain structures are a special case. It can also accommodate algebraic and/or differential constraints so that systems involving closed loops or rolling can be modeled. Fig. 3 illustrates a tree, with bodies and joints numbered. Every system contains a base reference frame which is designated body "0". Otherwise, bodies and joints can be numbered arbitrarily.

The tree is composed of a set of chains. For instance, the tree in Fig. 3 contains three chains composed of the following sequences of bodies:

$$\{0,1,2,4\};\ \{0,1,2,3,5\};\ \{0,1,2,3,6\}$$

All subchains of any tree will start with body 0, so we need not list it. The body lists alone do not adequately define a tree. For instance, bodies 5 and 6 both connect to body 3, but they do so through different joints. This information can be provided by defining each subchain as an ordered list of pairs_each pair consisting of a body and its inboard joint: {inboard joint, body}. For the example of Fig. 1:

{{1,1},{2,2},{5,4}}; {{1,1},{2,2},{3,3},{4,5}}; {{1,1},{2,2},{3,3},{6,6}}

In summary, the system structure as a tree is defined by the data structure:

```
Tree = {list of subchains}
Subchain = ordered list of pairs
{inboard joint, body}= {{first inboard
joint, first body},...,{last inboard
joint, last body}}
```

Joints

Joints characterize relative motion between bodies. This motion is defined in terms of the relative motion of an outboard reference frame with respect to an inboard reference frame. The relative velocity between the two frames is a 6-dimensional vector, V, in which the first three components correspond to the relative angular velocity vector, ω, and the last three components correspond to the relative linear velocity, v. A joint may have r degrees of freedom. Associated with each such joint is a quasi-velocity vector, p, of dimension r, and a joint coordinate vector, q, of dimension r. The joint velocity vector, V, is related to its *quasi-velocity vector*, p, through a *joint map matrix*, $H(q)$:

$$V=H(q)p$$

A simple joint admits relative motion along axes fixed in one of its two frames so that its joint map matrix H is independent of the joint configuration parameters, q.

In the *Dynamics* package, joints are defined in terms of the relative motion of a sequence of reference frames. The relative motion between each pair of successive frames is characterized by the action of a simple joint. For simple joints, the *joint map matrix*, H, is the only defining data required.

The action of a general joint (or compound *joint*) consists of relative motion of a sequence of serially connected simple joints. A compound joint composed of k simple joints is defined in terms of the k simple joint map matrices, each of which is constant.

A k-frame compound joint with n degrees of freedom is defined by the data structure **{r, H,q,p}**, where:

- **r** = k-vector whose elements define the number of degrees of freedom for each simple joint with $n=r_1+ ...+r_k$;
- **H** = $[H_1..H_k]$, a matrix composed of the k joint map matrices of the simple joints;
- **q** = n-vector of joint coordinate names; and
- **p** = n-vector of joint quasi-velocity names.

Rigid Bodies

In chain and tree structures a rigid body interacts with other bodies through joints. One of these is naturally an *inboard joint* and all others are *outboard joints* (see Fig. 4). A body-fixed reference frame can be defined with its origin at the inboard joint location. This we call the *body frame*. With reference to the body frame; the following quantities can be defined: the *center of mass location*, all *outboard joint locations* and associated joint identification, and the inertia tensor about the center of mass. These quantities, along with the mass, define the body

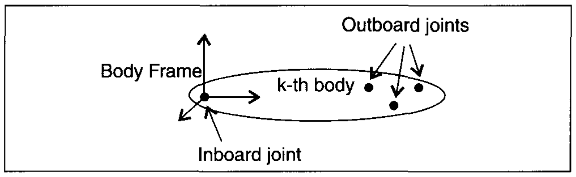


Fig. 4. A body fixed frame with origin at the inboard joint is associated with every rigid body.

Accordingly, a rigid body with k outboard joints is defined by the data structure

$$\{com, \{out_1, .., out_k\}, m, Inertia\}$$

where com is the center of mass location, out_i = {joint number, location} for the ith outboard joint, m is the mass, and $Inertia$ is the inertia tensor; (about the center of mass).

The *Dynamics* package also accommodates flexible bodies that satisfy the following conditions: (i) local deformations are small, so linear stiffness (quadratic strain energy) and dissipation (quadratic dissipation function) relations apply; (ii) body deformations can be characterized by a finite set of deformation coordinates; and (iii) body frame center of mass location and joint locations and orientations can be defined as affine functions of the deformation coordinates. Any flexible body model in which a modal representation of flexure is valid satisfies these assumptions. Even with these assumptions large *global* deformations are possible, in which case the body inertia matrix as represented in the primary body frame may be a function of the deformation coordinates.

A flexible body with k outboard joints and n deformation coordinates is defined by the data structure

$$\{C_{com}, \{out_1, .., out_k\}, m, \{M(x), B, K\}, x, v\}$$

where C_{com} is a matrix that defines the center of mass location; out_i = {joint number, C_{outi}}, where C_{outi} is a matrix that defines the orientation and location of the ith outboard joint; m is the mass; $M(x)$ is the inertia matrix, B is the dissipation matrix; K is the stiffness matrix; x is an n-vector of deformation coordinate names; and v is an n-vector of deformation velocity ($\dot{x}$) names.

Building Models

The following paragraphs describe the tools for creating mathematical models and simulations for multibody systems.

Kinematic Relations

The relative joint configuration, consisting of a general rotation and translation, is specified by a Euclidean configuration matrix, $X(q)$. As noted above, the joint velocity vector V is related to its quasi-velocity vector, p, through a joint map matrix, $H(q)$: $V=H(q)p$. In addition, the coordinate velocity vector $\dot{q}$ is related to the quasi-velocity vector by a square, nonsingular velocity transformation matrix $V(q)$

$$\dot{q} = V(q)p$$

The problem of joint modeling is the computation of the three matrices *V(q)*, *X(q)*, and *H(q)*. In the *Dynamics* package the required computations are carried out by the function Joints. Joints takes a set of joint definitions and returns corresponding lists of *V*, *X*, and *H*.

Building Systems

There are several alternatives for assembling system models as equations that can be simulated or analyzed. The most direct is to use the function CreateModel, which generates Poincare's equations. However, a user may want to examine intermediate results, such as some kinematic quantities or the system inertia matrix, or to develop relations other than the dynamical equations. In such cases, a step-by-step process is appropriate using more elemental constructions like Joints, TreeInertia, GravPotential, PoincareFunc, etc. The function CreateModel provides a shortcut for deriving the equations of motion. The equations are produced in the form

$$\dot{q} = V(q)p$$

$$M(q)\dot{p} + F(q,p) = Q_p$$

Its calling syntax is:

{V,X,H,M,F,p,q} =
CreateModel[JointLst,BodyLst,TreeLst,g,PE,Q]

where PE is the potential energy (constructed with other functions in the package). The function CreateModelSim produces the equations of motion in slightly different form which is more convenient for large systems, particularly if simulation code is required:

$$\dot{q} = V(q)p$$

$$M(q)\dot{p} + C(q,p)p + F(q,p) = Q_p$$

Its calling syntax is:

{V,X,H,M,Cqp,F,p,q} =
CreateModelSim[JointLst,BodyLst,TreeLst,g,PE,Q]

CreateModelSim is always used to assemble the model when simulation code is desired. In some applications, the matrix *C(q,p)* is of interest in its own right. Another difference between the two functions is that *V* is provided as a list of (joint) velocity transformation matrices, which is much more compact than when assembled into a system matrix.

Example: Steering Vehicle

Consider the simple ground vehicle shown in Fig. 5. The design problem is to develop a model for steering the vehicle along a prescribed trajectory.

The following Mathematica program generates the model equations for this simple system:

```
<Dynamics' (* Load the Modeling Pack-
age into Mathematica *)
(* Joint 1 *)
    r1={3}; q1={theta,x,y};
    p1={wth,vx,vy};
    H1={{0,0,0},{0,0,0},{1,0,0},{0,1,0},
    {0,0,1},{0,0,0}};
(* Joint 2 *)
    r2={1}; q2={delta}; p2={wdel};
    H2=Transpose[{{-
    s/Sqrt[1+s^2],0,1/Sqrt[1+s^2],0,0,0}
    }];
    JointLst={{r1,H1,q1,p1},{r2,H2,q2,p2
    }};
(* Body 1 *)
    cm1={0,0,0};out1={{2,{a,0,0}},{3,{-
    b,0,-R}}};
    I1=DiagonalMatrix[{Jxx,Jyy,Jzz}];
(* Body 2 *)
    cm2={s*R/2,0,-R/2};
    out2={{4,{s*R/2,0,-R}}};
    I2=DiagonalMatrix[{Ixx,Iyy,Izz}];
    BodyLst={{cm1,out1,m1,I1},{cm2,out2,
    m2,I2}};
    TreeLst={{{1,1},{2,2}}};
    ChnLst={{1,1},{2,2}};
    q={theta,x,y,delta};
    p={wth,vx,vy,wdel};
(* Joint Computations *)
    {V,X,H}=Joints[JointLst];
(* Front Tire Forces *)
    Force={0,0,0,0,-
    kappa*ArcTan[v4y/v4x],0};
    Vel-
    Names={w4x,w4y,w4z,v4x,v4y,v4z};
    TerminalNode=4;
    Q1=GeneralizedForce[ChnLst,Termi-
    nalNode,BodyLst,X,H,q,p,Forc e,Vel-
    Names];
(* Rear Tire Forces *)
    ChnLst={{1,1}};
    Force={0,0,0,F,-
    kappa*ArcTan[v3y/v3x],0};
    Vel-
    Names={w3x,w3y,w3z,v3x,v3y,v3z};
    TerminalNode=3;
```

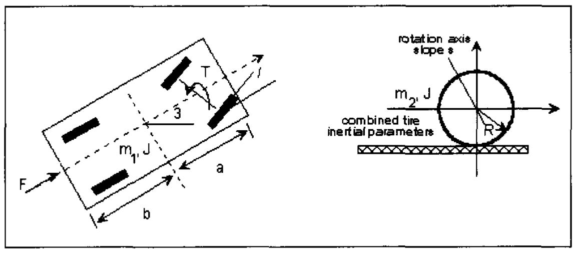


Fig. 5. Simple vhicle steering model. The center of mass is located at (x,y). The attitude is theta. The front wheels are rotated by angle delta about an axis of slope s. The slope is assumed small as are the tire inertial parameters.

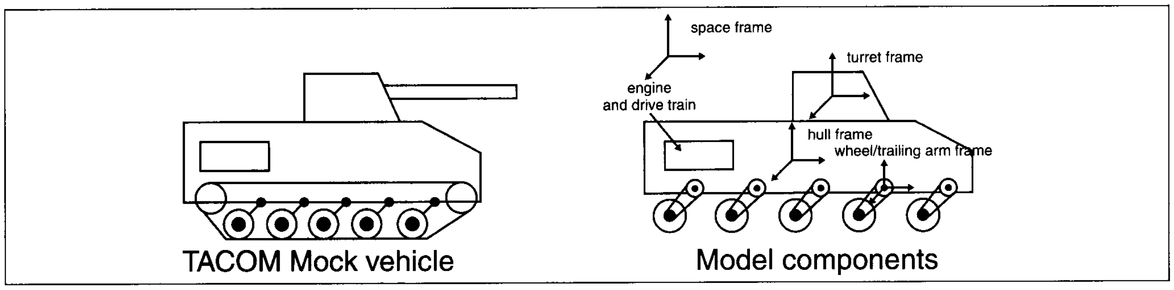


Fig. 6. Tracked vehicle modeled by the Dynamics package.

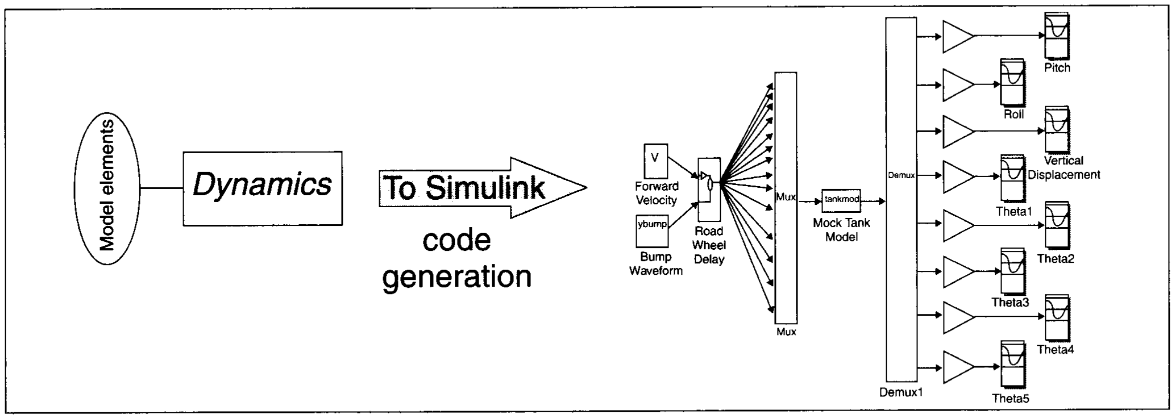


Fig. 7. Linking the Dynamics design package with MATLAB/Simulink.

```
    Q2=GeneralizedForce[ChnLst,Termi-
    nalNode,BodyLst,X,H,q,p,Forc e,Vel-
    Names];
(* Assemble Model *)
    Q3={0,0,0,T}; Q=Q1+Q2+Q3;
    {V,X,H,M,Cmat,Fsys,psys,qsys}=
    CreateModel-
    Sim[JointLst,BodyLst,TreeLst,g,0,Q,V
    ,X,H];
```

Under the simplifying assumptions that the tire mass m_2 is small (=0), and the rotation axis slope, s, is small (<<1), the equations take the form

$$\begin{bmatrix}\dot\theta\\ \dot x\\ \dot y\\ \dot\delta\end{bmatrix}=\begin{bmatrix}1&0&0&0\\0&\cos\theta&-\sin\theta&0\\0&\sin\theta&\cos\theta&0\\0&0&0&1\end{bmatrix}\begin{bmatrix}\omega_\theta\\ v_x\\ v_y\\ \omega_\delta\end{bmatrix}$$

$$\begin{bmatrix}I_{zz}+J_{zz}&0&0&I_{zz}\\0&m_1&0&0\\0&0&m_1&0\\I_{zz}&0&0&I_{zz}\end{bmatrix}\begin{bmatrix}\dot\omega_\theta\\ \dot v_x\\ \dot v_y\\ \dot\omega_\delta\end{bmatrix}+\begin{bmatrix}0\\ m_1 v_y\omega_\theta\\ -m_1 v_x\omega_\theta\\ 0\end{bmatrix}+\begin{bmatrix}f_1\\ f_2\\ f_3\\ f_4\end{bmatrix}=0$$

The details of the functions $f_1,...,f_4$ are omitted (see [19] for details).

Suppose our objective is to solve the problem of steering the vehicle along a path of constant radius at constant speed V_d. There are several ways of formulating this problem. One common approach is to replace the constant radius condition by the requirement that the angular velocity ω_θ is a constant, say ω_d. This leads to a constant curvature path of radius $R=V_d/\omega_d$. From a control theoretic point of view this suggests defining two output variables

$$y_1=v_x^2+v_y^2-V_d^2$$

$$y_2=\omega_\theta-\omega_d$$

and designing a controller to cause these outputs to track the prescribed trajectory. This is a simple problem in nonlinear control, readily solved by feedback linearization methods (for example). A key issue in this design method is the construction of "normal forms" and analysis of the (nonlinear) "zero dynamics." In the next section we shall use the LocalZeroDynamics function in the *Controls* package to compute the *zero dynamics* of the system relative to these two outputs and the two controls *(T,F)*. The (local) analysis of the zero dynamics shows that the system is inherently non-minimum phase.

Application: Modeling a Multi-Wheeled, Tracked Vehicle

The previous example illustrates some of the basic features of the *Dynamics* package. In [23], the package was used to

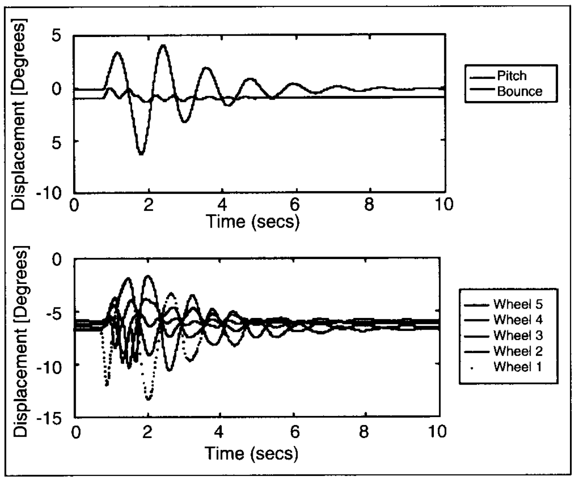


Fig. 8. Simulation of the performance of the tracked vehicle using MATLAB/Simulink.

compute the equations of motion of the tracked vehicle shown in Fig. 6. This vehicle has ten wheels and a torsion bar based suspension. Its basic components include the hull, turret, engine and drive train, and the wheels and tracks. In our analysis the hull is assumed to bounce, roll, and pitch, and each of the ten wheels has one degree of freedom (a revolute joint). Thus, the (rigid body) model has 13 degrees of freedom. (The case when the hull is a flexible body was also treated in [23].) Using the *Dynamics* package, the model equations were generated as a MATLAB (C-code) MEX file and compiled with a MATLAB compatible C-compiler. The model generation process requires about one hour on a 486/33 MHz IBM-compatible PC.

Once the MEX file is generated, the user has three options for exercising the model in MATLAB/Simulink: (i) from a MATLAB script using one of MATLAB's ODE solvers; (ii) from a MATLAB script using one of Simulink's ODE solvers; or (iii) from Simulink's graphical interface using one of Simulink's ODE solvers. A schematic of the last option is shown in Fig. 7. A simulation of the vehicle traversing a bump is shown in Fig. 8.

Design of Nonlinear Control Laws

Given the capability to generate models with embedded (control) forces and torques, the natural complement is a system for the computation of effective control laws. Since we are interested in designing the architecture of the control system as well as in crafting specific algorithms, it is important to use symbolic computing methods in the design process. As the examples in the previous section indicate, typical systems of interest are highly nonlinear, and their models are too complex to be analyzed by hand. While there has also been a large body of work on software for the design and analysis of linear control systems, there has been much less work on tools for the design of nonlinear control systems. In this section we shall describe one approach to the synthesis of such tools starting from the geometric formulation of nonlinear control theory.

In 1987 O. Akhrif developed the first computational tools for the design of nonlinear control systems using symbolic computing (Macsyma) [1]. This work was inspired by the work of J.P. Quadrat and his colleagues on the use of Macsyma (and Prolog) in the treatment of optimal stochastic control problems [7]. The work here builds on the tools developed by Akhrif. It employs new techniques of nonlinear adaptive control [9,10] and performance evaluation by simulation.

The *Controls* package includes several easy-to-use functions for computation of mathematical objects frequently encountered in control system analysis, such as Lie derivatives, Lie brackets, and controllability distributions, along with functions for synthesis (e.g., the dynamic extension algorithm, decoupling control algorithms of Hirschorn and Singh, adaptive and approximate linearization algorithms of Ghanadan and Blankenship, and Kokotovic and Kanellopoulos, etc.), as well as functions for automatic C and MATLAB code generation.

The tools presented here have been applied to realistic nonlinear problems for which hand calculation is not feasible and for which conventional tools (e.g., MATLAB, MatrixX, etc.) are not well suited. Earlier versions of this package were used to design controllers for an active automotive suspension and a magnetic levitation system [4,11]. In this section we illustrate the power of the tools by designing an adaptive tracking controller for conical magnetic bearings, an 18-dimensional system with complicated nonlinear dynamics [25]. At the end we return to the vehicle steering problem defined above and show how to compute the (local) zero dynamics of the model produced by the *Dynamics* package.

Controls Package Description

The *Controls* package deals with MIMO nonlinear systems in the following form:

$$\dot{x} = f(x,t;\theta) = f(x;\theta) + g(x;\theta)u$$

$$y = h(x)$$

where

$$x \in R^n, u \in R^m, y \in R^l, f: R^n \to R^n, g: R^n \to R^{n\times m}; h: R^n \to R^l$$

and Θ is a vector of (unknown) parameters. The tools in the *Controls* package fall into four general categories: (i) basic analysis tools; (ii) model representation; (iii) controller design; and (iv) simulation.

Basic Analysis Tools

There are several mathematical operations that occur frequently in nonlinear control systems design. Although these operations involve straightforward mathematics, actual computation is tedious and time-consuming, especially for large (n>5 state) systems. The most common of these mathematical tools are Lie derivatives and Lie brackets. The Lie derivative of a function h relative to a function f is defined by

$$L_f^0 h = h \, L_f^1 h = L_f h = \frac{\partial h}{\partial x} \cdot f$$

$$L_f^n = L_f L_f^{n-1} h, n = 1, 2, \ldots$$

This algorithm may be expressed in *Mathematica* as the following sequence of "rules":

```
LieDerivative[f_, h_, x_,0]:=h
LieDerivative[f_,h_,x_]:=Dot[Ja-
cob[h,x], f]
LieDerivative[f_,h_,x_,1]:=LieDeriva-
tive[f,h,x]
LieDerivative[f_,h_,x_,n_]:=LieDeriva-
tive[f,LieDerivative[f, h,x], x,n-1]
```

Here Jacob[h,x] is the Jacobian matrix of h with respect to x and Dot is a *Mathematica* function for multiplying arrays (matrices). The definitions of LieDerivative make use of *Mathematica's* pattern checking and conditional definition capabilities to ensure that both the arguments and the answers make sense.

Mathematica has the capability to use "pure" functions in rules. This is a particularly convenient construction for creation and maintenance of control system models. The power of this feature can be seen in the definition of the Jacobian in *Mathematica.*

The first line in the following is a usage statement associated with the help system in *Mathematica.* The second line is the computation of the gradient.

```
Grad::usage="Grad[f,varlist] computes
the Grad of the function f with re-
spect to the list of variables varist."
Grad[f_,var_List]:=D[f,#]& /@ var
```

In the definition of Grad, the expression D[f,#]& is a *pure* (un-named) function. The symbol D stands for derivative; so D[f,x] is the derivative of f with respect to a (single) variable x. To compute the gradient of a scalar function of a vector, we must compute its derivative with respect to each element of the vector. This is accomplished by "mapping" the operation "take the derivative of f with respect to a variable" (this is the meaning of the expression D[f,#]&). The symbol & stands for a "name" that one might assign to the function "take the derivative." However, since we will only use the pure function once, we do not need to name it. Similarly, we do not need to name the variable that is its argument, so the symbol # is used as a place marker.

Arguments to function definitions in *Mathematica* are of the form h[x_]:=x^2, which means any symbol substituted for the place holder x_ is raised to the second power. The form var_List means the argument must be a list, a form of data verification provided in *Mathematica.*

The symbol /@ stands for the *Mathematica* operation Map; so we could have written the definition as

```
Grad[f_,var_List]:=Map[D[f,#]&,var]
```

The use of pure functions and the capability to map functions over sets of arguments are powerful constructions which increase the expressive power of *Mathematica* programs. Map[] is especially useful in avoiding procedural programming constructions. The use of Map[] in the definition of Grad[] illustrates the capability of *Mathematica* to treat functions as objects like symbols or numbers and use them as arguments to other functions.

We use two lines (rules) to define the Jacobian of a function with respect to a vector. The first handles the case when the function is a vector function of a vector argument. The second handles the case of scalar functions (of vector arguments). These may be regarded as rules for the computation. *Mathematica* uses a kind of pattern matching to find the case that applies.[2]

```
Jacob[f_List,var_List]:=Outer[D,f,var]
/;VectorQ[f]
Jacob[f_,var_List]:=Grad[f,var]
```

Outer[] is a built-in *Mathematica* function which provides a generalized outer product. The test VectorQ[f] defined by the condition symbol " /;" checks that f is a vector. If the test succeeds, this rule is used. If not, the next one is used.

The next function illustrates the use of condition checking in *Mathematica* in more detail. The symbol && is logical "and." In the first rule, we check that the functions are vector valued, that their lengths are identical (==), and that the lengths equal the length of the vector of variables. If this compound test succeeds, the rule is used.

```
LieBracket [f_List,g_List,var_List]:
=(Jacob[g,var] . f -Jacob[f,var] . g /;
    VectorQ[f] && VectorQ[g]&&
    Length[f]==Length[g]==Length[var])
    (* Test the data *)
LieBracket[f_,g_,var_List]:=
    Jacob[g,var] f - Jacob[f,var] g
```

The next sequence illustrates the recursive power of the language to define the Ad operator. (We omit the vector cases.)

```
Ad::usage= "Ad[f,g,varlist,n] computes
    the nth Adjoint of the functions
    f,g with respect to the variables
    varlist."
Ad[f,g,var,0]=g
Ad[f,g,var,n]=
    LieBracket[f,Ad[f,g,var,n-1],var]
Ad[f,g,var]=Ad[f,g,var,1]
```

Using these functions, we can express the Hunt-Su-Meyer conditions [16] in Mathematica functions.

```
Control-
labilityDistribution[f_,g_,var_List]:=
        Module[{k},
    Table[Ad[f,g,var,k],
    {k,0,Length[var]-1}]]
    Controllable[f_,g_,var_List]:=
If[Rank[Control-
labilityDistribution[f,g,var]]
    ==Length[var],True,False];

FeedbackLinearizable[f_,g_,var_List]:=
        Module[{cm,cm1,k},
cm=Table[Ad[f,g,var,k],
{k,0,Length[var]-1}]
cm1=Drop[cm,-1];(* drop last element *)
If[Rank[cm]==Length[var]
        (* system is controllable *)
    && Involutive[cm1,var],
    True,False]];
```

The Module[] construction permits the use of local variables in the definition of functions. We use the *Mathematica* Table[] function to construct a set of derived vector fields. The function Involutive[] checks that a set of vector fields is involutive, that is, closed under the Lie Bracket.

```
Involutive[f_List,var_List]:=
```

[2]In the code examples that follow, we present selected components. In some cases additional code is required to complete the definition.

```
        Module[{k,h,vec},
            k=Length[f];
        h=Table[LieBracket[f[[i]],
        f[[j]],var],
            {i,1,k},{j,i+1,k}];
    vec=Union[Flatten[h,1],f];
        If[Rank[vec]Rank[f],False,True]]
```

In this expression the notation f[[i]] takes the element of the list (vector) f. Union and Flatten are *Mathematica* functions for manipulating lists.

With these simple operations we can define several useful functions for the analysis of nonlinear control systems; including RelativeDegree[f,g,h,x], VectorRelativeDegree[f,g,h,x], DynamicInverse[f,g,h,x], ZeroDynamics[f,g,h,x], etc. These functions were implemented by translating into *Mathematica* notation the definitions found in standard nonlinear control texts [16,26].

Model Representation

While it is natural to work with conventional function definitions for the vector fields that occur in nonlinear control problems, it is more useful to create a "data structure" for maintaining models. The pure function construction in *Mathematica* is an effective means for accomplishing this.

System. The data defining the controlled nonlinear system is stored as a *Mathematica* data object with "head" System and the associated structure

```
System[f,g,h,x,y,u,theta,analysisdata]
```

where f, g, and h are *Mathematica* functions, x, y, and u are lists containing labels of the states, the outputs and the inputs, respectively, and theta is a list of uncertain parameters found in f, g, or h. As various function are applied to the System, their results are appended in the list analysisdata.

The System object provides an economical and efficient organization for often bulky and unenlightening expressions.

MakeSystem. Constructing a System object is made relatively easy by the MakeSystem function which has the following syntax:

```
MakeFunction[f,g,h,x,u,y,u,theta]
```

Although generally the components of the system model (f,g,h) are stored as pure functions, the first three arguments of MakeSystem can also be given as ordinary Mathematica expressions. MakeSystem, if necessary, automatically converts the f,g,h to functions and makes sure the dimensions agree before returning a valid System object. For example, the data

```
var={x1[t],x2[t]};
f:={#[[2]],2 omega xi (1- mu #[[1]]^2
) #[[2]] -omega^2 #[[1]]}& ;
g:={{0},{1}}& ;
h:={#[[1]]}& ;
sys=MakeSystem[f,g,h,var];
```

constructs the equations of the controlled Van der Pol Oscillator with output

$$\dot{x}_1(t) = x_2(t)$$

$$\dot{x}_2(t) = 2\omega\xi(1-\mu x_1(t)^2)x_2(t) - \omega^2 x_1(t) + u(t)$$

$$y(t)=x_1(t)$$

ShowSystem and GetResults. In order to examine the contents of the System object and extract the results that were appended to it by previous analyses, two functions are provided: ShowSystem and GetResults.

ShowSystem[sys], where sys is a valid System, will display the data of the system, f,g,h, etc., as well as a list of any functions which have been applied and whose results are contained in the data portion of this System.

GetResults[sys, "analysis"] will return the results of function called analysis which has been applied to sys earlier. For example, to extract results of Singh from demosys one would use

```
GetResults[demosys, "Singh"]
```

If the results are not contained within the System a string "Not found" is returned.

Design Functions

In this section we describe functions for design of nonlinear control laws, including adaptive and approximate methods.

Hirschorn and Singh. The *Controls* package includes two functions for partial (input-output) feedback linearization via construction of right inverse systems using the algorithm of Hirschorn [14] and its extension by Singh [30]. Since Singh's algorithm is applicable to a wider class of systems, we discuss its implementation. (Hirschorn's algorithm is implemented in a function called Hirschorn with syntax identical to Singh.)

The recursive nature of Singh's algorithms is well suited for implementation in *Mathematica*. The command to apply Singh's algorithm is

```
Singh[sys,opts]
```

where sys is a valid System and opts are options described below. Singh will append the following to the System

```
SinghResults[D,c,K,z]
```

where $z=c+Du$ and K defines the relationship between z and y and its derivatives,

$$z = K\left(y_1^{(1)},\dots,y_1^{(r_1-1)},y_2^{(1)},\dots,y_m^{(r_m-1)}\right)^T$$

The control to track y is given by $u_t=D^{\Diamond}(z-c)$ where $D^{\Diamond}$ denotes the pseudoinverse of D.

Several options are available for Singh. ScreenOutput→False will disable almost all screen output. ReturnObject- List, instead of returning the original System with the results appended, will simply return a list of the results.

AdaptiveTracking. The function Singh forms the foundation for AdaptiveTracking, which implements the adaptive algorithm of Ghanadan and Blankenship [9], basically an adaptive observer. Given a System object with a list of uncertain parameters, Θ, AdaptiveTracking computes the control law and the parameter update law to track a desired trajectory.

The syntax for AdaptiveTracking is

```
AdaptiveTracking[sys, poles, adgain,
opts]
```

where sys is a valid System object, poles is a list of $1\times r_i$ lists with r_i being the ith element of the vector relative degree for the system, and adgain is the adaptive gain used in constructing the parameter update law. adgain can be supplied in two forms: a constant which sets the same gain for all parameters or a vector in which the ith element sets the gain for the ith parameter.

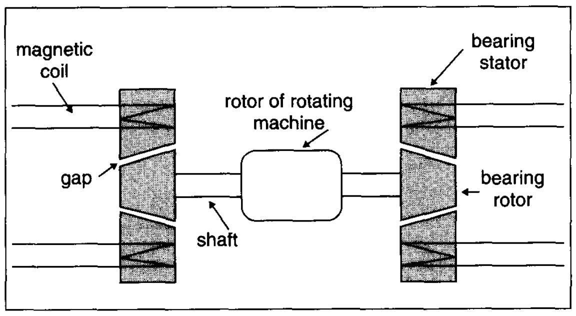


Fig. 9. Magnetic bearing system (after Mohamed and Emad 1992).

As in Singh, options for AdaptiveTracking include ScreenOutput and ReturnObject. In addition, Simulate→MATLAB option prepares the output to be simulated using MATLAB as described below.

ApproximateAdaptiveTracking. Results of approximate feedback linearization theory [13,17] are useful design alternatives to the more restrictive schemes based on exact (partial) feedback linearization. This scheme assumes milder involutivity and invertibility restrictions and can be applied to *slightly non-minimum phase* nonlinear systems as well. In [10] an adaptive approximate tracking and regulation scheme was presented for nonlinear systems with uncertain parameters. The function ApproximateAdaptiveTracking implements this scheme as a *Mathematica* function with syntax:

```
ApproximateAdaptiveTrack-
ing[sys,poles,observerpoles,
UpdateLawGain]
```

where observerpoles is a list of desired observer poles for the adaptive scheme of [10]. The tracking function searches for linear functions of unknown/uncertain parameters theta specified in the dynamics. The regulation version of this algorithm can handle parameters that do not appear linearly in the system.

Simulation

C, MATLAB, and Simulink Code Generation

Included in the package are two functions for automatic code generation in C and FORTRAN. These functions automatically write a subroutine compatible with the Numerical Recipes [28] integrator, odeint, compile the program, execute it and return the results to *Mathematica.* The following *Mathematica* command line will execute the operations listed above:

```
SimulateC[sys, rules, ic, tfin, "Adap-
tiveTracking", tol]
```

where sys contains results of AdaptiveTracking, rules is a list of substitutions which are made before simulation is executed, ic are the initial conditions and tol is an optional tolerance specification.

Simulate. Functions called Simulate, and MATLABSimulate are included in the package to provide simulation capabilities in MATLAB. This is important for large systems, like the magnetic bearing described in the following section. Due to memory limitations it is not possible to analyze such a large model using *Mathematica* alone. For example, in computing a control law for the conical magnetic bearing, the function Singh found the 5x8 decoupling matrix, D, which occupied 1.6 Kb (ASCII) and its pseudoinverse, $D^{\Diamond}$, which occupied 3.87 Mb (ASCII). Thus, $D^{\Diamond}$ was too large to be manipulated, and the control law, when saved as ASCII text, was approximately 16 Mb. Consequently, straight forward inclusion of the control and parameter update laws into C or Fortran code was impractical for this application.

MATLABSimulate writes a MATLAB function which at each time instant evaluates the components of the control law, numerically computes the pseudoinverse of D using the MATLAB function pinv and then performs the necessary matrix multiplications and additions to find the control. Besides allowing simulation for large systems, linking to MATLAB in this way provides extra flexibility in selecting time limits, tolerance, and initial conditions without the need to recompile every time a change is made. The disadvantage of this method is slower computation time.

If the simulation is to be performed using MATLABSimulate, the option

```
Simulate -> MATLAB
```

must be used when performing AdaptiveTracking, e.g.,

```
AdaptiveTracking[sys,poles,adgain,MAT-
LABSimulate->
True]
```

Next, we need to form the substitution rules for desired output trajectory and its derivatives as well as the actual output and its derivatives. The latter can be accomplished using the function BuildSubRules with the following syntax:

```
BuildSubRules[sys,vectorrelativedegree]
```

The vector relative degree is displayed in the course of running Singh or it can be computed using VectorRelativeDegree. The substitution rules for the desired output and its derivatives must be provided by the user.

Finally, we can automatically write a MATLAB function for simulation using

```
Simulate[sys,"MATLAB","dir","file-
name",rules]
```

The MATLAB function will be stored in the file called filename.m in directory dir and can be integrated using standard MATLAB integrators, e.g. ode45. In fact, two options are available to use MATLAB to simulate systems. If the option MATLAB is selected, then Simulate generates a MATLAB function that will simulate the system using the MATLAB ODE solvers. If the option Simulink is selected, then Simulate generates a file that generates a Simulink block diagram, and the simulation can be run from the Simulink environment.

Application: Adaptive Control of a Conical Magnetic Bearing

Conical magnetic bearings have been the subject of active research recently. They provide a non-trivial test for linear and nonlinear design methodologies. For the bearing configuration shown in Fig. 9 we use the model derived by Mohamed and Emad [25] which has 18 states, eight controls, eight outputs, and several disturbances. We include an uncertain parameter representing rotor angular velocity.[3] Using the functions in the *Controls* package, we first model the magnetic bearings as a System object

[3]Mohamed and Emad did not consider parametric uncertainty or adaptive control.

and then design and simulate a nonlinear adaptive control which achieves asymptotic tracking.

Model

The following *Mathematica* script defines the model based on the analysis in [25] First, we define the right-hand side of $\dot{x} = f(x,u,t;\theta)$

```
fl1:= k x11[t]*x11[t]*(1+ 2(D0 +
x1[t])/(pi*h))
fl2:= k x12[t]*x12[t]*(1+ 2(D0 -
x1[t])/(pi*h))
 fr1:= k x13[t]*x13[t]*(1+ 2(D0 +
x2[t])/(pi*h))
fr2:= k x14[t]*x14[t]*(1+ 2(D0 -
x2[t])/(pi*h))
fl3:= k x15[t]*x15[t]*(1+ 2(D0 +
x3[t])/(pi*h))
fl4:= k x16[t]*x16[t]*(1+ 2(D0 -
x3[t])/(pi*h))
fr3:= k x17[t]*x17[t]*(1+ 2(D0 +
x4[t])/(pi*h))
fr4:= k x18[t]*x18[t]*(1+ 2(D0 -
x4[t])/(pi*h))
ran=2R/(mu0*A*N); cos=Cos[sigma];
mg=m*g/2;
H1=((l*l/Jy) + 1/m) cos; H2=((l*l/Jy) -
1/m) cos;
rhs1:=x6[t];
rhs2:=x7[t];
rhs3:=x8[t];
rhs4:=x9[t];
rhs5:=x10[t];
rhs6:= alpha/(2m)(x1[t]+x2[t])-p
Jx/(2Jy)(x8[t]-x9[t])-
-H1((fl1-fl2)cos-mg)+H2((fr1-fr2)co s-
mg);
 rhs7:= alpha/(2m)(x1[t]+x2[t])+p
Jx/(2Jy)(x8[t]-x9[t])-
H1((fr1-fr2)cos-mg)+H2((fl1-fl2)cos -
mg);
rhs8:= alpha/(2m)(x3[t]+x4[t])+p
Jx/(2Jy)(x6[t]-x7[t])—H1(fl3-
fl4)cos+H2(fr3-fr4)cos;
rhs9:= alpha/(2m)(x3[t]+x4[t])-p
Jx/(2Jy)(x6[t]-x7[t])—H1(fr3-
fr4)cos+H2(fl3-fl4)cos;
rhs10:= -beta/m x5[t]-2 gamma/m
x10[t]+ @PTEXT =
+Sin[sigma]/m((fl1+fl2+fl3+fl4)-
(fr1+fr2+fr3+fr4));
rhs11:= ((43.284433+u1) - ran (D0 +
x1[t]) x11[t])/N;
rhs12:= ((44.208506+u2) - ran (D0 -
x1[t]) x12[t])/N;
rhs13:= ((43.284433+u3) - ran (D0 +
x2[t]) x13[t])/N;
rhs14:= ((44.208506+u4) - ran (D0 -
x2[t]) x14[t])/N;
rhs15:= ((44.208506+u5) - ran (D0 +
x3[t]) x15[t])/N;
```

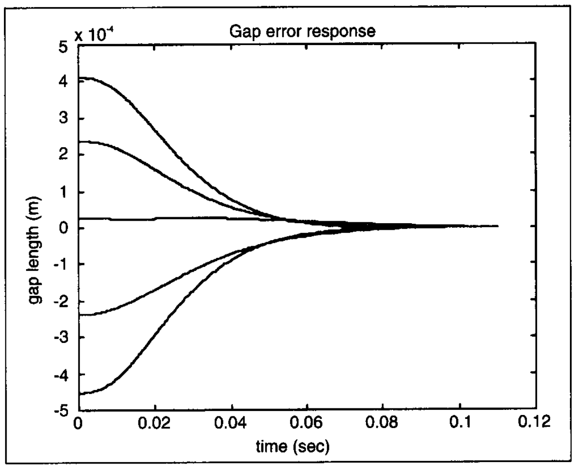


Fig. 10. Gap deviations with p=10^5 and initial parameter error of 10%.

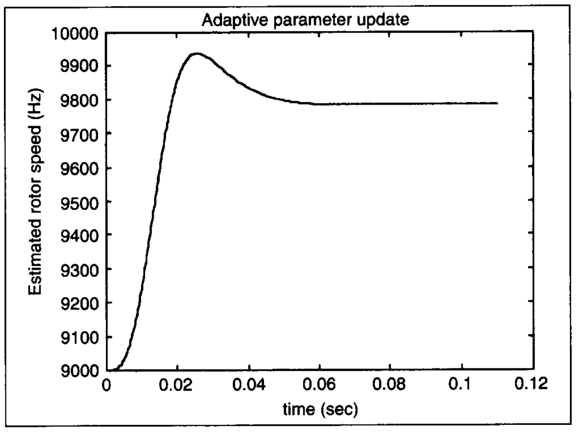


Fig. 11. Parameter update with p=10^5 and initial parameter error of 10%.

```
rhs16:= ((44.208506+u6) - ran (D0 -
x3[t]) x16[t])/N;
rhs17:= ((44.208506+u7) - ran (D0 +
x4[t]) x17[t])/N;
rhs18:= ((44.208506+u8) - ran (D0 -
x4[t]) x18[t])/N;
rhs:={rhs1, rhs2, rhs3, rhs4, rhs5,
rhs6, rhs7, rhs8, rhs9, rhs10,
rhs11, rhs12, rhs13, rhs14, rhs15,
rhs16, rhs17, rhs18};
Erhs:=Expand[rhs];
```

Then we identify $f(x;\Theta)$ and $g(x;\Theta)$ from $f(x, u, t; \Theta)$ using the fact that u appears linearly.

```
g1:=Coefficient[Erhs,u1,1];
g2:=Coefficient[Erhs,u2,1];
g3:=Coefficient[Erhs,u3,1];
g4:=Coefficient[Erhs,u4,1];
g5:=Coefficient[Erhs,u5,1];
g6:=Coefficient[Erhs,u6,1];
```

```
g7:=Coefficient[Erhs,u7,1];
g8:=Coefficient[Erhs,u8,1];
g:=Trans-
pose[{g1,g2,g3,g4,g5,g6,g7,g8}];
 f:=Erhs-g.u;
x:={x1[t],x2[t],x3[t],x4[t],x5[t],x6[t],
x7[t],x8[t],x9[t],x1 0[t],
x11[t],x12t],x13[t],x14[t],x15[t],x16[t]
,x17[t],x18[t]};
y:={y1[t],y2[t],y3[t],y4[t],y5[t]};
u:={u1[t],u2[t],u3[t],u4[t],u5[t],u6[t],
u7[t],u8[t]};
```

To speed up the calculation, wherever possible we substitute numerical values for any parameters that are known with certainty. In this case, Values is the list of substitution rules. The *Mathematica* symbol /. is a shorthand notation for the function Substitute. Chop removes values below a given precision level (due to numerical errors).

```
nf=Chop[f/.Values]; ng=Chop[g/.Val-
ues]; nh=Chop[h/.Values];
```

Using MakeSystem we create a valid System object:

```
magbear=MakeSystem[nf,ng,nh,x,y,u,{p}];
```

Control System Design

Given the model, we apply AdaptiveTracking with options which prepare the output for MATLAB simulation and suppress output to the screen:

```
poles=Table[-10^3, {5},{3}];
adgain=10^4;
magbear=AdaptiveTracking[ magbear,
adgain, poles,
          Simulate-MATLAB,
          ScreenOutput-False];
```

Throughout the above manipulations the rotor angular velocity was allowed to remain in its symbolic form, p. In general, the angular velocity may not be known with certainty and we treat p as an uncertain parameter. The parameter update law computed by AdaptiveTracking will allow us to track the desired output.

Since the outputs are the deviation from gap equilibrium value, our goal is to track zero. Therefore, we define the desired output and its derivatives

```
ydes[1]=0;
ydes[2]=0;
ydes[3]=0;
ydes[4]=0;
ydes[5]=0;
outdrule=Table[outd[i][y]-
D[ydes[i],{t,j}],{i,5},{j,0,3}]
```

Next, we use BuildSubRules to write *y(t)* and its derivatives as functions of *x*,

```
thetarule=Thread[p,p^2}-
thetabar1[t],thetabar2[t]]
yrule=BuildSubRules[mag-
bear,{3,3,3,3,3},thetarule]
```

Finally, we combine the above rules with the substitution rule for the actual value of p,

```
rules=Join[outdrule, yrule, {p-10^4}];
```

and write a MATLAB function to simulate the controlled system,

```
MATLABSimulate[magbear,"~/MagBear/",
"magsim",rules];
```

The following MATLAB code simulates the magnetic bearings,

```
ic=[zeros(1,10),
.002061,.002105,.002061,.002105,.002105,
.002105,.002105,.002 105, 0,0];
tfin=1; tstart=0;
[t,state]=ode45('magsim', tstart, tfin,ic, 1.e-7,1);
```

The simulation was performed for values of p ranging from 10 to 10^5. The nonlinear control law stabilized the system with initial errors on the order of the equilibrium gap length (0.5mm), a substantial improvement over the results obtained using linear controls in [25]. Figs. 10 and 11 show stabilization of the outputs and convergence of the parameter update with random initial conditions and parameter error of 10%.

Additional details of the analysis of this system can be found in [27].

Computing the Local Zero Dynamics

To illustrate the integrated use of the *Dynamics* and *Controls* packages, we return to the problem of steering the car. Recall that the equations for the car following a prescribed trajectory were generated by the *Dynamics* package. We use the *Controls* package to compute the zero dynamics for the case of motion along a straight path. The vector relative degree is found to be [1, 1]. Therefore, the zero dynamics involve three first-order differential equations in the zero dynamics "state" variables. The zero dynamics are computed using the following program.

```
<Controls' (* Load Nonlinear Control
Package *)
<GeoTools' (* Load Differential Geome-
try Tools used in Controls *)
(* compute relative degree *)
    ro=VectorRelativeOrder[f,g,h,var];
(* compute feedback linearizing/decou-
pling control *)
    {R1,R2,R3,R4,u}=IOLinear-
    ize[f,g,h,var];
(* compute linearizable coordinates *)
    z=NormalCoordinates[f,g,h,var,ro]
    /.{wd-0};
(* shift origin to point of interest *)
    {f,g,h,u,z}={f,g,h,u,z}
    /.{x2-x2+Vd};
(* compute zero dynamics *)
    u0=u/.{v1-0,v2-0};
    f0=LocalZeroDynamics
    [f,g,h,var,u0,z];
(* linearize zero dynamics and deter-
mine stability of origin *)
    Anu=Jacob[f0,{w1,w2,w3}]
    /.{w1-0,w2-0,w3-0,b-a+nu};
    Eigenvalues[Anu/.{nu-0,s-0}]
```

Up to fourth order terms the zero dynamics are defined by the equations

```
w1dot={w2}
w2dot={(kappa*(2*a + R*s)*w1) /
(2*Izz) - (kappa*(a +
R*s)*w1^3) / (2*Izz) + (kappa*(-2*a +
2*b -R*s)*w3) / (2*Izz*Vd) + (kappa*(-
2*a + 2*b - R*s)*w3^3) / (12*Izz*Vd^3)
```

```
+ w1^2*((kappa*(a + R*s)*w3) /
(2*Izz*Vd)) +
w2*((a*kappa*R*s)/(2*Izz*Vd) +
(a*kappa*R*s*w1*w3) / (2*Izz*Vd^2) -
(a*kappa*R*s*w3^2) / (4*Izz*Vd^3)
+w1^2*(-(a*kappa*R*s)/(2*Izz*Vd))}
w3dot={(kappa*w1) / m1 - (kappa*w1^3)
/ (2*m1) -(2*kappa*w3) / (m1*Vd) -
(kappa*w3^3) / (3*m1*Vd^3) +
w1^2*((kappa*w3) / (2*m1*Vd)) +
w2*((kappa*R*s) / (2*m1*Vd) +
(kappa*R*s*w1*w3) / (2*m1*Vd^2) -
(kappa*R*s*w3^2) /  (4*m1*Vd^3) -
@PTEXT = (kappa*R*s*w3^4)/(16*m1*Vd^5)
+ w1^2*(-(kappa*R*s)/(2*m1*Vd)))}
```

We can test the stability of the equilibrium point w=0 by examining the linearized zero dynamics computed at the end of the program

$$\dot{w} = \begin{bmatrix} 0 & 1 & 0 \\ \frac{\kappa(2a+Rs)}{2I_{zz}} & \frac{a\kappa Rs}{2I_{zz}V_d} & \frac{2\kappa(b-a)-\kappa Rs}{2I_{zz}V_d} \\ \frac{\kappa}{m_1} & \frac{\kappa Rs}{2m_1 V_d} & \frac{-2\kappa}{m_1 V_d} \end{bmatrix} w$$

The eigenvalues are readily obtained but they are lengthy functions of the parameters. Some insight is obtained, however, by examining the special case, a=b and s=0, in which case the eigenvalues simplify to

$$\lambda_1 = \frac{2\kappa}{m_1 V_d}, \lambda_{2,3} = \pm \frac{\sqrt{a\kappa}}{\sqrt{I_{zz}}}$$

Hence, we see that the zero dynamics are unstable. Because the eigenvalues vary smoothly as a function of parameters, this situation will be true for a-b and s small, but not necessarily zero. Furthermore, since I_{zz} is small, $\lambda_{2,3}$ are a pair of "parasitic" zeros, one of which is far into the right half plane, the other to the left. These locations may or may not make the vehicle difficult to control (by an experienced driver).

The technique for computing the (local) zeros of a nonlinear control system is described in [19]. Some earlier work which indicated the difficulty of this computation was reported by de Jager [8].

Conclusions

Integrated design of systems and their controls requires tools for symbolic modeling and manipulation. Our approach involves the integration of symbolic and numerical computing. Successful modeling of complex vehicle dynamics and design of adaptive tracking controls for a detailed model of a magnetic bearing demonstrate that this methodology can solve realistic problems.

Remark

The Controls package also includes a collection of functions for the design and analysis of linear control systems.

Acknowledgement

Portions of this work were supported in part by the US Army TACOM under Contract No. DAAE07-93-C-R022.

References

[1] O. Akhrif, *Using Computer Algebra in the Design of Nonlinear Control Systems*, M.S. thesis, University of Maryland at College Park, 1987.

[2] V.I. Arnold, V.V. Kozlov, and A.I. Neishtadt, *Mathematical Aspects of Classical and Celestial Mechanics, in Encyclopedia of Mathematical Sciences*, V.I. Arnold, ed., vol. 3, Springer-Verlag, Heidelberg, 1988.

[3] G.L. Blankenship and N. Sreenath, "Symbolic and Numerical Modeling and Design of Nonlinear Control Systems for Multibody Systems," *Proc. 1991 Vehicle Tech. Conf., Simulation and Ground Vehicle Robotics*, U.S. Army TACOM, June 1991, pp. 123-135.

[4] G.L. Blankenship, R. Ghanadan, and V. Polyakov, "Nonlinear Adaptive Control of Active Vehicle Suspension," *Proc. 1993 ACC*, San Francisco, June 1993.

[5] G.L. Blankenship, R. Ghanadan, H.G. Kwatny, and V. Polyakov, "Modeling and Design Tools for Control of Multibody Systems," *Proc. ASME Winter Annual Meeting: High Performance Computing in Vehicle Systems*, New Orleans, 1993.

[6] S. Cetinkunt, and H. Ittoop, "Computer-Automated Symbolic Modeling of Dynamics of Robotic Manipulators with Flexible Links," *IEEE Trans. Robotics and Automation*, 8(1992), pp. 94-105.

[7] J.P. Chancelier, C. Gomez, J.P. Quadrat, and A. Sulem, "Pandore," (*Proc. NATO Advanced Study Institute on CAD of Control Systems*, Il Ciocco, September 1987), in *Advanced Computing Concepts and Techniques in Control Engineering*, M. Denham and A. Laub, eds., Springer-Verlag, New York, pp. 81-125, 1988.

[8] B. de Jager, "The Use of Symbolic Computation in Nonlinear Control: Is It Viable?," *Proc. IEEE Conference on Decision and Control*, San Antonio, TX, 1993, pp. 276- 281.

[9] R. Ghanadan and G. L. Blankenship, "Adaptive Output Tracking of Invertible MIMO Nonlinear Systems," *Proc. of the 26th Conf. on Inf. Sciences and Systems*, Princeton, pp. 767-772, 1992.

[10] R. Ghanadan and G. L. Blankenship, "Adaptive Approximate Tracking and Regulation of Nonlinear Systems," in *Proc. of 32nd IEEE CDC, San Antonio*, Dec. 1993; and *IEEE Trans. on Aut. Cont.*, to appear.

[11] R. Ghanadan, *Adaptive Control of Nonlinear Systems with Applications to Flight Control Systems and Suspension Dynamics*, Ph.D. thesis, Institute for Systems Research, University of Maryland, College Park, 1993.

[12] E.J. Haug, ed., *Computer Aided Analysis and Optimization of Mechanical System Dynamics*. NATO ASI, vol. 59, Springer-Verlag, Berlin, 1984.

[13] J. Hauser, S. Sastry, and P. Kokotovic, "Nonlinear Control Via Approximate Input-Output Linearization: The Ball and Beam Example," *IEEE Trans. Aut. Cont.*, 37(1992), pp. 392-398.

[14] R.M. Hirschorn, "Invertability of Multivariable Nonlinear Systems," *SIAM J. Optim. and Control*, 17(1979), pp. 289-297.

[15] R.L. Huston, *Multibody Dynamics*, Butterworth Heinemann, Boston, 1990.

[16] A. Isidori, *Nonlinear Control Systems*, Springer-Verlag, Berlin, 1989.

[17] A.J. Krener, "Approximate Linearization by State Feedback and Coordinate Changes," *System Control Letters*, 5(1984), pp. 181-185.

[18] H.G. Kwatny and G.L. Blankenship, "Symbolic Construction of Models for Multibody Dynamics," *IEEE Trans. Robotics and Automation*, 10(1994), to appear.

[19] H.G. Kwatny and G.L. Blankenship, "Symbolic Tools for Variable Structure Control System Design: The Zero Dynamics," *Proc. Workshop on Robust Control via Variable Structure and Lyapunov Techniques*, Benevento, Italy, 1994.

[20] H.G. Kwatny and J. Berg, *Variable Structure Regulation of Power Plant Drum Level, in Systems and Control Theory for Power Systems*, Springer-Verlag: New York, 1993.

[21] H.G. Kwatny, "Variable Structure Control of AC Drives," in *Variable Structure Control for Robotics and Aerospace Applications*, K.D. Young, ed., Elsevier: Amsterdam, 1993.

[22] Kwatny, H.G. and H. Kim, "Variable Structure Regulation of Partially Linearizable Dynamics," *Systems & Control Letters*, 15(1990), pp. 67-80.

[23] C. LaVigna, H.G. Kwatny, G.L. Blankenship, *Flexible Multibody Dynamical Analysis System*, Techno-Sciences, Inc., Final Report, U.S. Army TACOM Contract No. DAAE07-93-C-R022, 1993.

[24] M.C. Leu and H. Hemati, "Automated Symbolic Derivation of Dynamic Equations for Robotic Manipulators," *ASME J. Dyn. Syst., Meas. and Control.*, 108(1986), pp. 172-179.

[25] A.M. Mohamed and F.P. Emad, "Conical Magnetic Bearings with Radial and Thrust Control," *IEEE Trans. Aut. Cont.*, 37(1992), pp. 1859-1868.

[26] H. Nijmeijer and A J. van der Shaft, *Nonlinear Dynamical Control Systems*, Springer-Verlag, Berlin, 1990.

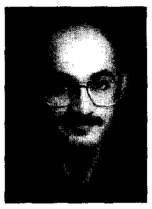
[27] V. Polyakov, R. Ghanadan, and G.L. Blankenship, "Nonlinear Adaptive Control of Conical Magnetic Bearings," 1994, to appear.

[28] W.H. Press, et al., *Numerical Recipes in C: The Art of Scientific Computing*, University Press, New York, 1992.

[29] R. Rutz, and J. Richert, "CAMeL_An Open CACSD Environment," *Proc. IEEE/IFAC Joint Symp. Computer-Aided Control System Design*, Tucson, March 1994, pp. 553-560.

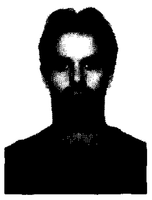
[30] S.N. Singh, "A Modified Algorithm for Invertibility in Nonlinear Systems," *IEEE Trans. Aut. Cont.*, 25(1981), pp. 595-598.

Gilmer L. Blankenship was born in Beckley, WV, on Sept. 11, 1945. He received the S.B., S.M., and Ph.D. degrees from the Massachusetts Institute of Technology, Cambridge, MA, in 1967, 1969, and 1971, respectively. He is a professor and associate chairman in the Department of Electrical Engineering, University of Maryland, College Park. He is a faculty associate with the Institute for Systems Research and a member of the Applied Mathematics Faculty. His research interests include discrete event systems scheduling theory and applications, nonlinear and adaptive control theory, scattering theory and the mechanics of advanced materials, and the applications of AI methods and computer algebra in these areas. Blankenship is a member of the Society for Industrial and Applied Mathematics, the Association of Computing Machinery, and the American Society for Composite Materials. He is a Fellow of the IEEE.

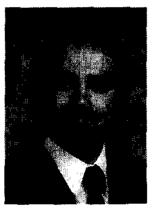


Reza Ghanadan received B.S. degrees in EE and physics, summa cum laude, M.S. and Ph.D. degrees in EE from the University of Maryland, College Park, in 1988, 1990, and 1993, respectively. In August 1993, he joined the University of California, Santa Barbara, as a visiting research engineer. His research interests are in the areas of nonlinear, and adaptive control, with applications to flight control systems, automotive vehicles, and robotics, and development of software tools for control engineering. He is a member of IEEE and Phi Kappa Phi, and a fellow of Tau Beta Pi.

Harry G. Kwatny received the B.S.M.E. degree from Drexel Institute of Technology in 1961, the S.M. in aeronautics and astronautics from the Massachusetts Institute of Technology in 1962, and the Ph.D. (E.E.) from the University of Pennsylvania in 1967. He is currently S. Herbert Raynes Professor of Mechanical Engineering at Drexel University, Philadelphia. His main research interests include the analysis and control of parameter dependent nonlinear dynamics and the use of combined numeric-symbolic computation for addressing these problems. He has contributed to disturbance accommodating control theory and variable structure control. He has a strong interest in physical system modeling and his application interests include the analysis and control of power plants and power systems, flexible spacecraft and space robotics, flight control, ground vehicle dynamics, and structural acoustics control.



Chris LaVigna received the B.S.M.E. and M.S.M.E. degrees from the University of Maryland in 1985 and 1993, respectively. He is currently a senior design engineer at Techno-Sciences Inc. in Lanham, MD. His main research interests include the use of symbolic computing for the design and analysis of control systems for nonlinear systems and the development of simulation tools for validation of advanced control strategies. He also has a strong interest in developing engineering software to support linear and nonlinear control design applications.



Vadim Polyakov was born in Kharkov, USSR, in 1967. He received a B.A. in Slavic literature and russian area studies from Princeton University in 1990 and a B.S. in electrical engineering from the University of Maryland, College Park, in 1992. Currently, he is a graduate student in the Department of Electrical Engineering and a Graduate Fellow at the Institute for Systems Research at University of Maryland, College Park.