MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Memo 955 *May 1987*

The Dynamicist's Workbench: I Automatic Preparation of Numerical Experiments

Harold Abelson and Gerald Jay Sussman

Abstract

The dynamicist's workbench is a system for automating some of the work of experimental dynamics. We describe a portion of our system that deals with the setting up and execution of numerical simulations. This part of the workbench includes a spectrum of computational tools - numerical methods, symbolic algebra, and semantic constraints. These tools are designed so that combined methods, tailored to particular problems, can be constructed on the fly.

Keywords: Dynamics, symbolic computation, numerical methods.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the Laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-86-K-0180.

The dynamicist's workbench is part of a larger project at M.I.T. to investigate the use of combined numerical and symbolic methods in scientific and engineering computing.

The Dynamicist's Workbench: I
Automatic Preparation of Numerical Experiments

Harold Abelson and Gerald Jay Sussman

Abstract

The dynamicist's workbench is a system for automating some of the work of experimental dynamics. We describe a portion of our system that deals with the setting up and execution of numerical simulations. This part of the workbench includes a spectrum of computational tools—numerical methods, symbolic algebra, and semantic constraints. These tools are designed so that combined methods, tailored to particular problems, can be constructed on the fly.

Before computers, exploring the behavior of a dynamical system was tedious, often requiring great skill to make the computations tractable. In 1801 it took a Gauss to compute the orbits of Ceres and Pallas; with a computer, anyone can do it. But experimental dynamics is still tedious: An investigator repeatedly selects interesting values for parameters and initial conditions, sets up numerical computations, decides when each run is completed, and classifies the results, decomposing the system's parameter space into regions of qualitatively different behavior. Even with powerful numerical computers, experimental dynamics typically requires significant human effort to set up simulations and relies upon human judgment to choose parameter values that are "interesting", to determine when each simulation run is "complete", and to decide when two behaviors are "qualitatively different". Much of this work, however, can be automated.

Our goal is to produce a *dynamicist's workbench*—a computer system that could, in principle, write many of the published papers that describe the behaviors of particular dynamical systems. Such a system must be able to set up and evolve numerical simulations. It must exploit algebraic and geometric constraints to determine when a simulation will produce no new interesting behavior, to classify trajectories and to recognize the bifurcations of critical sets. In this paper we describe a portion of our system that deals with the setting up and execution of numerical simulations.[1] This part of the workbench includes a spectrum of computational tools—numerical methods, symbolic algebra, and semantic constraints (such as dimensions). These tools are designed so that combined methods, tailored to particular problems, can be constructed on the fly. One can use symbolic algebra to automatically generate numerical procedures, one can use domain-specific constraints to guide algebraic derivations and to avoid complexity, and one can use numerical methods to identify and verify qualitative properties of systems.

We illustrate these ideas in the context of a few dynamical systems initially formulated as electrical networks. Section 1 presents a language for describing electrical networks. From these descriptions, the workbench generates the algebraic constraints and other information needed to support analysis. Section 2 illustrates how the workbench evolves the state of a dynamical system by automatically compiling a procedure to compute the system derivative, and combining this with an appropriate numerical integrator composed from primitive integrators and one of a number of strategies for adaptive step-size control. These system-derivative procedures generated by the workbench may incorporate iteration schemes when the state-variable derivatives cannot be expressed in closed form. Section 3 describes the automatic compilation of procedures that compute the frequency response of linear systems. This requires substantial symbolic manipulation, which is made tractable by using semantic markers to guide the algebra. In section 4 we demonstrate how one can explore the complex dynamics of the driven van der Pol oscillator. Here the workbench automatically compiles numerical procedures for finding periodic orbits and for tracking them as the system parameters vary.

1 Algebraic environments

Information about a dynamical system to be analyzed using the workbench is organized in an *algebraic environment*, which is a structure that maintains algebraic constraints among variables. A variable may be annotated with

[1]Ken Yip's Ph.D. thesis, in progress, shows how a program can classify trajectories and bifurcations and use the results to choose promising values for parameters and initial conditions.

semantic markers describing its role in a dynamical system. For example, some variables may be parameters of the system, while others may be state variables. One variable may name a quantity with dimensions of length while another may have dimensions of capacitance. The workbench performs symbolic algebra on expressions in these variables, using the semantic markers to guide the algebraic manipulations. Algebraic environments can be constructed from explicitly specified constraints and annotations. Alternatively, one can employ a special-purpose language tailored for constructing algebraic environments from descriptions of dynamical systems such as electrical circuits or signal-flow graphs.

The examples in this paper are drawn from electrical circuit theory. They are initially formulated in terms of a network-description language. The network language contains a few predefined parts corresponding to the simplest electrical elements: *resistor*, *capacitor*, *inductor*, *voltage-source*, and *current-source*. There are also two primitives in terms of which all these elements can be defined: the *branch* and the *constraint*. A branch defines an arbitrary two-terminal device with its associated quantities, and a constraint establishes a relationship among quantities. Using branches and constraints, one can also define elements such as op-amps and nonlinear resistors.

In the network language, compound networks are constructed by connecting together (primitive or non-primitive) parts. Any compound network, once defined, can be used as a part in constructing a still more complex network. The expression `define-network` is used to name newly-constructed networks. Here is a definition of the Twin-T circuit shown in figure 1 consisting of three resistors, three capacitors, and a voltage source:

```
(define-network twin-t () (n1 n2 n3 n4)
  (parts (s voltage-source (n+ n3) (n- gnd))
         (r1 resistor (n+ n3) (n- n2))
         (r2 resistor (n+ n2) (n- n4))
         (r3 resistor (n+ n1) (n- gnd))
         (c1 capacitor (n+ n3) (n- n1))
         (c2 capacitor (n+ n1) (n- n4))
         (c3 capacitor (n+ n2) (n- gnd))))
```

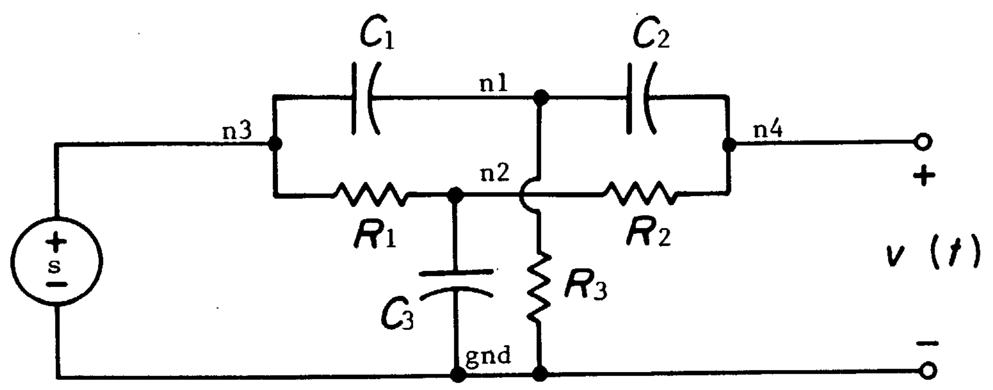


Figure 1: The Twin-T network is a third-order linear system that is often used as a notch filter in audio applications.

The general form of a network definition is

```
(define-network type arguments nodes
  (parts part1
         part2
         ...
         partn))
```

`Type` is the name for the new type of network being defined. `Arguments` and `nodes` are each a list of information that may be specified when a network of this type is used (see below). Each of the `part` entries consists of a name for the part, a specification of the type of the part, and specifications for the arguments and nodes of the part.

The nodes in a network definition describe the topology of how the parts are connected. This is accomplished by showing correspondence between the names of the nodes of the parts and the names of the nodes of the network being defined. In the case of the `twin-t` definition, there are four nodes: `n1`, `n2`, `n3`, and `n4`. The definition of a capacitor as a primitive element includes two nodes, `n+` and `n-`. The `twin-t` subpart specification

```
(c1 capacitor (n+ n3) (n- gnd))
```

stipulates that `c1` is a capacitor whose `n+` node is identified with the circuit node `n3` and whose `n-` node is identified with the node `gnd`, which is an additional external node defined for all circuits. Observe that the names of nodes defined for a part are local to the part. Thus, the capacitors and resistors each refer to their own two nodes as `n+` and `n-`, and in each case these are identified with different nodes of the Twin-T network. In a similar way, if we were to define a network that used a `twin-t` as a part, the appropriate part specification would indicate how the nodes `n1`–`n4` are to be identified with nodes defined in the larger network. In general, each pair of names in a part specification matches a name defined locally for the part with an object from the context in which the part is embedded.

Declarations and constraints

A network definition can be interpreted to generate algebraic constraints and semantic marker declarations for construction of an instance of the type described by the definition. We can construct an algebraic environment containing an instance of the prototype Twin-T network as follows:

```
(define twin-t-inst (make-algebraic-instance twin-t))
```

Some algebraic variables represent device parameters (such as the resistance of a resistor). Others represent circuit variables (such as node potentials, branch voltages, and terminal currents). Some circuit variables (such as the voltage across a capacitor) may be state variables of the system. Among the declarations created by executing the `twin-t` description are

```
(r.r2 parameter resistance)
(c.c1 parameter capacitance)
((v.r3 t) circuit-variable voltage)
((v.c2 t) state-variable voltage)
```

Thus, `r.r2` is a parameter with the dimensions of a resistance, and `c.c1` is a parameter with the dimensions of a capacitance. `V.r3` is a function of time, is a circuit-variable, and has the dimensions of a voltage. `V.c2` is a function of time, is a state-variable of the capacitor[2] and has the dimensions of a voltage.

Here are some of the constraints that are developed by interpreting the `twin-t` definition:

[2]`V.c2` is a state-variable of the capacitor considered in isolation, but it is not necessarily a state-variable of the network in which the capacitor is embedded.

```
(fact140 (= (+ (- 0 (i.r1 t)) (i.r2 t) (i.c3 t)) 0))

(fact155 (= (v.c3 t) (- (e.n2 t) (e.gnd t))))

(fact156 (= (i.c3 t) (* c.c3 (rate (v.c3 t) ($ t)))))
```

`Fact140` is Kirchhoff's Current Law at node `n2`. `Fact155` stipulates that the voltage across capacitor `c3` is the difference of the potentials at the nodes `n2` and `gnd` (Kirchhoff's Voltage Law). `Fact156` is the constituent relation for the capacitor `c3`, asserting that the current is the product of the capacitance and the time-derivative of the voltage.[3]

Declarations and constraints are derived ultimately from descriptions of primitive network elements, such as the following description of a capacitor:

```
(define-network capacitor
  ((c parameter capacitance)
   (v state-variable voltage)
   (i circuit-variable current))
  (n+ n-)
  (primitive-element
   (constraints '(= ,v (- ,(potential n+) ,(potential n-)))
                '(= ,i (* ,c ,(rate v))))
   (current-from-node '(,n+ ,i) '(,n- (- 0 ,i)))))
```

The prototype capacitor has a parameter C with the dimensions of a capacitance, a state variable v with the dimensions of a voltage, and a circuit-variable i with the dimensions of a current. There are two constraints—Kirchhoff's Voltage Law and the constituent relation $i = Cdv/dt$. The capacitor also answers the question "What is the current entering you from node `m`?". When parts are connected to form networks, Kirchhoff's Current Law is enforced by asserting, at each node, the constraint that the sum of the currents entering each part from that node is zero.

The circuit language is only one example of a language for translating domain-specific descriptions into algebraic constraints. Another example (also provided in the workbench although not discussed here) is a language for describing the block-diagram systems used in signal processing. One could

[3]The dollar-sign syntax in `fact156` indicates that the derivative (`rate`) is to be taken with respect to `t`.

also imagine a similar language for describing mechanisms constructed from rods, cams, and gears. In each case, in moving from the problem domain to the algebra, one needs to capture not only the algebraic equations but the semantic markers as well. We shall see below how this semantic information is used.

2 Evolving time-domain behavior

Given a dynamical system specified in terms of an algebraic environment, the workbench automatically generates procedures that support the simulation of the system. Some of these procedures are numerical. Others are higher-order "generators" that will be specialized when the simulation is run. These procedures are automatically combined with input and graphical output routines to generate simulation programs. If we instruct the workbench to prepare a time-domain simulation, we will be asked to specify an initial state and values for the parameters. The workbench will use these values to evolve the corresponding time behavior, and can report the values of any variable contained in the algebraic environment, or of any algebraic expression in these variables. Figure 2 shows three graphs produced by the workbench for the `twin-t` network described above.

2.1 Generating a system derivative

Given an algebraic environment containing constraints and semantic markers, the workbench's first step in producing a time-domain simulation is to express the derivatives of the state variables in terms of the parameters and the state variables, performing algebraic manipulation to eliminate intermediate variables as necessary.[4] In any particular circuit, the state variables of the individual parts, such as the voltages of capacitors, may be dependent. The workbench recognizes such dependencies and automatically chooses an independent set of state variables.

If the system's state vector is $\mathbf{y}$ then the result of this manipulation is a

[4]In nonlinear systems, solving explicitly for the derivatives may be beyond the capabilities of the algebraic manipulator. Section 2.4 describes how the workbench constructs iterative methods for dealing with such situations.

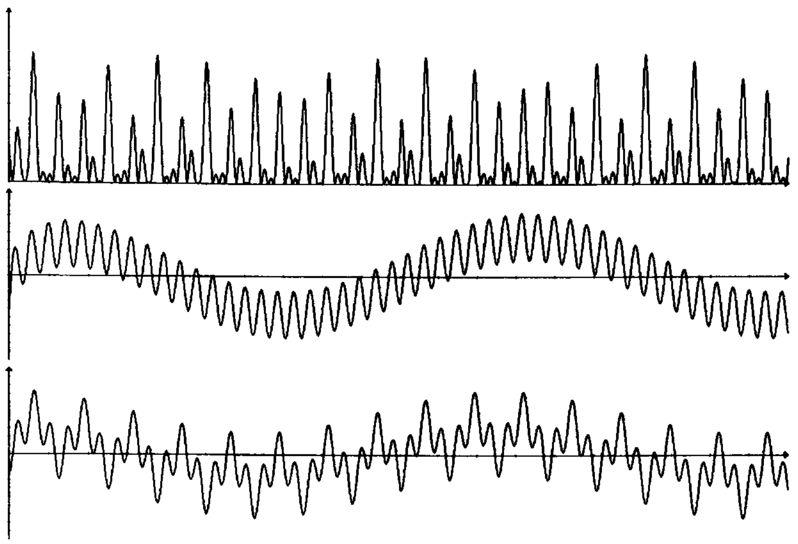


Figure 2: This is the time-domain behavior of the Twin-T network as evolved by the workbench. The parameters are $C_1 = C_2 = 0.1$, $C_3 = 0.2$, $R_1 = R_2 = 1$, $R_2 = 0.5$, and the source voltage is $\cos t - \cos 10t - \cos 30t$. The initial state (at $t = 0$) is **v.c1** $=$ **v.c2** $=$ **v.c3** $= 0$. The horizontal scale is $[0, 10]$ seconds. The bottom trace shows the source voltage—a superposition of three sinusoids. The vertical scale is $[-4, 4]$ volts. The middle graph show the potential at node **n4**—with this choice of parameters, the Twin-T network is behaving as a notch filter, suppressing the middle-frequency component of the input. The vertical scale here is $[-2, 2]$ volts. The top graph shows the power dissipated in resistor **r1**. The vertical scale here is $[0, 5]$ watts.

first-order system of the form:

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(t, \mathbf{y})$$

The resulting algebraic expressions for the components of the system derivative, $\mathbf{f}$, are compiled to form a *system-derivative generator* procedure. The generator takes as arguments numerical values for the system parameters and produces a *system-derivative* procedure, which takes a system state vector as argument and produces a differential state (a vector that when multiplied by an increment of time is an increment of state). The system-derivative procedure will be passed to an *integration driver* that returns a procedure, which given an initial state, evolves the system numerically. Other expressions are also compiled into procedures that compute values from a system state vector.

To eliminate variables, the solver repeatedly chooses an unused equation, picks a variable from the class of variables to be eliminated, and solves the equation for that variable. The resulting value is bound to the variable in the algebraic environment, so that subsequent evaluations of expressions with respect to that algebraic environment will return values that have no instances of the eliminated variables. Backsubstitutions are thus automatically subsumed by the evaluation process. Although for particular classes of equations, such as linear systems, there are vastly more efficient algorithms for producing the system derivative, we have chosen this elimination strategy because it can easily tolerate a few nonlinear relations, and it degrades gracefully when it cannot make progress.

This elimination strategy is simple in outline, but in practice it is important to carefully order the sequence of eliminations to minimize the size of intermediate expressions. Treating operator and function applications as variables also requires care. Consider the two equations $dx/dt = dy/dt$ and $y = x + z$ where we regard dx/dt and dy/dt as variables. If we first eliminate dx/dt (reducing it to dy/dt), then eliminate y (reducing it to $x + z$), and then attempt to evaluate dx/dt, we will not have performed a complete elimination, because dx/dt will reduce to $dx/dt + dz/dt$.

Example: The system derivative for the Twin-T network

For the Twin-T network the workbench produces the following expressions for the derivatives of the three state variables:

```
(rate (v.c3 t) ($ t)) = (/ (+ (* r.r1 (strength.s t))
                              (* -1 r.r1 (v.c1 t))
                              (* -1 r.r1 (v.c2 t))
                              (* -1 r.r1 (v.c3 t))
                              (* r.r2 (strength.s t))
                              (* -1 r.r2 (v.c3 t)))
                           (* c.c3 r.r1 r.r2))

(rate (v.c2 t) ($ t)) = (/ (+ (strength.s t)
                              (* -1 (v.c1 t))
                              (* -1 (v.c2 t))
                              (* -1 (v.c3 t)))
                           (* c.c2 r.r2))

(rate (v.c1 t) ($ t)) = (/ (+ (* r.r2 (strength.s t))
                              (* -1 r.r2 (v.c1 t))
                              (* r.r3 (strength.s t))
                              (* -1 r.r3 (v.c1 t))
                              (* -1 r.r3 (v.c2 t))
                              (* -1 r.r3 (v.c3 t)))
                           (* c.c1 r.r2 r.r3))
```

These expressions are compiled to form the system-derivative generator procedure shown in figure 3. The generator takes as arguments the `twin-t` network's seven declared parameters—three capacitances, three resistances, and the strength of the source. (The strength of the source is a procedure that computes a function of time). The result returned by the generator is a procedure whose single argument is a state vector with four components—time, and the voltages across the three capacitors. Notice that the workbench's expression-compiler has performed a bit of common-subexpression removal to make derivative computation more efficient and to prevent multiple evaluation of the `strength.s` procedure.

2.2 Compiling auxiliary expressions

In addition to generating algebraic expressions for the system derivatives, the workbench includes a general-purpose *algebraic evaluator* that can be called to evaluate algebraic expressions relative to a given algebraic environment. For instance, we can ask for the power (product of current and voltage) in the Twin-T network's resistor `r1`:

```
(lambda (c.c3 c.c2 c.c1 r.r3 r.r2 r.r1 strength.s)
  (lambda (*state*)
    (let ((t (vector-ref *state* 0))
          (v.c3 (vector-ref *state* 1))
          (v.c2 (vector-ref *state* 2))
          (v.c1 (vector-ref *state* 3)))
      (let ((g2 (strength.s t)))
        (let ((g6 (* g2 r.r2)) (g4 (* -1 v.c1))
              (g3 (* -1 v.c3)) (g5 (* -1 v.c2))
              (g1 (* -1 r.r1)))
          (vector 1
                  (/ (+ g6
                        (* g1 v.c1)
                        (* g1 v.c2)
                        (* g1 v.c3)
                        (* g2 r.r1)
                        (* g3 r.r2))
                     (* c.c3 r.r1 r.r2))
                  (/ (+ g2 g3 g4 g5)
                     (* c.c2 r.r2))
                  (/ (+ g6
                        (* g2 r.r3)
                        (* g3 r.r3)
                        (* g4 r.r2)
                        (* g4 r.r3)
                        (* g5 r.r3))
                     (* c.c1 r.r2 r.r3)))))))
```

Figure 3: The system-derivative generator procedure compiled for the Twin-T network takes as arguments a set of parameters and returns as its value a procedure that takes a state vector and returns a differential state. Each of the four components of the returned differential state is the time derivative of the corresponding component of the state vector.

```
==> (algebra-value '(* (v.r1 t) (i.r1 t)) (twin-t-inst 'time-domain))

(/ (+ (* (strength.s t) (strength.s t))
      (* -2 (strength.s t) v.c3)
      (* v.c3 v.c3))
   r.r1)
```

We can verify (as Tellegen's theorem implies) that the sum of the powers into all the elements in the network is zero:

```
==> (algebra-value '(+ (* (v.r1 t) (i.r1 t))
                       (* (v.r2 t) (i.r2 t))
                       (* (v.r3 t) (i.r3 t))
                       (* (v.c1 t) (i.c1 t))
                       (* (v.c2 t) (i.c2 t))
                       (* (v.c3 t) (i.c3 t))
                       (* (v.s t) (i.s t)))
                   (twin-t-inst 'time-domain))
0
```

We can also instruct the workbench to compile numerical procedures to compute various expressions, such as the powers into each of the resistors, as functions of the state and the parameters:

```
==> (compile-time-expressions '((* (v.r1 t) (i.r1 t))
                                (* (v.r2 t) (i.r2 t))
                                (* (v.r3 t) (i.r3 t)))
                              (twin-t-inst 'time-domain))
```

The procedure compiled for these expressions is shown in figure 4. The top graph in figure 2 was produced by plotting the values of each first component in the sequence of triples generated using this procedure.

2.3 Generating methods of integration

To evolve a dynamical system, a system derivative is combined with an *integration driver* to produce a procedure which, when called with an initial state, evolves the state numerically. Here is a typical integration driver:

```
(lambda (c.c3 c.c2 c.c1 r.r3 r.r2 r.r1 strength.s)
  (lambda (*state*)
    (let ((t (vector-ref *state* 0))
          (v.c3 (vector-ref *state* 1))
          (v.c2 (vector-ref *state* 2))
          (v.c1 (vector-ref *state* 3)))
      (let ((g7 (strength.s t)))
        (let ((g9 (* -2 g7))
              (g12 (* g7 g7)))
          (let ((g11 (+ (* g9 v.c1) (* v.c1 v.c1)))
                (g8 (+ g12 (* g9 v.c3) (* v.c3 v.c3)))
                (g10 (* 2 v.c1)))
            (list (/ g8 r.r1)
                  (/ (+ g11
                        g8
                        (* 2 v.c2 v.c3)
                        (* g10 v.c2)
                        (* g10 v.c3)
                        (* g9 v.c2)
                        (* v.c2 v.c2))
                     r.r2)
                  (/ (+ g11 g12)
                     r.r3))))))))
```

Figure 4: This is the procedure generator compiled to compute the powers dissipated by each of the three resistors in the Twin-T network. Note the extensive common-subexpression removal performed by the workbench here.

```
(define (system-integrator system-derivative max-h method)
  (let ((integrator (method system-derivative)))
    ;; integrator : state-and-step ---> state-and-step
    (define (next state-and-step)
      (output (state-part state-and-step))
      (let ((new-state-and-step (integrator state-and-step)))
        (next (make-state-and-step
               (state-part new-state-and-step)
               (min (step-part new-state-and-step) max-h)))))
    next))
```

`System-integrator` takes as arguments a system derivative, a maximum step-size `max-h`, and a general `method` of integration. The `method` is applied to the system derivative, producing an `integrator` which, given a data structure that contains a state and a step-size, integrates for one step. In order to admit integrators with adaptive step-size control, `integrator` is structured to return not only the next state, but also a predicted next step-size, wrapped up in a data structure constructed by `make-state-and-step`. The result produced by the integration driver is a procedure `next` which, given an initial state and an initial step-size, evolves the sequence of states, passing each state to an `output` procedure (which, for example, produces graphical output). At each time-step, the integration is performed using the step-size predicted during the previous step, provided that this is less than the specified `max-h`.[5]

The workbench includes various methods of integration that can be combined with integration drivers. Some of these methods are themselves automatically generated by operating upon simple integrators with *integrator transformers.*

One integrator transformer incorporates a general strategy described in [6], for transforming a non-adaptive integrator into an integrator with adaptive step-size control: Given a step-size h, perform one integration step of size h and compare the result with that obtained by performing two steps of size $h/2$. If the difference between the two results is smaller than a prescribed error tolerance, then the integration step succeeds, and we can attempt to

[5]`System-integrator` is only one of a number of possible integration drivers. The one actually used in the workbench produces a *stream* of states, so that integration steps are performed on a "demand-driven" basis. (See [1] for information on stream processing.)

use a larger value of h for the next integration step. If the difference is larger than the error tolerance, we choose a smaller value of h and try the integration step again.

More precisely, let `2halfsteps` be the (vector) result of taking two steps of size $h/2$, and let `fullstep` be the result of taking one step of size h. Then

$$E = \max_i \left| \frac{\texttt{2halfsteps}_i - \texttt{fullstep}_i}{\texttt{2halfsteps}_i} \right|$$

is an estimate of the relative error. Let $\texttt{err} = \frac{E}{\texttt{tolerance}}$ be the ratio of E to a prescribed error tolerance. We choose the new step-size to be

$$\texttt{newh} = h \times \texttt{err}^{\frac{-1}{n+1}} \times \texttt{safety}$$

where the underlying method of integration has order n, and where `safety` is a safety factor slightly smaller than 1. If the integration step fails ($\texttt{err} > 1$) we retry the step with `newh`. If the integration step succeeds ($\texttt{err} < 1$) we use `newh` for the next step. We can also make an order-$(n + 1)$ correction to `2halfsteps`, computing the new state components as

$$\texttt{2halfsteps}_i + \frac{\texttt{2halfsteps}_i - \texttt{fullstep}_i}{2^n - 1}$$

See [6] for more details.

The `make-adaptive` procedure, which implements this strategy, is shown in figure 5. The arguments to `make-adaptive` are a `stepper` that performs one step of a non-adaptive method of integration, together with the order of the method. `Make-adaptive` returns the corresponding adaptive integrator, which takes a system derivative f as argument and returns a procedure which, given a state and stepsize, returns the next state and a new stepsize.[6]

The `stepper` to be transformed by `make-adaptive` is a procedure that takes as arguments a system derivative f, a state y, a stepsize h, and the

[6]Some details of `make-adaptive`: `Zero-stop` is a small number that is used to avoid possible division by zero. `Scale-vector` is a procedure which, given a number, returns a procedure that scales vectors by that number. `Elementwise` takes as argument a procedure of n arguments. It returns the procedure of n vectors that applies the original procedure to the corresponding elements of the vectors and produces the vector of results.

```
(define (make-adaptive stepper order)
  (let ((error-scale (/ -1 (+ order 1)))
        (scale-diff (scale-vector (/ 1 (- (expt 2 order) 1)))))
    (lambda (f)
      (lambda (state h-init)
        (let ((der-state (f state)))
          (let reduce-h-loop ((h h-init))
            (let* ((h/2 (/ h 2))
                   (fullstep (stepper f der-state state h))
                   (halfstep (stepper f der-state state h/2))
                   (2halfsteps (stepper f (f halfstep) halfstep h/2))
                   (diff (sub-vectors 2halfsteps fullstep))
                   (err (/ (maxnorm
                            ((elementwise (lambda (y d)
                                            (/ d (+ zero-stop (abs y)))))
                             2halfsteps
                             diff))
                           tolerance))
                   (newh (* safety h (expt err error-scale))))
              (if (> err 1)
                  (reduce-h-loop newh)
                  (make-state-and-step
                   (add-vectors 2halfsteps (scale-diff diff))
                   newh)))))))))
```

Figure 5: This is an integrator transformer procedure, which transforms a non-adaptive integration `stepper` into an integration method with adaptive step-size control.

value dy/dt of f at y.[7] Here is a simple first-order backward Euler predictor-corrector stepper. Given a y and f, the stepper first computes a predicted next state $y_p = y + hf(y)$ and then estimates a corrected next state as $y + hf(y_p)$.

```
(define (backward-euler f dy/dt y h)
  (let* ((h* (scale-vector h))
         (yp (add-vectors y (h* dy/dt))))
    (add-vectors y (h* (f yp)))))
```

The corresponding adaptive integrator is constructed by

```
(define adaptive-backward-euler (make-adaptive backward-euler 1))
```

Here is a fourth-order Runge-Kutta stepper

```
(define 2* (scale-vector 2))
(define 1/2* (scale-vector (/ 1 2)))
(define 1/6* (scale-vector (/ 1 6)))

(define (runge-kutta-4 f dy/dt y h)
  (let* ((h* (scale-vector h))
         (k0 (h* dy/dt))
         (k1 (h* (f (add-vectors y (1/2* k0)))))
         (k2 (h* (f (add-vectors y (1/2* k1)))))
         (k3 (h* (f (add-vectors y k2)))))
    (add-vectors y
                 (1/6* (add-vectors (add-vectors k0 (2* k1))
                                    (add-vectors (2* k2) k3))))))
```

The corresponding adaptive integrator is

```
(define adaptive-runge-kutta-4 (make-adaptive runge-kutta-4 4))
```

Other transformation strategies lead to other sophisticated integrators. For example, the Bulirsch-Stoer integrator can be constructed by transforming a simple modified-midpoint stepper by means of a Richardson extrapolation generator [6].

[7]One could easily arrange for the stepper itself to compute dy/dt. The reason for passing dy/dt as an argument is to avoid computing it twice in each adaptive integration step—once when evaluating `fullstep` and once when evaluating `halfstep`.

2.4 Generating iteration schemes

In a nonlinear system, one can rarely solve algebraically for the state-variable derivatives as elementary functions of the state variables to produce an explicit system derivative of the form $\mathbf{x}' = \mathbf{F}(\mathbf{x})$. Instead, one encounters a system of nonlinear equations $\mathbf{E}(\mathbf{x}, \mathbf{x}') = \mathbf{0}$ where $\mathbf{x}$ is the vector of state variables, $\mathbf{x}'$ is the vector of corresponding derivatives, and $\mathbf{E}$ is a vector-valued function (one component for each scalar equation). Such systems of implicit differential equations can be attacked with iterative numerical schemes. In the Newton-Raphson scheme, for instance, one solves a system of nonlinear equations $\mathbf{G}(\mathbf{z}) = \mathbf{0}$ by choosing an initial guess $\mathbf{z}^{(0)}$ for the solution and iterating to approximate a fixed point of the transformation

$$\mathbf{z} \leftarrow \mathbf{z} - [\mathbf{DG}(\mathbf{z})]^{-1}\mathbf{G}(\mathbf{z})$$

where $\mathbf{DG}$ is the Jacobian derivative (matrix) of $\mathbf{G}$. This process can be carried out purely numerically, but it is greatly advantageous to use symbolic algebra to develop an explicit expression for $\mathbf{DG}$ and its inverse, because this avoids the need, at each iteration step, to numerically approximate the derivatives $\partial G_i / \partial z_j$ comprising the components of $\mathbf{DG}$.

The workbench uses this mixed symbolic-numerical method. In general, when attempting to compute the system derivative as outlined in section 2.1 the equation solver will fail to eliminate all non-state variables from the equations, and be left with a system of the form

$$\begin{aligned} \mathbf{G}(\mathbf{x}, \mathbf{u}) &= \mathbf{0} \\ \mathbf{x}' &= \tilde{\mathbf{F}}(\mathbf{x}, \mathbf{u}) \end{aligned}$$

Here $\mathbf{x}$ is the vector of state variables and $\mathbf{u}$ is a vector of additional "unknown" variables that could not be eliminated, leaving the implicit equations $\mathbf{G} = \mathbf{0}$. (Those derivatives of state variables that could not be eliminated in terms of state variables are included in the unknowns.) The easy case, in which all non-state variables are eliminated, corresponds to $\mathbf{u}$ being null. The workbench uses symbolic differentiation to derive expressions for the components of $\mathbf{DG}$, which it uses in turn to derive a symbolic expression for the Newton-Raphson transformation

$$\mathbf{u} \leftarrow \mathbf{u} - [\mathbf{DG}(\mathbf{x}, \mathbf{u})]^{-1}\mathbf{G}(\mathbf{x}, \mathbf{u})$$

The workbench also derives symbolic expressions $\mathbf{u}' = \mathbf{H}(\mathbf{x}, \mathbf{u})$ for the derivatives $\mathbf{u}'$. This is accomplished by differentiating the equation $\mathbf{G}(\mathbf{x}, \mathbf{u}) = \mathbf{0}$ to obtain

$$\mathbf{DG_x} \cdot \mathbf{x}' + \mathbf{DG_u} \cdot \mathbf{u}' = \mathbf{0}$$

solving this for $\mathbf{u}'$, and eliminating the $\mathbf{x}'$ in terms of $\mathbf{x}$ and $\mathbf{u}$.

The actual system derivative computation proceeds as follows: The "system state" to be evolved consists of the state variables $\mathbf{x}$ augmented by the variables $\mathbf{u}$. Given values for state variables and guesses $\mathbf{u}^{(0)}$ for the unknowns, Newton-Raphson iteration produces values $\mathbf{u}$ that satisfy $\mathbf{G}(\mathbf{x}, \mathbf{u}) = 0$. The equations $\mathbf{x}' = \tilde{\mathbf{F}}(\mathbf{x}, \mathbf{u})$ and $\mathbf{u}' = \mathbf{H}(\mathbf{x}, \mathbf{u})$ now provide the required $\mathbf{x}'$ and $\mathbf{u}'$. Observe that each integration step evolves not only an updated $\mathbf{x}$, but also an updated $\mathbf{u}$ to be used as the initial guess $\mathbf{u}^{(0)}$ to begin the Newton-Raphson iteration at the next time-step. Usually, the integrator itself will produce a sufficiently good value $\mathbf{u}^{(0)} \approx \mathbf{u}$ that the Newton-Raphson correction will be iterated only once, if at all, at any given time-step.

The workbench compiles a system derivative generator procedure that incorporates the symbolically-derived expressions for $\tilde{\mathbf{F}}$, $\mathbf{H}$, and the Newton-Raphson transformation. The system-derivative generator takes the network parameters as arguments and returns a system-derivative procedure that takes an augmented state as argument and produces the derivatives of the augmented state variables. Packaging things this way provides an important modularity—to evolve the system dynamics, the workbench can pass the system derivative to any general-purpose integration routine. The same integrators are used with the explicit system derivatives generated as in section 2.1 and with the implicit system derivatives that incorporate iterative schemes.

Example: A circuit with cube-law resistors

To illustrate the above strategy, consider the nonlinear RLC circuit shown in figure 6, containing a voltage source, a capacitor, an inductor, and two nonlinear resistors. The resistors are each cube-law resistors with *v*-*i* characteristic $v.a = i.b + (i.b)^3$ where a and b are parameters that scale voltage and current:[8]

[8]The primitive part employed here, `non-linear-resistor`, is a device with nodes `n+` and `n-`. Its parameters are a voltage `v`, a current `i` and a *v*-*i* characteristic `vic`, which is a procedure applied to `v` and `i` to produce an algebraic constraint.

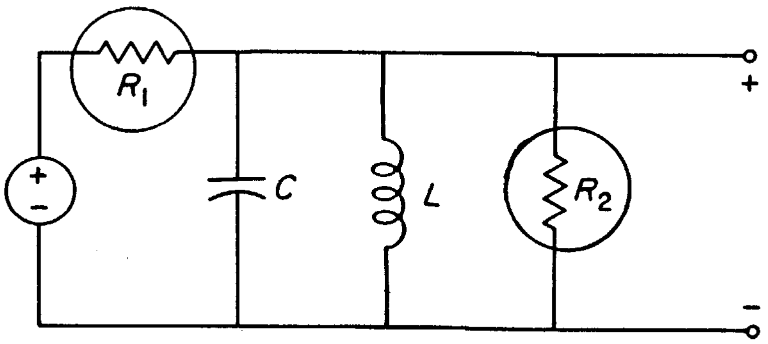


Figure 6: This second-order circuit contains two nonlinear resistors, each with a cubic v-i characteristic. Since the workbench's algebraic manipulator does not solve general cubic equations in closed form, the system derivative generated for this circuit incorporates a symbolically-generated Newton-Raphson interation scheme.

```
(define-network cubic-rlc
  ()
  (n1 n2)
  (parts (s voltage-source (n+ n1) (n- gnd))
         (r1 cube-law-resistor (n+ n1) (n- n2))
         (c capacitor (n+ n2) (n- gnd))
         (l inductor (n+ n2) (n- gnd))
         (r2 cube-law-resistor (n+ n2) (n- gnd))))
```

The equation solver attacks the resulting equations, in which the state variables are the inductor current i_L and the capacitor voltage v_C. The solver succeeds in eliminating di_L/dt, dv_C/dt, and all the non-state circuit variables except for two. These two "unknown" variables are the resistor current i_{R_2} and the source current i_S. The final two equations comprising the system $\mathbf{G}(\mathbf{x}, \mathbf{u}) = \mathbf{0}$ on which the solver cannot make further progress are

$$\begin{aligned} a_{R_2} b_{R_2}^2 i_{R_2} + a_{R_2} i_{R_2}^3 &= b_{R_2}^3 v_C \\ a_{R_1} b_{R_1}^2 i_S + a_{R_1} i_S^3 + b_{R_1}^3 v_S &= b_{R_1}^3 v_C \end{aligned}$$

```
(lambda (b.r2 a.r2 l.l c.c b.r1 a.r1 strength.s)
 (let ((g121 (* b.r1 b.r1)) (g118 (* b.r2 b.r2)))
  (let ((g119 (* b.r1 g121)) (g117 (* a.r2 c.c))
        (g120 (* a.r1 c.c)) (g116 (* b.r2 g118)))
   (lambda (*state*)
    (let ((t (vector-ref *state* 0))
          (i.l (vector-ref *state* 1))
          (v.c (vector-ref *state* 2)))
     (let ((g113 (* -1 i.l)))
      (let ((*values*
             (vector-fixed-point
              (lambda (*unknowns*)
                (let ((i.r2 (vector-ref *unknowns* 0))
                      (i.s (vector-ref *unknowns* 1)))
                  (let ((g123 (* a.r1 i.s i.s))
                        (g122 (* a.r2 i.r2 i.r2)))
                    (vector
                     (/ (+ (* 2 g122 i.r2) (* g116 v.c))
                        (+ (* 3 g122) (* a.r2 g118)))
                     (/ (+ (* -1 g119 (strength.s t))
                           (* 2 g123 i.s)
                           (* g119 v.c))
                        (+ (* 3 g123) (* a.r1 g121)))))))
             (vector (vector-ref *state* 3) (vector-ref *state* 4)))))
        (update-state! *state* 3 *values*)
        (let ((i.r2 (vector-ref *values* 0))
              (i.s (vector-ref *values* 1)))
          (let ((g115 (* -1 i.r2)) (g114 (* -1 i.s)))
            (vector 1
                    (/ v.c l.l)
                    (/ (+ g113 g114 g115) c.c)
                    (/ (+ (* g113 g116) (* g114 g116) (* g115 g116))
                       (+ (* 3 g117 i.r2 i.r2) (* g117 g118)))
                    (/ (+ (* g113 g119) (* g114 g119) (* g115 g119))
                       (+ (* 3 g120 i.s i.s) (* g120 g121))))))))))))
```

Figure 7: The system-derivative generator compiled for the `cubic-rlc` network incorporates an automatically-constucted Newton-Raphson iteration.

Following the method outlined above, the workbench differentiates **G** with respect to $\mathbf{u} = (i_{R_2}, i_S)$ and with respect to $\mathbf{x} = (i_L, v_C)$ to produce the Newton-Raphson transformation and the derivatives $\mathbf{u}'$, and compiles the resulting expressions to form the system-derivative generator shown in figure 7. This procedure takes the system parameters as arguments and returns a procedure that implements the update strategy: Given an augmented `*state*` vector $(t, i_L, v_C, i_{R_2}, i_S)$, extract from this the three state components (t, i_L, v_C) and the two `*unknown*` variables (i_{R_2}, i_S). The unknowns are used to initialize a `vector-fixed-point` operation whose returned `*values*` are the corrected unknowns—a fixed point of the Newton-Raphson transformation. The result returned by the system derivative is a vector whose components are the derivatives of the five variables in the augmented `*state*`, computed as a functions of the given t, i_L, v_C, and of the two corrected unknowns.[9]

3 Frequency-domain analysis

In addition to developing time-domain simulations, the workbench can perform frequency-domain analyses of linear systems. It does this by constructing an algebraic environment that contains the bilateral Laplace transforms of the constraint equations and algebraically solves these equations for the transforms of the circuit variables.

For instance, to analyze the Twin-T network, the workbench must deal with equations such as the transformed `fact156` given above in section 1:

```
(asserting fact287
          (= (transform ($ t s) (i.c3 t))
             (transform ($ t s)
                        (* c.c3 (rate (v.c3 t) ($ t))))))
```

To handle these constraints, the workbench's algebraic manipulator performs such simplifications as

[9]The `update-state!` expression in the system-derivative procedure updates the augmented state to reflect the correction of **u** obtained by Newton-Raphson iteration. This updating has no effect on the computations described in this paper. In the actual workbench integration driver, where we evolve and store a stream of states, the updating ensures that any procedures that later examine the stream of states will see the corrected values.

```
==> (algebra-value
     '(transform ($ t s)
                 (* c.c3 (rate (v.c3 t) ($ t)))))

(* c.c3 s (transform ($ t s) (v.c3 t)))
```

The simplification rules for transforms are expressed in a pattern-match and substitution language. The two rules

```
((transform ($ ?t:symbol? ?s:symbol?) (impulse ($ ?t) ?t0))
 (independent? t0 t)
 `(exp (* -1 ,t0 ,s)))

((transform ($ ?t:symbol? ?s:symbol?) (rate ?exp ($ ?t)))
 no-restrictions
 `(* ,s (transform ($ ,t ,s) ,exp)))
```

illustrate the kinds of transformations that can be specified. These rules encode the transform equations

$$\mathcal{L}\left[\delta(t-t_0)\right] = e^{-t_0 s} \; ;$$

i.e., the transform of a shifted impulse is an exponential, and

$$\mathcal{L}[dx/dt] = s\mathcal{L}[x] \; ;$$

i.e., time-differentiation transforms to multiplication by s. In general, a rule consists of a *pattern* to be matched, an additional predicate that the matching values must satisfy, and a *replacement* to be instantiated with the matching values if the match is successful. In each of the two rules above, the pattern stipulates that the expressions matching `t` and `s` must satisfy the `symbol?` predicate. The first rule also specifies that the impulse offset, `t0`, must be independent of `t`.

Because its simplifier incorporates a general pattern-match language, the workbench can readily be extended to deal with new operators and special functions.[10] The same language is used to implement the simplification rules

[10]This follows Macsyma [5], which provides a pattern matcher that allows users to extend the simplifier.

that handle derivatives in time-domain analysis. Here, for instance, is the rule for differentiating quotients

$$\frac{d(x/y)}{dt} = \frac{y(dx/dt) - x(dy/dt)}{y^2}$$

```
((rate (/ ?x ?y) ($ ?t))
 no-restrictions
 `(/ (- (* ,y (rate ,x ($ ,t)))
        (* ,x (rate ,y ($ ,t))))
     (* ,y ,y)))
```

After solving the frequency-domain equations, the workbench can compute the voltage-transfer ratio of the network as the quotient of two degree-three polynomials in s:

```
==> (algebra-value
     '(/ (- (transform ($ t s) (e.n4 t))
            (transform ($ t s) (e.gnd t)))
         (transform ($ t s) (v.s t))))

(/ (+ (* s s s r.r1 c.c3 r.r3 c.c2 r.r2 c.c1)
      (* s s r.r1 r.r3 c.c2 c.c1)
      (* s s r.r3 c.c2 r.r2 c.c1)
      (* s r.r3 c.c2)
      (* s r.r3 c.c1)
      1)
   (+ (* s s s r.r1 c.c3 r.r3 c.c2 r.r2 c.c1)
      (* s s r.r1 c.c3 r.r3 c.c2)
      (* s s r.r1 c.c3 r.r3 c.c1)
      (* s s r.r1 c.c3 c.c2 r.r2)
      (* s s r.r1 r.r3 c.c2 c.c1)
      (* s s r.r3 c.c2 r.r2 c.c1)
      (* s r.r1 c.c3)
      (* s r.r1 c.c2)
      (* s r.r3 c.c2)
      (* s r.r3 c.c1)
      (* s c.c2 r.r2)
      1))
```

Beginning with such a symbolic analysis, we can explore the effects of adding further constraints. For instance, the Twin-T circuit can be used as a notch filter, if we specialize the resistances and capacitances so that there is a zero in the transfer function at the chosen frequency. We can accomplish this by asserting extra constraints in the algebraic environment

```
(= c.c2 c.c1)
(= c.c3 (* 2 c.c1))
(= r.r2 r.r1)
(= r.r3 (/ r.r1 2))
```

and eliminating the variables c.c2, c.c3, r.r2, and r.r3. In this case, the voltage-transfer ratio reduces to the quotient of degree-two polynomials

$$H(s) = \frac{s^2R_1^2C_1^2 + 1}{s^2R_1^2C_1^2 + 4sR_1C_1 + 1}$$

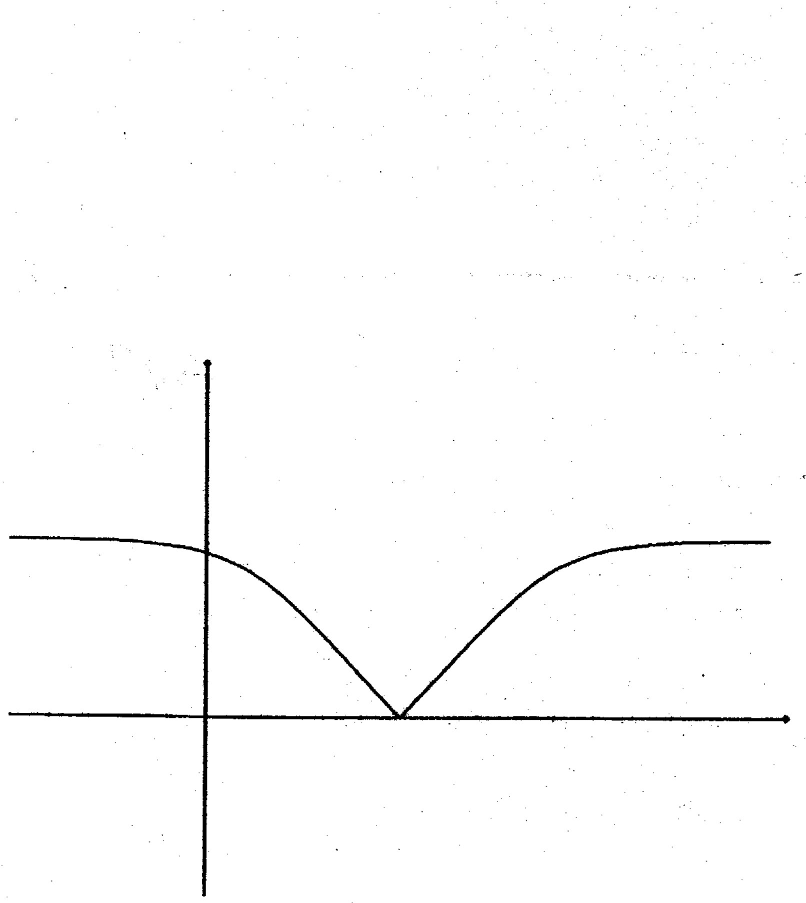
```
==> (algebra-value
      '(/ (- (transform ($ t s) (e.n4 t))
             (transform ($ t s) (e.gnd t)))
          (transform ($ t s) (v.s t))))

(/ (+ (* s s r.r1 r.r1 c.c1 c.c1) 1)
   (+ (* s s r.r1 r.r1 c.c1 c.c1)
      (* 4 s r.r1 c.c1)
      1))
```

As before, the workbench can use these expressions to compile procedures that graph functions of frequency. Figure 8 shows a graph of the magnitude of $H(j\omega)$ versus $\log \omega$.

3.1 Exploiting semantic information to minimize algebraic manipulation

An expert is more effective than a novice in doing scientific and engineering computations, not because he is better at computing per se, but because he knows what computing to do and, more importantly, what computing *not* to do. In determining the voltage-transfer ratio of an electrical network, a novice typically writes down many equations and attempts to solve them as a problem of pure algebra. For the expert electrical engineer, in contrast, the algebraic terms carry *meaning.* He knows, for example, that one cannot add a resistance to a capacitance, or that the transfer ratio for a circuit with a series capacitor has no constant term in the numerator. While the novice grapples with a complicated algebraic problem of many variables, the



[illegible] The frequency response of the Twin-T network graphed by the [illegible]h. The [illegible] axis is the magnitude of the voltage-transfer ratio. The [illegible] 1, 2]. The horizontal axis is the base-ten logarithm of the frequency in radians. The horizontal scale is $[-1, 3]$. The parameters are as in figure 2. The notch formed by the zero at $\omega = 1/R_1C_1$ explains the behavior of the output shown in figure 2.

expert can postulate a general form for the result, and can use constraints and consistency checks to determine the detailed answer in a few steps.

Even for small networks, a fully-symbolic frequency-domain analysis would exceed the capacity of all but the most powerful general-purpose algebraic manipulation systems. Dealing with rational functions of many variables is particularly troublesome in symbolic algebra, because, in order to avoid the explosion of intermediate expressions, one must repeatedly reduce quotients to lowest terms, which requires a multivariate greatest-common-divisor computation.[11]

Although the workbench performs symbolic algebra, it also exploits special properties of the domain to minimize the amount of raw algebraic manipulation required. For example, in the Twin-T circuit there are six symbolic device parameters and the frequency variable s. If the algebra is done without reducing rational functions using a full GCD algorithm, but rather by removing only the most obvious common factors, the expression for the voltage-transfer ratio turns out to be the ratio of two seventh-degree polynomials in s each with about 70 terms. This is obviously the wrong expression, because there are only three capacitors, and so the degrees of the numerator and the denominator can be at most three in s. Moreover, by a theorem of P. M. Lin [4], each device parameter can occur to degree at most one.

Unfortunately, the degree requirements alone do not sufficiently constrain the algebra—for six device parameters, a polynomial of degree three in s can have up to 256 terms. To reduce the problem further, the workbench exploits constraints based on the dimensional information declared for each variable. For instance, the sum of a capacitance and a resistance cannot appear in a well-formed expression, because resistance and capacitance have different units; but the expression $RCs+1$ is well-formed because the product of resistance and capacitance has the dimensions of time, and time is the inverse of frequency. The workbench's algebraic manipulator can determine

[11]Sussman and deKleer [3] used the Macsyma symbolic computation system, running on a PDP10, to perform symbolic analysis and synthesis of electrical networks. For all but the very simplest networks, Macsyma was unable to perform the required reductions. Subsequently, Richard Zippel's ***sparse modular algorithm*** [7] enormously improved Macsyma's ability to compute multivariate GCDs. With current algorithms, a circuit of the complexity of the Twin-T network is near the limit of what Macsyma running on a PDP10 can cope with.

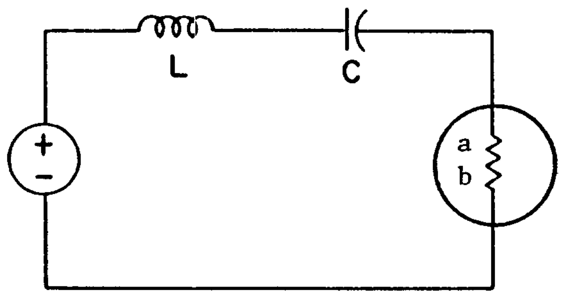


Figure 9: **A driven van der Pol oscillator may be constructed from a series RLC circuit with a nonlinear resistor.**

the units of an algebraic expression. It computes the dimensions of the rational function to be reduced, thereby constraining the possible terms that can appear in the reduced form. In the case of the transfer ratio for the Twin-T network, the possible numerators and denominators turn out to have at most 20 terms each. Such small systems can be easily solved by numerical interpolation, even without a sophisticated GCD algorithm.

4 Periodic orbits of driven oscillators

The elements that the workbench constructs for performing simulations can be incorporated into procedures that perform higher-level analyses of dynamical systems. In this section, we illustrate how the workbench automatically generates programs for investigating the behavior of periodically-driven nonlinear oscillators. One way to study the dynamics of periodically-driven oscillators is through the structures of periodic orbits whose periods are integer multiples of the drive period. Such orbits may be stable, in that small perturbations in initial conditions remain close to the periodic orbits, or they may be unstable. The workbench compiles procedures that find periodic orbits, determine their stability characteristics, and track how the orbits change as the parameters of system are varied.

Figure 9 shows the circuit diagram of a driven van der Pol oscillator, one of the simplest nonlinear systems that displays interesting behavior. The nonlinear resistor has a cubic v-i characteristic $v = ai^3 - bi$ that exhibits negative resistance for small currents and positive resistance for large currents. If there is no drive and the effective Q is large, the system oscillates stably at a frequency primarily determined by the inductance and the capacitance. In state space (v_C and i_L) the undriven oscillation approaches a stable limit cycle. Although the dynamics of the undriven system are well-understood, if we drive the system with a periodic drive, the competition between the drive and the autonomous oscillatory behavior leads to extremely complex, even chaotic behavior. In this section we will use the workbench to explore the behavior of the van der Pol oscillator when driven at a period close to a subharmonic of its autonomous oscillatory frequency.

4.1 Locating periodic orbits

One can find periodic orbits by a fixed-point search. Given an initial state $\mathbf{x}$, integrate the equations through one period of the drive and find the end state $\mathbf{S}(\mathbf{x})$. If the chosen initial state is a fixed point of this *period map* $\mathbf{S}$ then the orbit is periodic. Moreover, the stability of the periodic orbit can be determined by linearizing the period map in a neighborhood of this fixed point and examining the eigenvalues.[12]

Fixed-points can be found by Newton-Raphson iteration, provided we can compute the Jacobian derivative of the period map $\mathbf{x} \mapsto \mathbf{S}(\mathbf{x})$. This can be done by a mixture of numerical and symbolic computation. Since $\mathbf{S}$ is obtained by integrating the system derivative $\mathbf{x}' = \mathbf{F}(\mathbf{x})$, the Jacobian of $\mathbf{S}$ is obtained by integrating the associated *variational system*, which is the linear system

$$(\delta\mathbf{x})' = \mathbf{A} \cdot \delta\mathbf{x} \qquad \text{where } A_{ij} = \frac{\partial F_i}{\partial x_j}$$

Thus we can compute the Jacobian matrix $\mathbf{D_x S}$ by integrating the variational system along the orbit, starting with an orthonormal basis. Even though the integration must be performed numerically, the variational system can be developed symbolically by differentiating the expressions in $\mathbf{F}$. The workbench prepares a system derivative augmented with a variational system for use in this fixed-point search.

[12]This is Floquet's method for analyzing nonlinear systems with periodic drives, generalized by Poincaré for other systems with periodic orbits.

We illustrate this strategy applied to the driven van der Pol system. Here is the system as described to the workbench:

```
(define-network driven-van-der-pol
  ((a parameter v/i^3) (b parameter resistance) (d drive voltage))
  (n1 n2 n3)
  (parts (nl-res nonlinear-resistor (n+ n3) (n- gnd)
                 (vic (lambda (v i)
                        `(= ,v (- (* ,a ,i ,i ,i) (* ,b ,i))))))
         (l inductor (n+ n2) (n- n3))
         (c capacitor (n+ n1) (n- n2))
         (s voltage-source (n+ n1) (n- gnd) (strength d))))
```

and here are the resulting expressions for the system derivative, as computed by the workbench:

```
(rate (v.c t) ($ t)) = (/ i.l c.c)
(rate (i.l t) ($ t)) = (/ (+ (* -1 a i.l i.l i.l)
                             (* b i.l)
                             (d t)
                             (* -1 v.c))
                          l.l))
```

Just as with the Twin-T network of section 2, we can use the system derivative to evolve the time-domain trajectories. Figure 10 shows a particular trajectory. We see from the figure that the trajectory approaches a periodic orbit.

To search for a periodic orbit, the workbench compiles an augmented system derivative generator, shown in figure 11, that (as a function of the parameters) computes a system derivative for the variational system. This is a procedure that given a *variational system state*

$$\texttt{*varstate*} = (t, v_C, i_L, v_C[\delta v_C], v_C[\delta i_L], i_L[\delta v_C], i_L[\delta i_L])$$

computes the state derivative.[13] Combining the system derivative with an integrator produces an end-point procedure that maps variational states to variational states by integrating over a given period. The result is passed through a Newton-Raphson transformation to realize the fixed-point search.

[13]The meaning of the final four components of the state is that $v_C[\delta i_L]$, for example, is the component in the direction v_C of the variation vector δi_L.

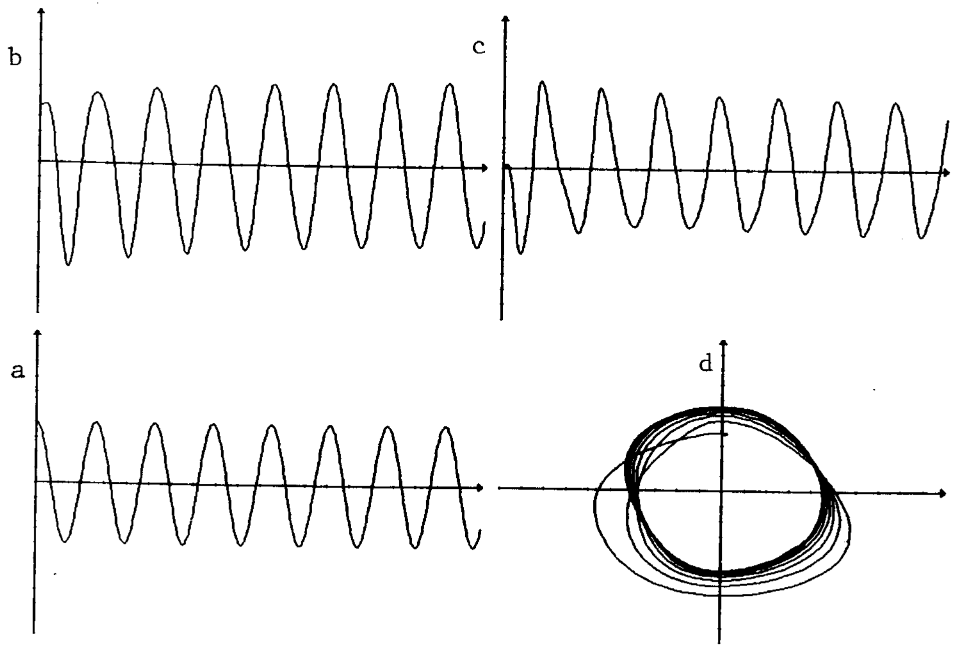


Figure 10: Time-domain plots of the driven van der Pol oscillator show the approach to a periodic orbit. Trace (a) shows the drive. Trace (b) shows the voltage across the capacitor. Trace (c) shows the current through the inductor. Trace (d) shows the state-space trajectory. The abscissa is the current through the inductor and the ordinate is the voltage across the capacitor. We show 30 seconds of simulated time. The voltage scales are $[-100, 100]$. The current scale is $[-0.2, 0.2]$ amperes. The parameters are $C = .001, L = 100, a = 10000, b = 100$. The drive is $d(t) = 40 \cos 1.6t$. In this example, we chose the drive frequency to be slightly higher than half the resonant frequency $1/\sqrt{LC}$. The initial state is $v_C = 37$ volts and $i_L = 0$ amps.

```
(lambda (c.c l.l d b a)
  (lambda (*varstate*)
    (let ((t (vector-ref *varstate* 0))
          (v.c (vector-ref *varstate* 1))
          (i.l (vector-ref *varstate* 2))
          (v.c.del.v.c (vector-ref *varstate* 3))
          (v.c.del.i.l (vector-ref *varstate* 4))
          (i.l.del.v.c (vector-ref *varstate* 5))
          (i.l.del.i.l (vector-ref *varstate* 6)))
      (let ((g27 (* a i.l i.l)))
        (let ((g28 (* -3 g27)))
          (vector 1
                  (/ i.l c.c)
                  (/ (+ (* -1 g27 i.l)
                        (* -1 v.c)
                        (* b i.l)
                        (d t))
                     l.l)
                  (/ v.c.del.i.l c.c)
                  (/ (+ (* -1 v.c.del.v.c)
                        (* b v.c.del.i.l)
                        (* g28 v.c.del.i.l))
                     l.l)
                  (/ i.l.del.i.l c.c)
                  (/ (+ (* -1 i.l.del.v.c)
                        (* b i.l.del.i.l)
                        (* g28 i.l.del.i.l))
                     l.l)))))))
```

Figure 11: This is the augmented system derivative generator compiled to evolve variational states for the driven van der Pol oscillator.

Starting the fixed-point search at the initial state $(37, 0)$ produces the following periodic point:

```
((periodic-point (54.0623 -2.40923e-3))
 (orbit-type spiral attractor)
 (eigenvalues ((*rect* .762164 .187228) mag .784824)
              ((*rect* .762164 -.187228) mag .784824))
 (trace 1.52433)
 (det .615949))
```

The Jacobian matrix not only directs the Newton-Raphson search for the fixed point, but it also provides information about the stability of the fixed point and the associated periodic orbit. For this fixed point, there are complex-conjugate eigenvalues with magnitude less than one. This is therefore a stable fixed point, indicating that the associated periodic orbit is a spiral attractor.[14]

4.2 Tracking periodic orbits

The workbench can also compile procedures to track how periodic orbits move as system parameters are varied.

Let $\mathbf{x} \mapsto \mathbf{S}(\mathbf{p}, \mathbf{x})$ be the period map with explicit dependence on the parameters $\mathbf{p}$ of the system. Let $\mathbf{x}$ be a state such that $\mathbf{S}(\mathbf{p}, \mathbf{x}) = \mathbf{x}$ for a particular choice of the parameters $\mathbf{p}$. For an incremental change $\Delta\mathbf{p}$ we compute, to first order, the corresponding incremental change $\Delta\mathbf{x}$ such that $\mathbf{x} + \Delta\mathbf{x} = \mathbf{S}(\mathbf{p} + \Delta\mathbf{p}, \mathbf{x} + \Delta\mathbf{x})$.

$$\begin{aligned} \mathbf{x} + \Delta\mathbf{x} &= \mathbf{S}(\mathbf{p} + \Delta\mathbf{p}, \mathbf{x} + \Delta\mathbf{x}) \\ &= \mathbf{S}(\mathbf{p}, \mathbf{x}) + \mathbf{D_x S} \cdot \Delta\mathbf{x} + \mathbf{D_p S} \cdot \Delta\mathbf{p} \\ &= \mathbf{x} + \mathbf{D_x S} \cdot \Delta\mathbf{x} + \mathbf{D_p S} \cdot \Delta\mathbf{p} \end{aligned}$$

where $\mathbf{D_x S}$ and $\mathbf{D_p S}$ are the matrices of partial derivatives of $\mathbf{S}$ with respect to $\mathbf{x}$ and $\mathbf{p}$. Subtracting $\mathbf{x}$ from both sides yields

$$\Delta\mathbf{x} = \mathbf{D_x S} \cdot \Delta\mathbf{x} + \mathbf{D_p S} \cdot \Delta\mathbf{p}$$

[14]See Abraham and Shaw [2] for fewer details.

and we conclude that

$$\Delta \mathbf{x} = (\mathbf{1} - \mathbf{D_x S})^{-1}\mathbf{D_p S} \cdot \Delta \mathbf{p}$$

We use the first-order approximation $\mathbf{x}+\Delta\mathbf{x}$ as the initial guess for the actual fixed-point, and iteratively correct the guess with Newton-Raphson.

The matrix $\mathbf{D_x S}$ is computed as in section 4.1, by numerical integration of the symbolically-derived variational equations. $\mathbf{D_p S}$ is also computed by a mixture of symbolic differentiation and numerical integration: $\mathbf{S}(\mathbf{p},\mathbf{x})$ is the integral of the parameterized system derivative $\mathbf{x}' = \mathbf{F}(\mathbf{p},\mathbf{x})$. Thus $\mathbf{D_p S}$ can be found by numerically integrating the symbolically obtained partial derivatives $\partial F_i/\partial p_j$ along the orbit.

For the driven van der Pol system there are four parameters—the capacitance C, the inductance L, and the resistor-characteristic parameters b and a. The machine-generated procedure shown in figure 12 computes the system derivative, augmented by the $\partial S_i/\partial p_j$. The components of the augmented state, `*varstate*`, are t, v_C, i_L, dv_C/dC, di_L/dC, dv_C/dL, di_L/dL, dv_C/db, di_L/db, dv_C/da, and di_L/da.

We can use this procedure to track a fixed point of the driven van der Pol oscillator as we decrease the capacitance so that the resonant frequency of the oscillator passes through a resonance with the second harmonic of the drive. We start at $C = .001$ and decrease it in steps of 5×10^{-6}.

```
((periodic-point (54.0623 -2.40923e-3))
 (orbit-type spiral attractor)
 (eigenvalues ((*rect* .762164 .187228) mag .784824)
              ((*rect* .762164 -.187228) mag .784824))
 (trace 1.52433)
 (det .615949)
 (parameters .001 100 100 10000))
```

The next value of capacitance is .000995, which leads to an estimate for the new fixed-point

```
(estimating-next-point (53.9397 -2.53539e-3))
```

Starting the Newton-Raphson iteration with this guess produces the actual fixed point:

```
(lambda (c.c l.l d b a)
  (lambda (*varstate*)
    (let ((t (vector-ref *varstate* 0))
          (v.c (vector-ref *varstate* 1))
          (i.l (vector-ref *varstate* 2))
          (v.c.c.c (vector-ref *varstate* 3))
          (i.l.c.c (vector-ref *varstate* 4))
          (v.c.l.l (vector-ref *varstate* 5))
          (i.l.l.l (vector-ref *varstate* 6))
          (v.c.b (vector-ref *varstate* 7))
          (i.l.b (vector-ref *varstate* 8))
          (v.c.a (vector-ref *varstate* 9))
          (i.l.a (vector-ref *varstate* 10)))
      (let ((g34 (* i.l i.l)))
        (let ((g30 (* a g34)))
          (let ((g32 (d t)) (g29 (* -1 i.l))
                (g33 (* i.l.l.l l.l)) (g31 (* -3 g30)))
            (vector 1
                    (/ i.l c.c)
                    (/ (+ g32 (* -1 v.c) (* b i.l) (* g29 g30)) l.l)
                    (/ (+ g29 (* c.c i.l.c.c))
                       (* c.c c.c))
                    (/ (+ (* -1 v.c.c.c) (* b i.l.c.c) (* g31 i.l.c.c))
                       l.l)
                    (/ i.l.l.l c.c)
                    (/ (+ v.c
                          (* -1 g32)
                          (* -1 l.l v.c.l.l)
                          (* b g29)
                          (* b g33)
                          (* g30 i.l)
                          (* g31 g33))
                       (* l.l l.l))
                    (/ i.l.b c.c)
                    (/ (+ i.l (* -1 v.c.b) (* b i.l.b) (* g31 i.l.b)) l.l)
                    (/ i.l.a c.c)
                    (/ (+ (* -1 v.c.a) (* b i.l.a)
                          (* g29 g34) (* g31 i.l.a))
                       l.l))))))))
```

Figure 12: This is the augmented system derivative generator compiled to track how periodic points of the driven van der Pol system vary with the system parameters.

```
((periodic-point (53.9402 -2.53249e-3))
 (orbit-type spiral attractor)
 (eigenvalues ((*rect* .793545 .160865) mag .809686)
              ((*rect* .793545 -.160865) mag .809686))
 (trace 1.58709)
 (det .655591)
 (parameters .000995 100 100 10000))
```

The magnitude of the eigenvalues has increased and the magnitude of the phase angle has decreased so the rate of local contraction to the orbit, and the rotation rate, have both slowed.

As we decrease the capacitance, this trend continues until the eigenvalues become real—the local rotation stops and the topological type of the orbit changes to a node, first a barely stable node and then a saddle:

```
((periodic-point (53.4632 -2.97031e-3))
 (orbit-type nodal attractor)
 (eigenvalues (.992466 mag .992466)
              (.841266 mag .841266))
 (trace 1.83373)
 (det .834928)
 (parameters .000975 100 100 10000))

((periodic-point (53.3453 -3.06418e-3))
 (orbit-type nodal saddle)
 (eigenvalues (1.05071 mag 1.05071)
              (.84258 mag .84258))
 (trace 1.89329)
 (det .885305)
 (parameters .00097 100 100 10000))
```

Observe that this transition happens just as the resonant frequency $1/\sqrt{LC}$ passes through twice the drive frequency.

As we further decrease the capacitance, the eigenvalues increase until they are both greater than one; the topological type changes again to a nodal, then a spiral repellor:

```
((periodic-point (52.7759 -3.47088e-3))
 (orbit-type nodal repellor)
 (eigenvalues (1.08793 mag 1.08793)
            (1.07915 mag 1.07915))
 (trace 2.16708)
 (det 1.17404)
 (parameters .000945 100 100 10000))

((periodic-point (52.6649 -3.53871e-3))
 (orbit-type spiral repellor)
 (eigenvalues ((*rect* 1.10764 .113169) mag 1.1134)
             ((*rect* 1.10764 -.113169) mag 1.1134))
 (trace 2.21527)
 (det 1.23967)
 (parameters .00094 100 100 10000))
```

Further decreasing the capacitance leads to even larger eigenvalues with higher rotation rates.

Acknowledgements

The dynamicist's workbench is part of a larger project at M.I.T. to investigate the use of combined numerical and symbolic methods in scientific and engineering computing. Tim Chinowsky, David Espinosa, Bill Siebert, Matthew Halfant, Jacob Katzenelson, and Jack Wisdom contributed to the development of the workbench software. We also thank Bill Dally, Yekta Gürsel, Ken Yip, Eric Grimson, Julie Sussman, Josh Barnes, and Piet Hut for their help with the presentation.

References

[1] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press: Cambridge, MA, 1985.

[2] Ralph H. Abraham and Christopher D. Shaw. *Dynamics—The Geometry of Behavior. Part II: Chaotic Behavior.* Aerial Press: Santa Cruz, CA, 1983.

[3] Johan de Kleer and Gerald Jay Sussman. "Propagation of constraints applied to circuit synthesis." *Circuit Theory and Applications*, vol. 8, pp. 127–144, 1980.

[4] P. M. Lin. "A survey of applications of symbolic network functions." *IEEE Transactions on Circuit Theory*, vol. CT-20, no. 6, pp. 732–737, November, 1973.

[5] Mathlab Group. *Macsyma Reference Manual.* Laboratory for Computer Science, Massachusetts Institute of Technology, 1977.

[6] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes: The Art of Scientific Computing.* Cambridge University Press, 1985.

[7] Richard Zippel. *Probabilistic Algorithms for Sparse Polynomials.* Ph.D. thesis, Massachusetts Institute of Technology, 1979.

CS-TR Scanning Project
Document Control Form

Date : 5 / 26 / 95

Report # AIM-955

Each of the following should be identified by a checkmark:
Originating Department:

☒ Artificial Intellegence Laboratory (AI)
☐ Laboratory for Computer Science (LCS)

Document Type:

☐ Technical Report (TR) ☒ Technical Memo (TM)
☐ Other: ____________

Document Information

Number of pages: 39(43-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☒ Single-sided or
☐ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☒ Laser Print
☐ InkJet Printer ☐ Unknown ☐ Other: ____________

Check each if included with document:

☐ DOD Form ☐ Funding Agent Form ☐ Cover Page
☐ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: ____________

Page Data:

Blank Pages(by page number): ____________

Photographs/Tonal Material (by page number): ____________

Other (note description/page number):

Description : Page Number:
(A) IMAGE MAP (1) UN#'ED TITLE PAGE
(2-39) PAGES #'ED 1-38
(40) SCANCONTROL
(41-43) TRGT'S (3)
(B) CUT&PASTE FIGS ON PAGES 1,4,8,20,26,28,31 (C) XEROX MARKS ON MOST PAGES

Scanning Agent Signoff:
Date Received: 5 / 26 / 95 Date Scanned: 6 / 8 / 95 Date Returned: 6 / 8 / 95

Scanning Agent Signature: [illegible]

Rev 9/94 DS/LCS Document Control Form cstrform.vsd