The real purpose of scientific method is to make sure Nature hasn't misled you into thinking you know something you don't actually know.
If you get careless or go romanticizing scientific information, Nature will soon make a complete fool out of you. It does it often enough anyway when you don't give it opportunities.

— Robert Pirsig
Zen and the Art of Motorcycle Maintenance, p. 94

Appendix B

Performance of the Implementations

This appendix provides some empirical tests measuring the performance of the prototype implementation. The absolute performance numbers are, in a sense, not important as the hardware and the implementation are continually changing. These benchmarks are provided to:

- Provide empirical evidence of the scaling properties of the techniques, as analyzed in Section 4.2.
- Give some idea of the scale of problems solvable with currently (circa 1993) available hardware.
- Provide some intuition for where the performance bottlenecks are.

Understanding the performance of a program running on a modern, high-performance workstation is a difficult task. Complicated factors such as memory hierarchy organization interact in complex ways. For example, different problems may distribute themselves differently along the lines of the data cache. Although we have attempted to design benchmarks that reduce these types of effects, any numbers must be viewed with some caution.

The benchmarks were run on a Silicon Graphics Indigo 2 workstation, except where noted. The machine had a 150MHz R4400 processor and 32 megabytes of main memory. All of the problems fit into main memory on the machine, so no paging occurred. The clock accuracy on the workstation is 10 milliseconds. Where greater precision is reported, it was found by averaging over enough trials that the extra digit remains stable as more trials are run. Some of the simpler benchmarks were run on a Silicon Graphics Personal Iris 4D/25, with a 20MHz R3000 processor, 16 megabytes of main memory,

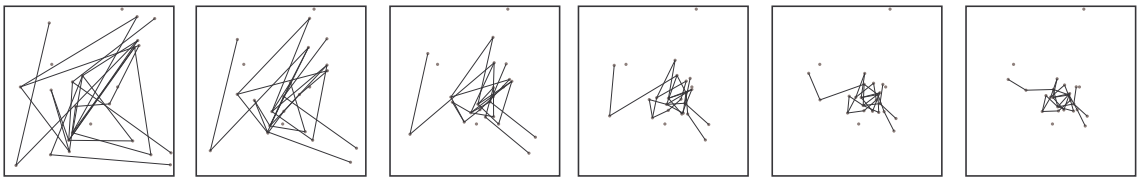

Figure B.1: A sample run of the synthetic benchmark with $m = 50$ and $n = 25$. Lines are used to denote pairs of points with a control placed between them. Each step, the controls push the points closer to one another.

and small instruction and data caches. Current generation personal computers provide significantly better performance than this older machine. Where figures are provided for this machine, they will be explicitly noted.

B.1 A Synthetic Benchmark

In order to evaluate the performance and scalability of the solver, a contrived problem was developed that can be arbitrarily scaled. The problem is designed to mimic realistic problems in which the constraints may have random structure. An instance of the problem can be defined for any number of variables m and constraints n. The problem places $m/2$ points on the plane and places n controls, each connecting 2 points. A control computes the distance between the points and has a `GoTowards` controller placed on it.

A synthetic workload generator creates instances of the problem. It randomly distributes the points in a 10x10 square. Pairs of points are selected to have the distance controls placed on them. The workload generator insures that there are no duplicate controls, but makes no other checks on the distribution of the constraints. An example run is shown in Figure B.1.

Generating arbitrarily sized synthetic workloads for the solver that are reliable measures of performance is difficult. The speed of the solver depends almost as much on the structure of the constraint problem as it does on its size. This can create a bias based on the density of constraints in variables. For example, consider an example with 2 constraints. If there are only 3 variables, no matter how the 2 constraints are placed, the constraints will be in a single partition. However, if there are 4 variables, the constraints might not access any common variables, so the solver will partition the matrix. As the number of variables goes up, the probability of this goes up as well. This can lead to the counterintuitive result that for a fixed number of constraints, larger numbers of variables might actually be faster to solve. For randomly generated problems of the same size, there can be substantial differences in solving time based on how "hard" the system is to solve: if one set of constraints is more tightly connected than another.

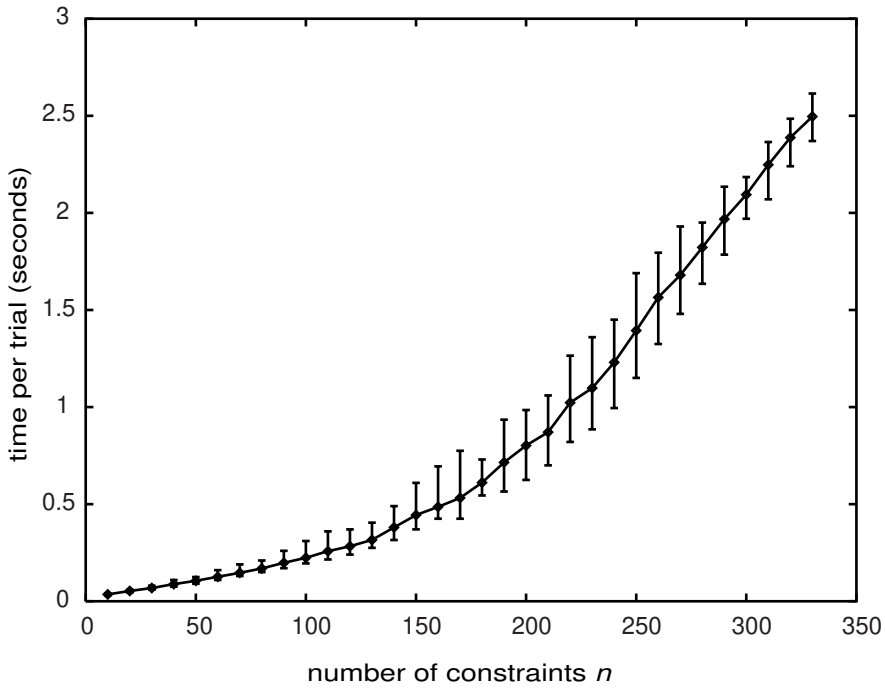

Figure B.2: Results of the synthetic benchmark. The ordinate enumerates the number of controls, the abscissa is the time required to run 5 4th order Runge-Kutta steps. Error bars represent the range of time for each value. The central ticks mark the mean values for the trials. $m = 400$ variables were used in all trials.

To run the trials, 5 constraint configurations and 5 initial positions were generated for each problem size. This leads to 25 different trials per problem size. Each trial was run for 5 Runge-Kutta 4 steps (that is, the differential optimization was solved 20 times). Each trial was duplicated a small number of times.

Graphing the results of running the trials for a fixed number of variables and varying number of constraints yields an expected result, as shown in Figure B.2. The error bars represent the range of time for each problem, while the ticks and line graph show the mean value. The graph shows the expected quadratic performance, but also shows a wide variance of times for a given problem size. Some of this variance can be attributed to how different randomly generated constraint sets can be partitioned.

A similar experiment fixed the number of constraints, but varied the number of variables. The results shown in Figure B.3 are inconclusive: the effects of problem "hardness" are more significant than the number of variables. This is sensible because there are very few parts of the algorithm that are $O(m)$, and all of these have very small constants. While the $O(n^2)$ is able to dominate the problem hardness, the $O(m)$ terms are not.

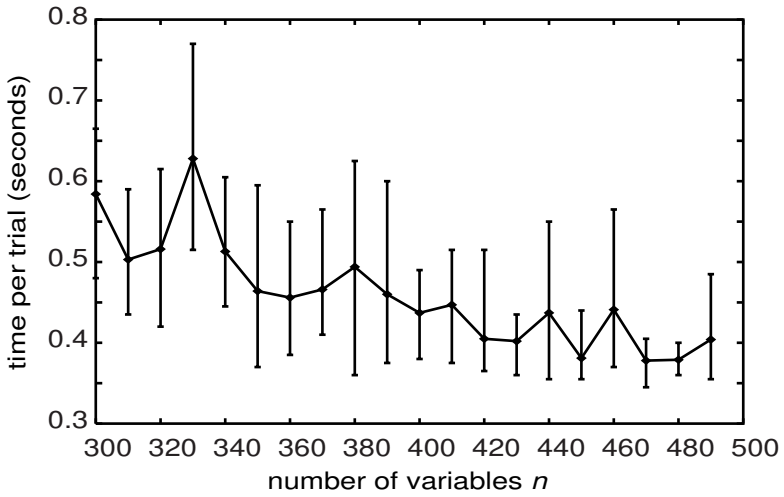

Figure B.3: Results of the synthetic benchmark for 150 constraints and a varying number of variables.

In absolute terms, if we require 5 Runge-Kutta steps per second, an Indigo 2 can handle approximately 200 point distance constraints on 400 variables. This, of course, leaves no time for redraw. More realistically, if we wanted to leave half of the time for drawing and other system functions, the benchmarks could handle approximately 150 constraints. The absolute performance numbers are not really what is important here as they depend heavily on the implementation, the machine, and the problem.

To understand the performance of the benchmarks, a number of trials were run with a version of the benchmark driver compiled with the `pixie` profiling tool available on the Iris. Pixie provides detailed information about where a program spends it time by instrumenting the code. Pixie's output is not hierarchical, so only low level procedures can be accurately monitored.

Table B.1 shows the results of running a number of trials through pixie. The table displays the time in percent that the program spent in the most used basic blocks. The top two lines of the table are important: a very large part of the program's running time is spent in two lines of code. These two lines of code are the half-sparse matrix times vector and half-sparse matrix transpose time vector inner loops. This is not surprising because these form the inner loops of the $O(n^2)$ part of the algorithm, so as the number of constraints grow, the percentage of time in these loops also grows. The `linComb` function computes the linear combination of vectors, and is used to compute the gradients of function blocks. The `cgradI` procedure actually executes the conjugate-gradient solver, but its time does not include time spent in the procedures it calls, including the matrix vector multiply functions, and `dot`, a function that computes the dot product of two vectors. All calls to `dot` in this program occur inside of

procedure	% time, $n = 150$	% time, $n = 200$	% time $n = 250$
`HSpMat::multT`	27.01	28.66	29.96
`HSpMat::mult`	18.62	21.21	23.04
`linComb`	8.39	6.49	4.84
`cgradI`	5.08	6.49	6.58
`dot`	4.16	4.76	5.22

Table B.1: Profiling results for the synthetic constraint benchmark. Numbers represent the percent of total running time spent in the basic procedure blocks. A few of the most time consuming parts of the program are shown.

the conjugate-gradient solver.

B.2 Application Benchmarks

To provide a more realistic evaluation of the absolute performance of the prototypes running on the Iris, the performance of various Bramble applications was measured on a number of examples. In each case, the example object was created beforehand. The timings are measured only while the solver is running with all of the controls, for example during dragging or while a mechanism is being driven by motors.

B.2.1 Direct Manipulation Interaction Techniques

Traditional direct manipulation interaction techniques involve a small number of controls, usually the same number as the degrees of freedom of the input device. When implemented in the differential approach, these direct manipulation techniques usually require some slightly larger number of controls, but still a small constant.

Table B.2 describes the performance of Bramble while executing some of the direct manipulation methods discussed this thesis. All of the techniques require short enough periods of time such that solving will not be the bottleneck in interactive performance. Redraw, which must be done in a conventional direct manipulation implementation as well, is more likely to limit the frame rate. Statistics are also provided for the Personal Iris.

B.2.2 Constrained Models

One use of the differential approach is to permit the user to specify an arbitrary number of controls, in order to provide a constraint-based interface. Here we discuss the performance for constraint benchmarks using Bramble applications with constraint-based interfaces. The important concern here is how large a model can the user create

		Indigo 2 times per		Personal Iris times per	
	controls	RK4 step	redraw	RK4 step	redraw
dragging a spiral Section 8.1.1	2	.001	.01	.01	.05
3D Jack Widget Section 8.3.6	2	.003	.02	.03	.11
Image Alignment Section 8.2.4	8	.009	.02	.06	.07

Table B.2: Performance of various direct manipulation techniques on two different computers. Time is in seconds per 4th order Runge-Kutta step, and for complete redraw of the view window.

before performance becomes unacceptable. The frame rate at which direct manipulation becomes unacceptable seems to vary by application, task, and user. However, the experiments here show that the prototype implementations can permit the direct manipulation of models with dozens of interactive controls, and this number can be raised substantially using the methods of Section 4.4.

A set of benchmarks was run with the MechToy application. For the first set of tests, a number of 5 bar linkages were animated by enabling their motors. An example is shown in Figure B.4. Because the mechanisms are all independent, we would expect that the performance would be linear. Even though the MechToy program does not use partitioning, the conjugate-gradient solver does partitioning automatically for this problem. The expected linear behavior is evidenced in the performance of the system, plotted in Figure B.5. For the case of 9 mechanisms (the most that fit easily on the screen), the Runge-Kutta 4 steps averaged 56 milliseconds, each call to the conjugate-gradient solver averaged 8 milliseconds, and each Jacobian construction averaged 3 milliseconds. Redrawing averaged 46 milliseconds.

The next mechanism example is more tightly coupled: all the pieces are interconnected to form a single 4 bar linkage. As the motor rotates, each "truss" rocks back and forth. No matter how big the mechanism, it only has a single degree of freedom. As more parallel trusses are added, the number of variables and constraints grow. A picture of the mechanism with 5 trusses is shown in Figure B.6. Performance figures are given in Table B.3, and graphed in Figure B.7 This example shows that even with completely connected constraints, models with around 100 constraints are practical on a machine such as the Indigo 2.

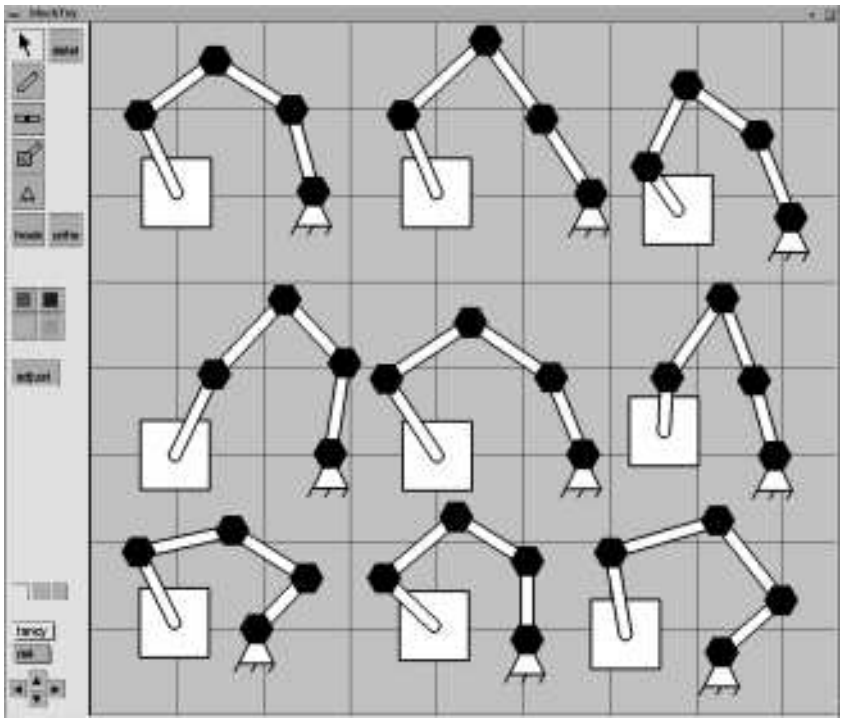

Figure B.4: MechToy animating 9 5-bar linkage mechanisms. Each mechanism is independent of the others, although mechtoy simulates them simultaneously.

.

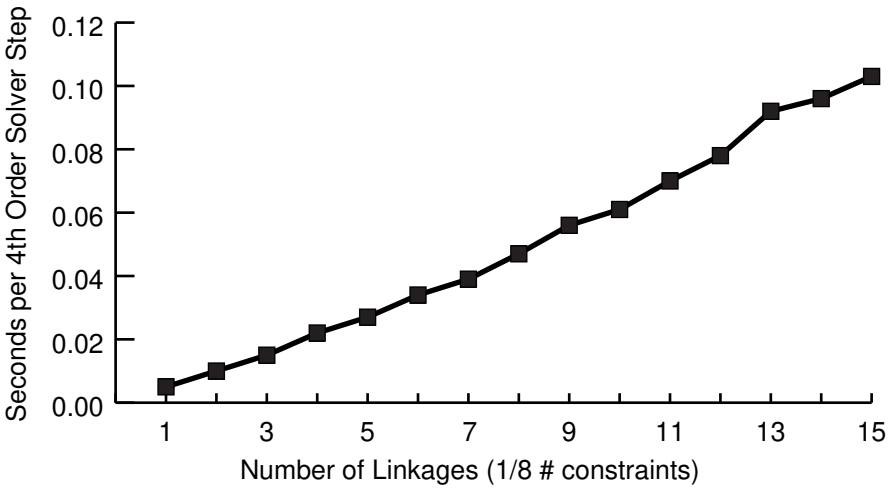

Figure B.5: Performance of MechToy simulating a number of 5 bar linkages simultaneously. The number of constraints is 8 times the number of linkages.

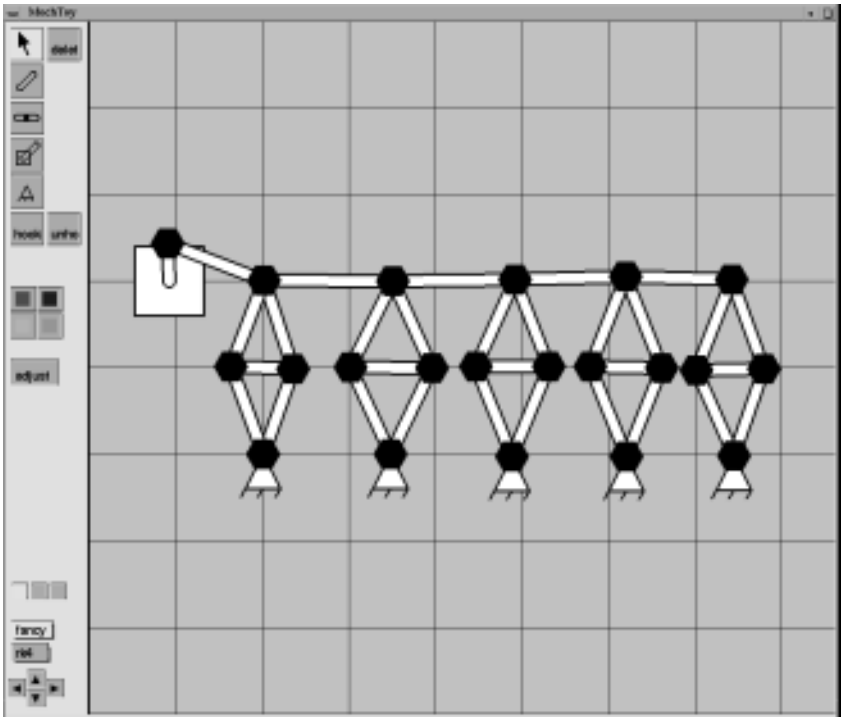

Figure B.6: A 4-bar linkage with 5 parallel trusses.

number of			seconds per				frames/sec	
trusses	vars	consts	step	cgrad	jac	draw	1 step	2 steps
1	19	18	.013	.002	.001	.015	35.4	25.1
2	37	36	.029	.005	.002	.022	19.8	12.9
3	55	54	.048	.008	.002	.029	12.9	8
4	73	72	.068	.012	.003	.038	9.6	5.8
5	91	90	.093	.017	.004	.044	7.3	4.3
6	109	108	.121	.023	.005	.051	5.5	3.4
7	127	126	.159	.031	.005	.062	4.7	2.6

Table B.3: Performance figures for MechToy simulating a mechanism with varying numbers of parallel trusses. Columns denote the time for an average 4th order Runge-Kutta step, solving the linear system with conjugate-gradient, forming the Jacobian, and redrawing the entire view. 4 calls to `cgrad` and Jacobian formation are required for each step. The rightmost columns show the frame rates using 1 and 2 solver steps per redraw.

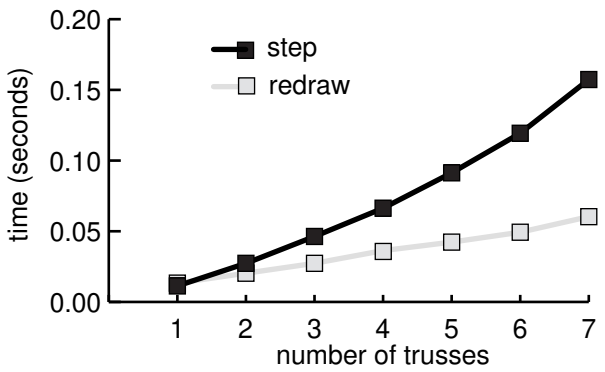

Figure B.7: Performance running the simulation of the truss mechanism. $O(n^2)$ solving time quickly grows to dominate the $O(m)$ drawing time.