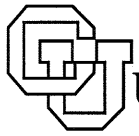
MATHEMATICA AS A PERFORMANCE ANALYSIS TOOL

David B. Wagner

CU-CS-746-94



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Mathematica as a Performance Analysis Tool

David B. Wagner

Department of Computer Science
Campus Box #430
University of Colorado, Boulder 80309–0430

CU-CS-746-94 October 1994

University of Colorado at Boulder

Technical Report CU-CS-746-94
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

Department of Computer Science
Campus Box #430
University of Colorado, Boulder 80309–0430

Mathematica as a Performance Analysis Tool

David B. Wagner

University of Colorado – Boulder

wagner@cs.colorado.edu

1 Introduction

Mathematica [Wolfram91] is a well-known system for doing mathematics by computer. It provides built-in functions for performing an enormous variety of tasks, from factoring polynomials to solving eigensystems, and most of these functions can operate on either numerical (with arbitrary precision) or symbolic arguments. If the built-in functions are not enough, *Mathematica* also provides its users with a very elegant and powerful interpreted programming language. The functionality of the system can also be extended by loading packages, which are ordinary ASCII files containing *Mathematica* programs that conform to certain coding conventions. A few dozen packages come with *Mathematica* (of these, we shall be discussing the symbolic summation and transform packages), and hundreds more are available via anonymous ftp from mathsource.wri.com. Finally, the results of *Mathematica* computations can be rendered in a large variety of graphical formats.

This paper is a tutorial demonstration of *Mathematica*, with a special emphasis on applications to performance analysis. The presentation assumes a basic familiarity with *Mathematica*; for readers with no *Mathematica* experience at all, Appendix A describes the most basic features of the system. Readers with this amount of background should be able to understand at least the gist of the material presented here, although casual users should not expect to understand every nuance of all the code examples contained herein.

2 Comparing the M/M/1 and M/M/2 queues

Suppose that we wish to compare the response time and waiting time of an M/M/1 queue to the corresponding measures for an M/M/2 queue having an identical offered load. First we'll define some functions that calculate these quantities; then we'll use *Mathematica*'s symbolic algebra features to do an analytic comparison. Finally, we'll graphically compare the response times to illustrate some of *Mathematica*'s graphics functions.

2.1 Function definitions

The mean response time and mean service time of an M/M/1 queue [Kleinrock75] having arrival rate λ and service rate μ are as follows:

```
mm1resp[lambda_, mu_] := 1/(mu-lambda)
mm1wait[lambda_, mu_] :=
    Module[ {rho= lambda/mu}, rho/(mu-lambda) ]
```

Here are the corresponding functions for the M/M/2 queue.

```
mm2resp[lambda_, mu_] :=
    Module[{rho=lambda/(2mu)}, 1/(mu(1-rho^2)) ]
mm2wait[lambda_, mu_] :=
    Module[{rho=lambda/(2mu)}, rho^2/(mu(1-rho^2)) ]
```

2.2 Analytic comparison

We first evaluate each of the functions just defined using arrival rate λ for each queue and service rates 2μ for the M/M/1 queue and μ (per server) for the M/M/2 queue. The mean response times are:

```
m1r = mm1resp[lambda,2mu]
```

$$\frac{1}{-\lambda + 2\mu}$$

```
m2r = mm2resp[lambda,mu]
```

$$\frac{1}{\left(1 - \frac{\lambda^2}{4\mu^2}\right)\mu}$$

Which is larger? It's hard to tell, so we'll take a ratio:

```
rratio = m1r/m2r
```

$$\frac{\left(1 - \frac{\lambda^2}{4\mu^2}\right)\mu}{-\lambda + 2\mu}$$

This isn't much better! However, if we substitute $\rho*2\mu$ for λ and tell *Mathematica* to simplify the result, the answer becomes obvious.

```
Simplify[rratio /. lambda -> rho*2mu]
```

$$\frac{1 + \rho}{2}$$

Since $\rho < 1$, this ratio is less than one. Hence, `m1r` < `m2r`.

The `Simplify` command attempts to do general algebraic simplification. More specific algebra commands include `Factor`, `Expand` (the inverse of `Factor`), `Cancel` (for canceling common factors from a quotient), `Apart` (for computing partial fractions), and Together (the inverse of `Apart`).

The analogous computation for average waiting times is:

```
m1w = mm1wait[lambda, 2mu];
m2w = mm2wait[lambda, mu];
Simplify[m1w/m2w /. lambda -> rho*2mu]
```

$$\frac{1}{2} + \frac{1}{2\,\rho}$$

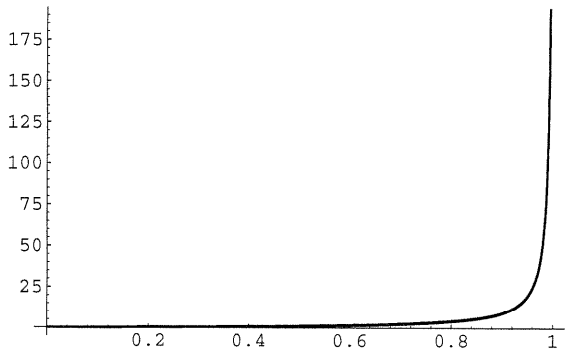
Since $\rho < 1$, this expression is larger than 1. Hence `m1w` > `m2w`.

2.3 Graphical comparison

Let us graphically compare the response times of the M/M/1 queue to the M/M/2 queue. Observe what happens as we attempt to plot response times all the way from $\rho = 0$ to 1:

```
subs = { lambda -> rho*2mu, mu->1 };
Plot[Evaluate[{m1r, m2r} //. subs], {rho,0,1}];
```

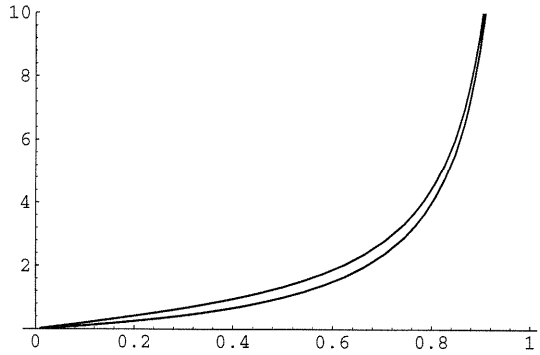
```
                                    1
Power::infy: Infinite expression -- encountered.
                                    0.
```



Mathematica complains, but does the best job it can. (To avoid error messages, we won't plot the endpoints any more.)

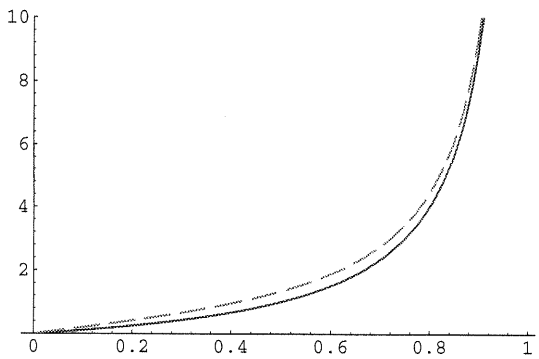
This graph doesn't contain enough detail. To see more, we need to override *Mathematica*'s choice of y-range. This is accomplished by specifying a rule[1] of the form `PlotRange->{ylow,yhigh}` as a trailing argument to the Plot command.

```
Plot[Evaluate[{m1r, m2r} //. subs],
      {rho,0.01,.99}, PlotRange->{0,10}
];
```



This is somewhat better, but the curves ought to be distinguished somehow. Let's make `m1r` red and m2r green and dashed:

```
Plot[Evaluate[{m1r, m2r} /. subs],
     {rho,0.01,.99}, PlotRange->{0,10},
      PlotStyle->{{RGBColor[1,0,0]},
                  {RGBColor[0,1,0], Dashing[{.05,.02}]}}
];
```



3 Solving a Discrete-Time Markov Chain

In this section we'll apply two techniques from *Mathematica*'s arsenal of linear algebra functions to the problem of solving for the steady-state probabilities of a discrete-time Markov chain. Just for good measure, we'll also show how to calculate n-step transition probabilities. In the next section we will write a program to simulate this (or any other) DTMC.

3.1 A machine reliability model

The following problem is from [Taylor84]:

An airline reservation system has two computers, only one of which is in operation at any given time. A com-

1. Many *Mathematica* functions use rules to represent optional arguments.

puter may break down on any given day with probability p. It is assumed that both machines never break down in the same day. There is a single repair facility that takes 2 days to restore a computer to normal. Only one computer can be repaired at a time. Find the long-run probability that both computers are inoperative as a function of p.

The state space is:

- State 0: both machines are operational
- State 1: one machine has just failed, and the other is operational.
- State 2: one machine is operational, and the other has undergone one day of repairs.
- State 3: neither machine is operational, and one machine has undergone one day of repairs.

Note that no other states are possible because of the assumption that only one machine can break down on any given day.

3.2 Method 1: LinearSolve

We could solve this problem by a straightforward application of `Solve` to the state-transition equations for the DTMC. As an alternative to explicitly writing down the state-transition equations, we will instead formulate the state transition matrix `m` and solve the linear system `m.x==x` using `LinearSolve`:

```
?LinearSolve
```

```
LinearSolve[m, b] gives the vector x which solves the
   matrix equation m.x==b.
```

Since `LinearSolve` expects vector multiplication on the right, we have to make each column of the state transition matrix be the conditional pdf of the next state. The state transition matrix is thus (`MatrixForm` simply prints the matrix in columns, without the list braces; it is strictly cosmetic):

```
m = {{1-p,0,1-p,0}, {p,0,p,1}, {0,1-p,0,0}, {0,p,0,0}};
MatrixForm[m]
```

```
1 - p   0       1 - p   0
p       0       p       1
0       1 - p   0       0
0       p       0       0
```

The system `m.x==x` is equivalent to the system `(m-I).x==0` where `I` is the identity matrix. Thus,

```
MatrixForm[m2 = m-IdentityMatrix[4]]
```

```
-p      0       1 - p   0
p       -1      p       1
0       1 - p   -1      0
0       p       0       -1
```

To add the constraint that the state-transition probabilities must sum to 1, replace the first row of `m2` with `{1,1,1,1}`, and change the right-hand side of the equation to `{1,0,0,0}`.

```
MatrixForm[m3 = ReplacePart[m2,{1,1,1,1},1]]
```

```
1       1       1       1
p       -1      p       1
0       1 - p   -1      0
0       p       0       -1
```

```
solution1 = Simplify[LinearSolve[m3, {1,0,0,0}]]
```

$$\left\{\frac{(-1+p)^2}{1+p^2},\ \frac{p}{1+p^2},\ \frac{(1-p)\,p}{1+p^2},\ \frac{p^2}{1+p^2}\right\}$$

3.3 Method 2: Eigensystem

The solution to the Markov chain is an eigenvector of `m`:

```
{vals, vecs} = Simplify[Eigensystem[m]];
```

Since `Eigensystem` returns a list of two elements, "`{vals, vecs} = ...`" performs a simultaneous

assignment to both `vals` and `vecs`. The eigenvalues are:

```
vals
```

$$\{0,\ 1,\ \frac{-p - \text{Sqrt}[(4 - 3\ p)\ p]}{2},\ \frac{-p + \text{Sqrt}[(4 - 3\ p)\ p]}{2}\}$$

Since the second eigenvalue is 1, let's examine the second eigenvector:

```
vecs[[2]]
```

$$\{\frac{(-1 + p)^2}{p^2},\ \frac{1}{p},\ -1 + \frac{1}{p},\ 1\}$$

This does not agree with the previous solution because it is not normalized—that is, the components of the vector don't sum to 1. This is easy to fix (the expression "`Plus@@list`" sums the elements in a list):

```
Simplify[vecs[[2]] / Plus@@vecs[[2]]]
```

$$\{\frac{(-1 + p)^2}{1 + p^2},\ \frac{1}{\frac{1}{p} + p},\ \frac{(1 - p)\ p}{1 + p^2},\ \frac{1}{1 + p^{-2}}\}$$

It is easy to see that this solution is identical to the previous one.

In general, `Eigensystem` is impractical for large symbolic matrices because it solves the characteristic equation for the matrix (which would be a high-degree polynomial). Even if it works, it's painfully slow. However, it is a reasonable choice for numerical matrices.

3.4 n-step transition probabilities

Although it is possible to obtain closed-form expressions for the n-step transition probabilities [Molloy89], it requires the inversion of a matrix z-transform, which is time-consuming and not feasible for larger matrices. For any given `n`, however, m^n can be computed efficiently using the built-in function `MatrixPower`. For example, when `p=0.9` the steady-state probabilities are

```
solution1 /. p->.9
```

```
{0.00552486, 0.497238, 0.0497238, 0.447514}
```

Observe the convergence of the n-step transition probabilities to these values:

```
Table[ Transpose[MatrixPower[m /. p->.9, n]][[1]],
       {n,100,1000,100} ]
```

```
{{0.00399944, 0.332365, 0.0663635, 0.597272},
 {0.00491754, 0.431596, 0.0563487, 0.507138},
 {0.00528306, 0.471103, 0.0523614, 0.471252},
 {0.00542859, 0.486832, 0.0507739, 0.456965},
 {0.00548653, 0.493095, 0.0501419, 0.451277},
 {0.0055096, 0.495588, 0.0498902, 0.449012},
 {0.00551879, 0.496581, 0.04979, 0.44811},
 {0.00552244, 0.496976, 0.0497501, 0.447751},
 {0.0055239, 0.497133, 0.0497343, 0.447608},
 {0.00552448, 0.497196, 0.0497279, 0.447551}}
```

`MatrixPower` can be used on symbolic matrices as well, but the symbolic expressions become unwieldy when `n` is this large.

4 Simulating a DTMC

Simulating a DTMC is an excellent way to demonstrate a wealth of functional programming techniques. (For the uninitiated, Appendix B discusses some indispensable *Mathematica* primitives for functional programming.) After defining the functions necessary to simulate a DTMC, we'll simulate the chain from the previous section, visualize the sample path, and compare the estimated state probabilities to the analytic solution already obtained.

4.1 Function definitions

We first write a function `onestep` that takes the current state of a DTMC as an argument and chooses the next state according to the transition matrix `m`. The general strategy of `onestep` will be to pick a uniformly distributed random number `k` and compare it to the entries in the `k`-th column of `m`.

This would be much easier to do if the rows of `m` were the conditional pdf's, since *Mathematica* stores matrices in a row-major form. This is no problem; we'll simply transpose `m` before we begin:

```
MatrixForm[ mT = Transpose[m] ]
```

```
1 - p   p       0       0
0       0       1 - p   p
1 - p   p       0       0
0       1       0       0
```

Now when we are in state `k`, `k=0..3`, the transition probabilities are given by `mT[[k]]`.

Our task would be easier still if `mT[[k]]` were a CDF rather than a pdf. In other words, we want to create a new matrix whose rows are the cumulative partial sums of the rows of `mT`. This is easily accomplished by `Map`'ing the `cumulsum` function from Appendix B onto mT, since each element of `mT` is a row of the matrix:

```
MatrixForm[ mcdf = Map[cumulsum, mT] ]
```

```
1 - p   1       1       1
0       0       1 - p   1
1 - p   1       1       1
0       1       1       1
```

Now we are ready to write onestep.

```
onestep[state_] :=
    Module[ {cdf, prob=Random[], index=1},
        cdf=cdfMatrix[[state+1]];
        While[cdf[[index]] < prob, index++];
        Return[index-1]
    ]
```

The "`state+1`" at the beginning of onestep and "`index-1`" at the end compensate for the fact that *Mathematica* list indexing is 1-based, not 0-based.

The function has been written in a general purpose manner: First, note that none of the code depends upon the size of the matrix. Second, although it appear that the transition matrix has been hard-coded into the algorithm, this is not the case, since the right-hand side of the function definition (and hence the symbol `cdfMatrix`) will not be evaluated until the function is invoked.

As a final aside, the `Random` function returns a Uniform[0,1] random number by default. Optional arguments can make it return uniformly distributed real or integer numbers over any range; if the `Statistics` packages [Boyland92] are loaded, it can also generate random variates from a wide selection of distributions.

The `simulate` function will take a transition probability matrix, transpose it, convert each row to a cdf, and then iterate `onestep` for the given number of steps. This is much easier than it sounds:

```
simulate[matrix_, initial_, steps_] :=
(
    cdfMatrix = Map[cumulsum, Transpose[matrix]];
    NestList[onestep, initial, steps]
)
```

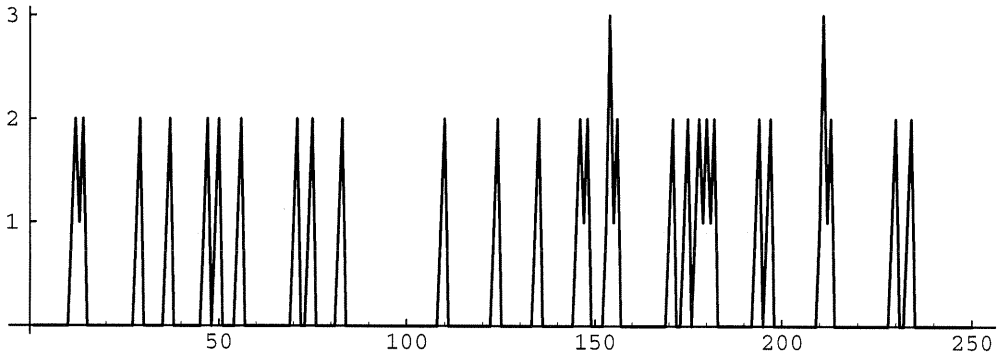
4.2 A sample path

This command computes a sample path of length 1000 for our Markov reliability model with `p=.1`:

```
path = simulate[m /. p->.1, 0, 1000];
```

Here is a plot of the first 250 steps in the sample path. (Plotting more that this makes the individual lines in the plot hard to discern.)

```
ListPlot[Take[path,250], PlotJoined->True,
        PlotStyle->{Thickness[.003]}, PlotRange->All,
        AspectRatio->1/3, Ticks->{Automatic,{0,1,2,3}}];
```



To compute steady-state probabilities we merely divide the number of times the integer `k`, `k=0..3`, appears in `path` by the total number of elements in `path`. The entire steady-state probability vector can be computed with one command:

```
simsolution = Table[N[Count[path, k]/Length[path]], {k,0,3}]
```

```
{0.806194, 0.0969031, 0.0879121, 0.00899101}
```

How does this compare with the analytic solution?

```
analyticsolution = solution1 /. p->.1
```

```
{0.80198, 0.0990099, 0.0891089, 0.00990099}
```

5 Solving Recurrence Relations

The package `DiscreteMath`RSolve`` contains functions for solving recurrence relations [Boyland92]. The package is loaded using the following command:

```
Needs["DiscreteMath`RSolve`"]
```

5.1 Example: the M/M/1 queue

```
RSolve[lambda*p[k] == mu*p[k+1], p[k], k]
```

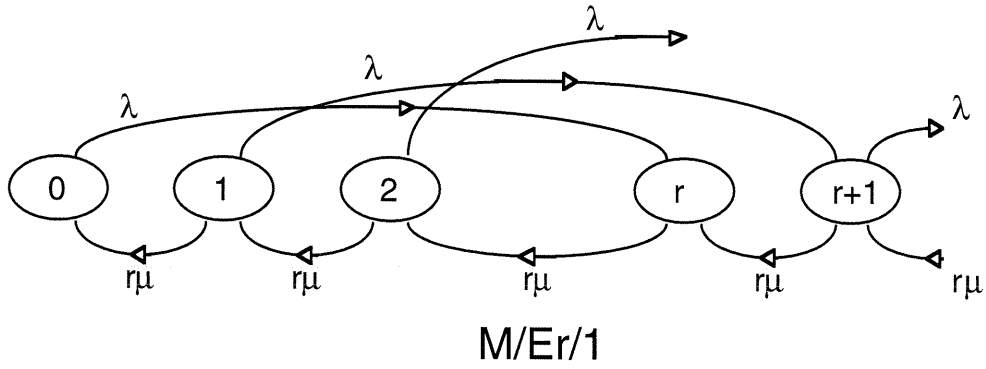
$$\{\{p[k] \to \frac{\text{lambda}^k\ p[0]}{\text{mu}^k}\}\}$$

The first two arguments to `RSolve` are analogous to those of `Solve`; the third argument specifies the index variable. You can specify initial conditions by making the first argument a list of equations:

```
RSolve[{lambda*p[k] == mu*p[k+1], p[0]==1-lambda/mu}, p[k], k]
```

```
{{p[k] -> lambda^k mu^(-1 - k) (-lambda + mu)}}
```

5.2 Example: the M/E$_r$/1 queue



M/Er/1

There are three flow balance equations for this system: one for state 0, a second for states `1..r`, and a third for states beyond `r`. The constraints on the equations passed to `RSolve` are expressed using the syntax "`equation /; constraint`":

```
erlangsolver[r_, lambda_, mu_] :=
    RSolve[{lambda*p[0]==r*mu*p[1],
            (lambda+r*mu)*p[k]==
                r*mu*p[k+1] /; (k>0 && k<r),
            (lambda+r*mu)p[k]==
                lambda*p[k-r]+r*mu*p[k+1] /; k>=r
            }, p[k], k]
```

For `r=1`, this should be M/M/1.

```
erlangsolver[1, lambda, mu]
```

$$\{\{p[k] \to \frac{\text{lambda}^k}{\text{mu}^k}\, p[0]\}\}$$

`r=2` is quite a bit more interesting. However, for reasons unknown, `RSolve` is unable to solve this recurrence for symbolic λ and μ, but is able to solve it for numeric values. We arbitrarily choose λ=3 and μ=4, yielding a queue with a utilization of 3/4.

```
mer2soln = erlangsolver[2, 3, 4]
```

$$\{\{p[k] \to 64\,\left(\frac{\left(\frac{3}{16}-\frac{\text{Sqrt}[105]}{16}\right)^k\left(-\frac{3}{16}+\frac{\text{Sqrt}[105]}{16}\right)}{3\ \text{Sqrt}[105]} - \frac{\left(-\frac{3}{16}-\frac{\text{Sqrt}[105]}{16}\right)\left(\frac{3}{16}+\frac{\text{Sqrt}[105]}{16}\right)^k}{3\ \text{Sqrt}[105]}\right) p[1]\}\}$$

As expected, without an initial condition the solution contains one unknown, `p[1]`. We can eliminate the unknown by setting the sum of all the probabilities to 1. Although the built-in `Sum` function can evaluate only finite sums, the package `Algebra`SymbolicSum`` extends its capabilities to infinite sums.

```
Needs["Algebra`SymbolicSum`"]
Sum[p[k] /. mer2soln[[1]], {k,0,Infinity}]
```

$$\frac{32\ p[1]}{3}$$

Obviously, `p[1]` is 3/32. The utility of rules should now be apparent as we define a function for the fully-solved recurrence:

```
mer2[k_] = p[k] /. mer2soln[[1]] /. p[1]->3/32
```

$$6\left(\frac{\left(\frac{3}{16}-\frac{\sqrt{105}}{16}\right)^{k}\left(-\frac{3}{16}+\frac{\sqrt{105}}{16}\right)}{3\sqrt{105}}-\frac{\left(-\frac{3}{16}-\frac{\sqrt{105}}{16}\right)\left(\frac{3}{16}+\frac{\sqrt{105}}{16}\right)^{k}}{3\sqrt{105}}\right)$$

We noted above that the utilization of this queue is 3/4; as a sanity check, `mer2[0]` should be one minus the utilization:

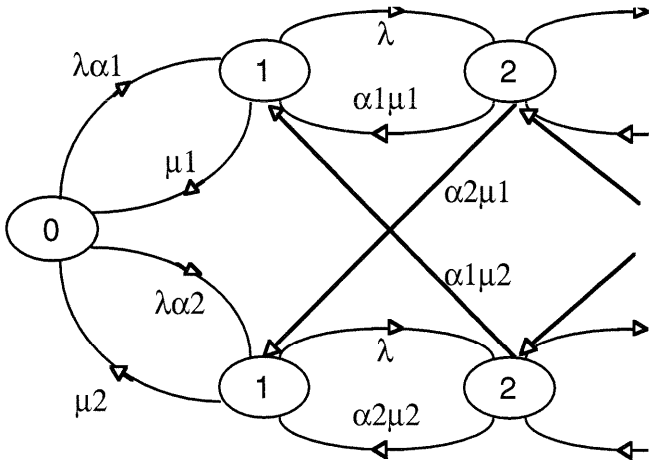
```
mer2[0] // Simplify
```

$$\frac{1}{4}$$

Note that the answer is exact.

5.3 The M/H_2/1 queue

The M/H_2/1 queue models a situation in which customers are of two types: Customers of type j appear with probability α_j and have average service times $1/\mu_j$. The state transition diagram for this queue consists of two parallel chains. A state in the upper chain, which we will denote `a[k]`, indicates that there are k customers at the queue and the customer in service is of type 1. Similarly, a state in the lower chain is denoted `b[k]` and indicates that the customer in service is of type 2. State 0, denoted `s0`, is common to both chains since there is no customer in service.



M/H2/1

Enumerating all of the state transition equations is tedious but straightforward:

```
h2solver[lambda_, {alpha1_,mu1_}, {alpha2_,mu2_}] :=
    RSolve[{
        lambda*s0 == mu1*a[1]+mu2*b[1],
        (lambda+mu1)*a[1] ==
            alpha1*lambda*s0+alpha1*mu1*a[2]+
            alpha1*mu2*b[2],
        (lambda+mu2)*b[1] ==
            alpha2*lambda*s0+alpha2*mu1*a[2]+
            alpha2*mu2*b[2],
        (lambda+mu1)a[k] ==
            lambda*a[k-1]+alpha1*mu1*a[k+1]+
            alpha1*mu2*b[k+1] /; k>1,
        (lambda+mu2)b[k] ==
```

```
            lambda*b[k-1]+alpha2*mu1*a[k+1]+
            alpha2*mu2*b[k+1] /; k>1
        },
        {a[k], b[k]}, k
    ]
```

The following queue has an average service rate equal to 8/5 and a utilization equal to 5/8.

```
h2soln = h2solver[1, {1/4, 1}, {3/4, 2}]
```

$$\{\{a[k] \to \frac{-\left(\left(\frac{2}{5}\right)^{-1+k} s0\right)}{20} + \frac{\left(\frac{2}{3}\right)^{-1+k} s0}{4},$$

$$b[k] \to \frac{3\left(\frac{2}{5}\right)^{-1+k} s0}{20} + \frac{\left(\frac{2}{3}\right)^{-1+k} s0}{4}\}\}$$

Compute `s0` by summing all state probabilities and setting the result equal to 1:

```
Solve[s0+Sum[a[k]+b[k]/.h2soln[[1]], {k,1,Infinity}] == 1]
```

$$\{\{s0 \to \frac{3}{8}\}\}$$

Once again, we note that `s0` is equal to one minus the utilization of the queue's server.

6 The M/G/1 Queue

Mathematica has built-in functions for calculating limits, derivatives, and integrals, and for solving differential equations. In addition, there are standard packages that implement Laplace transforms and z-transforms. Combined with the `SymbolicSum` package demonstrated earlier, these features allow the straightforward calculation of moments and convolutions of both discrete and continuous random variables.

In this section we will demonstrate most of these capabilities (except for differential equations) by solving the *Pollaczek-Khinchin* transform equation [Kleinrock75] for the M/G/1 queue.

6.1 The P-K transform equation

The Pollaczek-Khinchin (P-K) transform equation gives the z-transform of the queue length pdf of an M/G/1 queue in terms of the Laplace transform of the service time pdf. We need to load a package first:

```
Needs["Calculus`LaplaceTransform`"]
```

The following function evaluates the P-K equation for a queue with service time pdf `s[t]`, arrival rate λ, and offered load ρ. (The parameter z simply specifies what the independent variable will be.)

```
nstar[s_, t_, lambda_, rho_, z_] :=
    Module[{sstar},
        sstar = LaplaceTransform[s, t, lambda-lambda*z];
        sstar*(1-rho)(1-z)/(sstar-z)
    ]
```

As an example, consider the M/M/1 queue with service rate μ:

```
nstar[mu Exp[-mu x], x, lambda, lambda/mu, z] // Simplify
```

$$\frac{\text{lambda} - \text{mu}}{-\text{mu} + \text{lambda}\ z}$$

After simplification, the result is recognizable as the z-transform of a geometric distribution.

6.2 Inverting the z-transform

The package `DiscreteMath`RSolve`, which we have already loaded, also contains functions for finding and inverting z-transforms. The function for inverting a z-transform goes by the unlikely name of `SeriesTerm`. For example, to invert the transform found in the previous subsection:

```
SeriesTerm[(lambda-mu)/(lambda z - mu), {z,0,n}]
```

```
lambda^n mu^(-1 - n) (-lambda + mu) If[n >= 0, 1, 0]
```

The result states that the n-th term in the Laurent series expansion around z=0 of the given function is $\lambda^n \mu^{-n-1}(\mu - \lambda)$ when n is non-negative, and is 0 otherwise. In other words, the `If` expression is being used as an indicator function.

This indicator function is rather annoying, but fortunately it can be eliminated. `SeriesTerm` calls the function `Info[n]` to try to get information about the independent variable that might simplify the analysis. For queueing theory it usually makes sense to specify that the independent variable is non-negative. We use this technique in the function `npdf`, which calls `nstar` and tries to invert the result:

```
npdf[s_, t_, lambda_, rho_, n_] :=
(    Info[n] = (n>=0);
    SeriesTerm[nstar[s,t,lambda,rho,z], {z,0,n}]
)
```

Here is the entire calculation for the M/M/1 queue. There's an awful lot going on here: a forward Laplace transform, an inverse z-transform, and a lot of algebra!

```
npdf[mu Exp[-mu x], x, lambda, lambda/mu, k]
```

```
lambda^k mu^(-1 - k) (-lambda + mu)
```

Now we'll use these functions to attack some less trivial problems.

6.3 The M/H$_2$/1 queue revisited

First we must define a pdf for the service time. The pdf below has the same parameters as the M/H$_2$/1 queue that was solved in Section 5.3 using `RSolve`.

```
h2[x_] := 1/4 lambda Exp[-lambda x] +
          3/4 (2 lambda) Exp[-2lambda x]
```

The average service rate of the server is given by the reciprocal of the first moment of `h2`:

```
mu2 = 1/Integrate[x h2[x], {x,0,Infinity}]
```

$$\frac{8\ \text{lambda}}{5}$$

The utilization of this queue, assuming arrival rate lambda, is equal to 5/8. The queue length distribution is

```
nh2[n_] = npdf[h2[x], x, lambda, lambda/mu2, n]
```

$$\frac{3\ (\frac{2}{5})^n}{32} + \frac{9\ (\frac{2}{3})^n}{32}$$

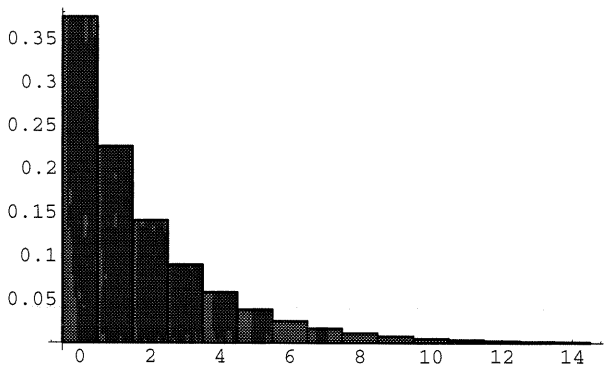
A quick check shows that this answer is identical to the answer obtained by solving the Markov chain:

```
Simplify[h2a[n]+h2b[n]]
```

$$\frac{3\ ((\frac{2}{5})^n + 3\ (\frac{2}{3})^n)}{32}$$

Let's visualize this pdf by plotting it as a histogram. To make such a plot, we have to load yet another package.

```
Needs["Graphics`Graphics`"]
GeneralizedBarChart[
    Evaluate[Table[{k,nh2[k],1}, {k,0,14}]],
    PlotRange->All, AxesOrigin->{-1/2,0}
];
```



6.4 The M/D/1 queue

The degenerate distribution can be defined on the entire real line by expressing it as a Dirac delta function. The function `DiracDelta` was defined when we loaded the `LaplaceTransform` package.

```
dpdf[x_,a_] := DiracDelta[x-a]
```

`a` is the mean (in fact, the only) value of this distribution. We'll arbitrarily choose `a=1/4` and a utilization of 3/4 (which implies an arrival rate equal to three). The z-transform of queue length is easy to find:

```
dstar[z_] = nstar[dpdf[x,1/4], x, 3, 3/4, z]
```

$$\frac{E^{(-3 + 3\,z)/4}}{4\,(E^{(-3 + 3\,z)/4} - z)}(1 - z)$$

However, `SeriesTerm` can't invert this transform.

```
npdf[dpdf[x,1/4], x, 3, 3/4, k]
```

```
Solve::tdep:
   The equations appear to involve transcendental functions
     of the variables in an essentially non-algebraic way.
```

$$\frac{\text{If}[k == 0, 1, 0] - \text{If}[k == 1, 1, 0]}{4} +$$

$$(E^{3/4}\,(\text{SeriesTerm}[\frac{1}{-E^{(3\,z)/4} + E^{3/4}\,z}, z, -2 + k] -$$

$$\text{SeriesTerm}[\frac{1}{-E^{(3\,z)/4} + E^{3/4}\,z}, z, -1 + k]))\,/\,4$$

Without *Mathematica*, we'd be stuck. However, we can have *Mathematica* generate a power series for the z-transform and then read off the coefficients. The built-in function `Series` can symbolically expand a function in a Taylor series about an arbitrary point:

```
Series[f[x], {x,a,2}]
```

$$f[a] + f'[a]\,(-a + x) + \frac{f''[a]\,(-a + x)^2}{2} + O[-a + x]^3$$

Generating the series for `dstar[z]` is very cumbersome, though, due to the large symbolic expressions that are built up as higher order derivatives are computed—it takes about 1 second to generate just the first 6 terms of the series. Numericalizing `dstar[z]` before calling `Series` allows more simplification of intermediate results, which speeds up the computation a great deal (and makes the results much more compact):

```
Series[N[dstar[z]], {z,0,5}] // Timing
```

$$\{0.15\ \text{Second},\ 0.25 + 0.27925\,z + 0.194235\,z^2 +$$

$$0.116667\,z^3 + 0.0676429\,z^4 + 0.0390206\,z^5 + O[z]^6\}$$

Although the result of the `Series` command appears to be a sum, it is not:

```
InputForm[%[[2]]]
```

```
SeriesData[z, 0, List[0.25, 0.2792500041531686671,
   0.1942347603164710381, 0.1166673569112961039,
   0.06764286157987146755, 0.03902055014041806159], 0, 6, 1]
```

The list of coefficients is the third element of this data structure. Putting it all together, we can define a function to generate just the list of coefficients:

```
zprobs[f_, z_, k_] := Series[N[f], {z,0,k-1}][[3]]
```

Here are the first fifteen coefficients. Note once again that `p[0]` is equal to one minus the utilization.

```
zprobs[dstar[z], z, 15]
```

```
{0.25, 0.27925, 0.194235, 0.116667, 0.0676429, 0.0390206, 0.022506,
  0.0129821, 0.0074885, 0.00431962, 0.00249171, 0.0014373, 0.000829084,
  0.000478244, 0.000275867}
```

These fifteen probabilities account for almost all of the probability mass.

```
Plus@@%
```

```
0.999624
```

Of course, even though we were unable to invert the z-transform symbolically, we can still calculate moments of the pdf directly from the derivatives of the z-transform. However, simply substituting `z->1` into the derivative of the transform leads to an indeterminate form:

```
dstar'[z] /. z->1
```

```
Indeterminate
```

We need to take a limit to evaluate this expression. Fortunately, there is a built-in function for doing so:

```
m1 = Limit[dstar'[z], z->1, Direction->1]
```

$$\frac{15}{8}$$

The second moment is

```
m2= Limit[dstar''[z], z->1, Direction->1]+m1
```

$$\frac{231}{32}$$

Note that each of these results is exact. There is no numerical approximation involved, as there was in the calculation of the individual probabilities.

7 Algorithms for Closed Queueing Networks

One of the most powerful programming paradigms offered by the *Mathematica* programming language is rule-based programming. In this section we will demonstrate how to implement some well-known computational algorithms for closed BCMP [Baskett75] queueing networks using rule-based programming.

7.1 Convolution

The convolution recurrence [Buzen73] for computing the normalization constant for a closed BCMP queueing network is:

```
g[n_,k_] := g[n,k-1]+d[[k]]g[n-1,k]
g[0,k_]  := 1
g[n_,0]  := 0
```

where `d` is a vector containing the service demands of the servers in the network. The definitions given above constitute a working *Mathematica* program:

```
d={2,1,7,4};
Table[g[n,k],{n,5}, {k,Length[d]}] // TableForm
```

```
2     3     10       14
4     7     77       133
8     15    554      1086
16    31    3909     8253
32    63    27426    60438
```

This is a simple example of rule-based programming. In *Mathematica*, a function definition is actually a rule definition. Thus, one can have multiple definitions for the same function—these definitions are merely added to the rule base for that function. When *Mathematica* sees an expression of the form `g[_,_]`, it attempts to match that expression to the most specific rule possible. In other words, the expression `g[0,5]` is matched with the rule for `g[0,k_]` rather than the rule for `g[n_,k_]` (matching the latter would result in an infinite loop).

Rule-based programming is a very elegant paradigm because the statement of a problem often constitutes an algorithm for solving that problem as well. Such is the case in this example. Also, rule-based algorithms obviate the need for many, if not all, conditional expressions (e.g., if or switch statements). Instead, *Mathematica*'s main evaluation routine decides which rule to fire based on pattern-matching. (The user can optionally attach boolean conditions to rules.) Finally, rule-based programs are easy to modify. Quite often, the functionality of a program can be significantly extended by adding one or more rules, without any changes to the existing rules—a consequence of the fact that more specific rules always match before more general ones.

Unfortunately, the example program is quite inefficient:

```
Timing[g[10,4]]
{3.61667 Second, 1072817109}
```

The reason for this is that, in the course of evaluating `g[n,k]`, `g[n-1,k-1]` will be evaluated twice: once during the evaluation of `g[n-1,k]` and once during the evaluation of `g[n,k-1]`. In fact, `g[n-i,k-j]` will be evaluated `Binomial[i+j,i]` times, and the asymptotic time complexity of this program is $O(n^k/k!)$.

This situation, called overlapping sub-problems, is characteristic of recursive solutions to a wide variety of very important problems. Because of this, such problems are almost always solved using dynamic programming, wherein a solution is constructed "bottom up" instead of "top down." The main disadvantage of dynamic programming is that it is often non-trivial to write code that evaluates the sub-problems in the optimal order.

However, there is an elegant alternative to dynamic programming: *memoization*. This is a technique in which the results of all recursive calls are cached. The second and subsequent evaluations of a particular sub-problem are thus O(1), reducing the overall running time considerably.

In *Mathematica*, memoization is accomplished by a very modest change to the definition of a function:

```
g[n_,k_] := g[n,k] = g[n,k-1]+d[[k]]g[n-1,k]
```

What is really going on here? When `g` is instantiated with particular values for the formal parameters `n` and `k`—say, `n0` and `k0`, respectively—*Mathematica* evaluates the right-hand side of this definition. This results in the assignment of a value to the expression `g[n0,k0]`. Now, the next time the value of `g[n0,k0]` is required, no recursive call is made.

Here's a demonstration of this. Before any calls to the "new" `g`, this is the rule base for `g`:

```
?g
Global`g
g[0, k_] := 1
g[n_, 0] := 0
g[n_, k_] := g[n, k] = g[n, k - 1] + d[[k]]*g[n - 1, k]
```

Let's see what happens after a call to `g[1,1]`.

```
g[1,1];
?g
Global`g
g[1, 1] = 2
g[0, k_] := 1
g[n_, 0] := 0
g[n_, k_] := g[n, k] = g[n, k - 1] + d[[k]]*g[n - 1, k]
```

Note that a rule for `g[1,1]` has been added to the rule base for `g`. Also note that this rule is listed first in the rule

base, because it is more specific than any of the others. Thus, it will "trap" all calls to `g[1,1]`.

Compare the running time of the memoized function to the running time of the original:

```
Timing[g[10,4]]
{0.3 Second, 1072817109}
```

It's more than 10 times faster for these modest values of `n` and `k`; this factor will grow even larger as these values are increased. Furthermore, all values of `g[n,k]` for $n \le 10$ and $k \le 4$ have been memoized and are available nearly instantaneously:

```
Timing[Table[g[n,k], {k,4}, {n,10}]] // TableForm
0.05 Second
2      3      10          14
4      7      77          133
8      15     554         1086
16     31     3909        8253
32     63     27426       60438
64     127    192109      433861
128    255    1345018     3080462
256    511    9415637     21737485
512    1023   65910482    152860422
1024   2047   461375421   1072817109
```

One drawback of the rule-based approach is that it is essentially recursive, and so stack overflow is a possibility for large enough problems. A second, more subtle drawback is that the rule base has to be re-initialized each time the parameters to the problem are changed. This can be dealt with easily by defining a "wrapper function" that first initializes the rule base and then makes the call to the actual problem-solving code.

The convolution algorithm is simple enough to program in a procedural language. However, there is still a big advantage to using *Mathematica*: one need not worry about numerical underflow or overflow. If the coefficients are all rational, all results have infinite precision. For example, the throughput of the example network with 100 customers in it is

```
g[99,4]/g[100,4]
5699005716347110448611564621749985005853323027305259165709747 97\
   0879886276615029350 /
  3989304001442977314028096622010788217357542336556695301832305\
   4109270934090957269289
```

Or, for those of us who aren't so good at long division,

```
N[%]
0.142857
```

If the coefficients are not rational, simply approximate them with rational numbers or real numbers having any desired precision.

7.2 Mean-Value Analysis

7.2.1 The single-class case

A rule-based program for performing single-class Mean-Value Analysis of a closed BCMP queueing network [Reiser80] is shown below:

```
r[n_] := r[n] = d(1+q[n-1]);
x[n_] := x[n] = n/(z+Plus@@r[n]);
q[n_] := q[n] = x[n] r[n];
q[0] := Table[0, {Length[d]}]     (* Length[d] == number of queues *)
```

This code again assumes that a vector of service demands `d` and also a scalar think time `z` have been pre-defined by the user. We'll use the same demands that we used for the convolution problem, and assume a think time of 0. Then, for example, to find the vector of per-device response times when there are 3 customers in the network,

```
z=0;
r[3]
```

$$\left\{\frac{330}{133}, \frac{148}{133}, \frac{280}{19}, \frac{820}{133}\right\}$$

The throughput is

```
x[3]
```

$$\frac{133}{1086}$$

Note that this is equal to `g[2,4]/g[3,4]`, as calculated by the convolution algorithm.

Here are some noteworthy features of this code:

- `r[_]` and `q[_]` are lists, and `x[_]` is a scalar. Most of the arithmetic operations in this algorithm are performed on entire lists, eliminating the need for any looping constructs.
- How do `r[_]` and `q[_]` "know" how big they should be? The answer is that the length of the lists is unknown until the rule for `q[0]` fires—in fact, *Mathematica* does not even know that `r[_]` and `q[_]` are lists until the base case is reached! Lists of the proper length are returned as the recursion "unwinds."
- Although perhaps not immediately obvious, without the memoization this algorithm would have exponential time complexity. This is because there are two recursive calls to `r[n]`: once in the expression for `x[n]`, and again in the expression for `q[n]`.

7.2.2 The multi-class case

Multi-class MVA is the quintessential dynamic programming problem, and in this case rule-based programming really shines. The code for the multi-class case is not much more complicated than the code for the single class case!

Our basic strategy will be to use almost the exact same rule definitions as before, except that the arguments to the functions (the network population) will now be vectors instead of scalars. Let `c` be the number of customer classes and let `k` be the number of queueing centers. Then the values returned by `x[_]` will be lists of length `c`, and the values returned by `r[_]` and `q[_]` will be two-dimensional lists of size `c*k`. (As pointed out in the previous subsection, the only place in the code where this will be apparent is in the base case for `q[_]`.)

The only tricky part of the algorithm is the recursive calling of `q[_]`. If the given population vector is `{n1,n2,...}` then calls must be made to `q[{n1-1,n2,...}]`, `q[{n1,n2-1,...}]`, and so on. Therefore we need the following utility function that subtracts one from the `c`'th component of a list:

```
subtractone[n_, c_] := ReplacePart[n, n[[c]]-1, c];
```

Here is a snippet of code that subtracts one from each component of a list in turn:

```
Table[subtractone[{x,y,z}, c], {c,3}]
```

```
{{-1 + x, y, z}, {x, -1 + y, z}, {x, y, -1 + z}}
```

Next, in order to compute response times we need to be able to sum `q[_][[c,k]]` over the index `c` (i.e., by columns) and in order to compute throughputs, we need to able to sum `r[_][[c,k]]` over the index `k` (i.e., by rows). So we need two more utility functions:

```
colsums[m_] := Plus@@m
rowsums[m_] := Map[colsums, m]
```

Now we're ready to write down the main rules:

```
r[n_] := r[n] =
    Table[d[[c]](1+colsums[q[subtractone[n,c]]]), {c,Length[n]}]
x[n_] := x[n] = n/(z+rowsums[r[n]])
q[n_ /; VectorQ[n,NonNegative] && Plus@@n > 0] := q[n] = x[n] r[n]
q[n_] := Table[0, {Length[n]}, {Length[d[[1]]]}]
```

Here we see an example of a condition attached to a rule using the "`/;`" construct. The first rule for `q[n_]` will fire only if (a) `n` is a vector whose elements are all non-negative, and (b) at least one element of `n` is greater than 0. The second rule for `q[n_]` is a catch-all case. Note that the dimensions of the result vectors are manifested here. Also note that since the first rule is more specific, it will be checked before the second rule.

Here is an example involving two customer classes and three queueing centers. Note that `z` is now a vector and `d`

is now two-dimensional.

```
z = {5,10};
d = {{1,3,5},{2,4,6}};
r[{3,3}]
```

$$\left\{\left\{\frac{438866}{335861}, \frac{2051667}{335861}, \frac{17372020}{1007583}\right\}, \left\{\frac{603177}{236264}, \frac{2886335}{354396}, \frac{724353}{33752}\right\}\right\}$$

Discounting the three trivial utility functions, our code for multi-class MVA consists of four "one-liners". In addition, this code is completely general: it works for any number of customer classes and any number of queueing centers. The reader is invited to compare this code to a procedural-language implementation of multi-class MVA. Usually, the number of customer classes is fixed by the programmer; otherwise, the programmer has to calculate multi-dimensional loop indices manually. For an example of the latter approach to solving this problem, also using *Mathematica*, refer to [Allen91].

8 Summary

We have taken a whirlwind tour of some features of *Mathematica* that are useful to performance analysts. This is only a limited sample—we haven't even touched the statistical packages or the built-in functions for numerical operations on data, for example. However, it should be quite clear by now that *Mathematica* makes it possible to do symbolic calculations that few human beings could ever hope to perform correctly, if at all, and that the *Mathematica* programming language facilitates elegant solutions to non-trivial problems. In addition, the interpreted environment is ideal for fast prototyping and experimental "poking around" on problems lacking an obvious mode of attack.

Mathematica's programming language and its infinite-precision calculation often make it the tool of choice for numerical as well as symbolic computations. Certainly, the same algorithm coded in C or Fortran will run faster than it will in *Mathematica*—a consequence of *Mathematica*'s dynamic typing, pattern matching, and interpreted execution. But if the programmer's time is factored into the equation, *Mathematica* will often emerge as the fastest path to a solution. Debugging in *Mathematica* is significantly easier than debugging in C or Fortran, for the simple reason that calling a function with symbolic (as opposed to numerical) arguments makes it immediately apparent if the function is doing the right thing.

Finally, *Mathematica* allows users to visualize the results of their computations in a variety of sophisticated ways without any additional programming. In addition to the simple graphics demonstrated here, users can choose from two- and three-dimensional line/surface, parametric, polar/spherical, density, contour, and vector field plots.

References

[Allen91]
A.O. Allen and G. Hynes. Solving a Queueing Model with *Mathematica*. The *Mathematica* Journal 1(3):108–112.

[Baskett75]
F. Baskett, K.M. Chandy, R.R. Muntz, and F.G. Palacios. Open, Closed, and Mixed Networks of Queues with Different Classes of Customers. JACM 22(2):248–260.

[Boyland92]
P. Boyland, J. Keiper, E. Martin, et al. Guide to Standard *Mathematica* Packages, 2e. Wolfram Research, Inc., Champaign, IL, 1992.

[Buzen73]
J.P. Buzen. Computational Algorithms for Closed Queueing Networks with Exponential Servers. CACM 16(9): 527–531.

[Kleinrock75]
L. Kleinrock. Queueing Systems, Volume I: Theory. Wiley & Sons, New York, 1975.

[Molloy89]
M.K. Molloy. Fundamentals of Performance Modeling. Macmillan, New York, 1989.

[Reiser80]
M. Reiser and S.S. Lavenberg. Mean-Value Analysis of Closed Multichain Queueing Networks. JACM 27(1):72–79.

[Schweitzer79]
P. Schweitzer. Approximate Analysis of Multiclass Closed Networks of Queues. In Proc. International Conf. on Stochastic Control and Optimization. Amsterdam, 1979.

[Taylor84]
H.M. Taylor and S. Karlin. An Introduction to Stochastic Modeling. Academic Press, Orlando, FL, 1984.

[Wolfram91]
S. Wolfram. *Mathematica*: A System for Doing Mathematics by Computer, 2e. Addison-Wesley, Reading, MA, 1991.

A *Mathematica* Basics

A.1 Notational conventions

```
This is input to Mathematica.
```

```
This is output from Mathematica
```

A.2 Basic syntax

```
2(3+4)
```

```
14
```

```
ArcSin[1]
```

$$\frac{\text{Pi}}{2}$$

```
Sin[%]
```

```
1
```

```
(-b + Sqrt[b^2-4a c])/(2a)
```

$$\frac{-b + \text{Sqrt}[b^2 - 4\ a\ c]}{2\ a}$$

Note the following:

- Ordinary parentheses are used for grouping; this is the only thing that parentheses are used for in *Mathematica*.
- The asterisk is optional in multiplicative expressions. However, if the asterisk is omitted, a space is necessary in expressions such as "`4a c`". The reason is that *Mathematica* would interpret "`4ac`" as the product of two quantities: the integer `4` and the symbol `ac`. And finally, "`a4c`" is a single symbol.
- Square brackets denote a function call. Thus "`f[x+y]`" is a function call, but "`f(x+y)`" is a multiplication. Note that an alternative function call syntax (for functions of a single argument) is "`expression // f`". This form is usually used at the end of a line to apply formatting functions to a result.
- Expressions can contain symbolic as well as numeric quantities. All system-defined symbols (e.g., `ArcSin`) begin with capital letters, so it's a good idea to begin user-defined symbols with lower-case letters to avoid name conflicts.
- The "`%`" character stands for the result of the previous evaluation. In general, *n* percent-signs in a row refer to the *n*-th previous result.

A.3 Lists

Aside from parentheses and square brackets, there are two other sets of delimiters that one encounters frequently. Curly braces denote a list:

```
{1,2,Sqrt[3]}
```

```
{1, 2, Sqrt[3]}
```

To extract a particular element from a list, use double-square brackets:

```
%[[3]]
```

```
Sqrt[3]
```

Note that lists always use 1-based indexing.

Lists are pervasive in *Mathematica*, and there are many built-in functions for operating on them efficiently. In fact, virtually all of the built-in operators can take lists are arguments. Here are some examples:

```
1+{a,b,c}
```

```
{1 + a, 1 + b, 1 + c}
```

```
{a,b,c}+{d,e,f}
```

```
{a + d, b + e, c + f}
```

```
{a,b,c}{d,e,f}
```
```
{a d, b e, c f}
```

This last result might be surprising—one might have expected a dot-product. This would not be consistent with the way the other arithmetic operators work on lists, however, so there is a special dot-product operator:

```
{a,b,c}.{d,e,f}
```
```
a d + b e + c f
```

More sophisticated operations on lists will be presented in Appendix B.

Lists can be nested, for example:

```
{{a,b}, {c,d}}
%[[1,2]]
```
```
{{a, b}, {c, d}}
b
```

Nested lists are used to represent matrices and the dot-product operator is used for matrix multiplication.

A.4 Exact vs. approximate values

The reader may have wondered why the expression `Sqrt[3]` used above was not evaluated, whereas the expression `ArcSin[Pi/2]` was evaluated. The reason is that `ArcSin[Pi/2]` is exactly equal to `1`, whereas there is no representation of the square root of `3` as a finite string of decimal digits.

Mathematica considers expressions such as `1` (an integer), `Pi` (a built-in constant), `2/3` (a rational number), and `Sqrt[3]` (an irrational number) to have infinite precision:

```
Precision[{1,Pi,2/3,Sqrt[3]}]
```
```
Infinity
```

No exact (infinite-precision) quantity will ever be converted to an approximate (finite precision) quantity unless the user specifically requests it. The numericalization operator is `N`:

```
N[{1,Pi,2/3,Sqrt[3]}]
```
```
{1., 3.14159, 0.666667, 1.73205}
```

The default number of digits of precision (which are not all shown in the output unless specifically requested) is determined by the floating-point hardware of the computer that is being used; on the computer used to prepare this report (which is based on the Motorola 68040 chip), the default precision is

```
$MachinePrecision
```
```
19
```

However, *Mathematica* will gladly approximate exact quantities to any degree of precision requested:

```
N[Sqrt[3], 90]
```
```
1.732050807568877293527446341505872366942805253810380628055806 97\
 945193301690880003708114618 7
```

If we square the previous result, we get back an approximation to the exact number `3`:

```
%^2
```
```
3.
```

(Note the decimal point.) It's a pretty darned good approximation, though.

```
FullForm[%]
```
```
3.000000000000000000000000000000000000000000000000000000000000\
 000000000000000000000000000
```

Note that an input containing a decimal point is always considered to be an approximate number; for *Mathematica* to assume otherwise would be incorrect. For example,

```
3/4-0.75
```
```
0.
```

If the user really means "exactly three-fourths", then that is what should be entered.

A.5 Assignments

There are two assignment operators, "=" (which is called `Set`) and ":=" (which is called `SetDelayed`). For example, to define `n` to be the number `24`,

```
n = 24
```

```
24
```

Note that *Mathematica* has printed out the return value of the expression "`n=24`", which is simply the number `24`. In *Mathematica*, every expression returns a value. Even expressions that have nothing useful to return will return a value, namely the special symbol `Null` (which is never printed unless it is part of a larger expression). The output of an expression can be suppressed (actually, made `Null`) by appending a semicolon to it:

```
x = 2;
```

Now that we have defined `n` and `x` we can used these symbols in other calculations.

```
r = n/x
```

```
12
```

Note that changing the value of `n` or `x` will not affect the value of `r`. Had we instead defined `r` using `SetDelayed`, things would be different:

```
Clear[r]
r := n/x
```

The first thing we notice is that `SetDelayed` does not return a value. This is because the expression on the right-hand side has not been evaluated. The right-hand side is evaluated each time the left-hand side is evaluated:

```
r
```

```
12
```

Thus, subsequent changes to the variables in the right-hand side of `r`'s definition will change the value of `r`:

```
n = 20;
r
```

```
10
```

The example just given is a sort of "poor-man's function definition"; a better way to do this would be to declare `r` as a function of two parameters, `n` and `x`. We'll see how to do this next.

A.6 Defining Functions

A.6.1 A simple function definition

A function for the mean response time of an M/M/1 queue [Kleinrock75] having arrival rate λ and service rate μ can be defined as follows:

```
mm1resp[lambda_, mu_] := 1/(mu-lambda)
```

Note the use of `SetDelayed` to prevent existing definitions of μ and λ from being substituted into the right-hand side of the definition.

The underscore character "_" following each name in the declaration indicates that that name is a formal parameter rather than a literal value. If the underscore had been omitted, as in

```
Clear[foo]
foo[x] := x^2
```

then `foo` would evaluate to `x^2` only when the literal parameter "x" was passed in:

```
foo[x]
```

$$x^2$$

but not for any other parameter:

```
foo[3]
```

```
foo[3]
```

A.6.2 Local variables

Here is the definition of a function for the mean waiting time of an M/M/1 queue:

```
Clear[mm1wait]
mm1wait[lambda_, mu_] := (
    rho = lambda/mu;
    Return[ rho/(mu-lambda) ]
)
mm1wait[a, b]
```

$$\frac{a}{b\,(-a+b)}$$

The parentheses are required to delimit the body of a multi-line function; the semicolon separates the individual calculations. The use of `Return` is unnecessary in this case, since by default a function returns the last result it calculates.

This definition of `mm1wait` has the unfortunate side-effect of defining ρ as a global variable and/or clobbering any previous value of ρ:

```
?rho
Global`rho

rho = a/b
```

This is probably undesirable. A better way of defining the function is to declare ρ to be a local variable. The `Module` command is used to create a local scope:

```
Clear[rho, mm1wait]
mm1wait[lambda_, mu_] :=
    Module[ {rho},
        rho = lambda/mu;
        rho/(mu-lambda)
    ]
```

Now, the value of any global variable named ρ will be unchanged by `mm1wait`:

```
rho = Sqrt[19];
mm1wait[1,2]
rho
```

$$\frac{1}{2}$$

```
Sqrt[19]
```

A.7 Rules

An expression of the form "a->b" is called a *rule*. A rule is used to substitute one expression for another using the `ReplaceAll` operator "/.". For example:

```
rho/(1-rho) /. rho->lambda/mu
```

$$\frac{\frac{lambda}{mu}}{\left(1-\frac{lambda}{mu}\right)\,mu}$$

Rules are encountered frequently because they are returned by functions that solve equations. For example, here is the solution to a very simple set of equations (the "==" means boolean equivalence, as opposed to assignment):

```
Clear[x,y]
sol = Solve[{x^2+y^2==a^2, x-y==b}, {x,y}]
```

$$\left\{\left\{x \to \frac{b - Sqrt[2\,a^2 - b^2]}{2}\right.,\right.$$

```
              2          2    2
     -2 b - Sqrt[4 b  - 8 (-a  + b )]
y -> ---------------------------------},
                    4

            2    2
     b + Sqrt[2 a  - b ]
{x -> -------------------,
              2

              2          2    2
     -2 b + Sqrt[4 b  - 8 (-a  + b )]
y -> ---------------------------------}}
                    4
```

The first parameter is a list of equations to be solved, and the second parameter is a list of the variables to solve for. (Note that it is not necessary to specify the variables to solve for when there are only as many unknowns as there are equations.)

Each of the sub-lists in the result is a solution to the given system of equations. Note that `Solve` always returns a two-dimensional list, even when there is only one solution.

We have named the solution `sol` so that it is easy to use these rules in future computations. For example,

```
x^2+y^2 /. sol[[1]]  // Simplify

 2
a
```

A.8 Getting help

Information on any symbol can be obtained with the expression `?name`:

```
?Sqrt

Sqrt[z] gives the square root of z.
```

This even works for user-defined symbols, to the extent possible:

```
?r

Global`r

r := n/x
```

This concludes our discussion of *Mathematica* preliminaries. The interested reader should now understand the syntax and semantics of *Mathematica* statements well enough to follow nearly all of the examples contained in this paper.

B Some functional programming primitives

For those who are unfamiliar with functional programming techniques, the material in this appendix is a prerequisite for Section 4 (simulating a DTMC) and Section 7 (closed queueing network algorithms).

B.1 `Map` and `Apply`

Anyone with experience using a functional programming language such as Lisp is familiar with the concept of mapping and applying functions to lists. In *Mathematica*, these operations are called `Map` and `Apply`.

`Map` simply wraps a given function around each element of a list:

```
Map[Sqrt, {a,b,c}]

{Sqrt[a], Sqrt[b], Sqrt[c]}
```

This example is not particularly useful because the same result could have been obtained by using `Sqrt[{a,b,c}]`. More commonly, Map is used to achieve this effect with user-defined functions.

`Apply` takes a function and a list as arguments and passes all of the elements of the list as parameters to that function. For example,

```
Apply[Plus, {a,b,c}]

a + b + c
```

A shorthand notation for `Apply[f, list]` is `f@@list`.

B.2 NestList and FoldList

Mathematica contains several functions for composing a given function repeatedly.

`NestList` composes a function of a single argument a given number of times, starting with a given initial value:

```
NestList[f, a, 4]
{a, f[a], f[f[a]], f[f[f[a]]], f[f[f[f[a]]]]}
```

The function `FixedPointList` is analogous to `NestList`, except that it continues to compose the given function until the generated sequence converges or the iteration limit is reached. (An optional argument can specify the function to be used to test for convergence.) This is useful for performing fixed-point calculations such as approximate Mean-Value Analysis [Schweitzer79].

The function `FoldList` is somewhat like `NestList`, except that `f` is a function of two arguments:

```
FoldList[f, a, {b,c,d}]
{a, f[a, b], f[f[a, b], c], f[f[f[a, b], c], d]}
```

In particular, if `f` is the `Plus` function, `FoldList` will compute cumulative partial sums of a list. Here is a function that takes a list as an argument and returns a list of the cumulative partial sums of its elements:

```
cumulsum[l_] := FoldList[Plus, First[l], Rest[l]]
cumulsum[{a,b,c,d}]
{a, a + b, a + b + c, a + b + c + d}
```