

Best MATLAB Practices for Queueing Theory

Andrew M. Ross

Industrial and Systems Engineering Department
Lehigh University

2005-07-21

Main MATLAB tip: take someone else's code and change it.

Outline, Part 1

General stuff

Structured Programming

How I Do Experiments

Plotting

Misc.

- Common Handy Functions

- Other Tips

Outline, Part 2

PH Distributions

Steady-State CTMCs

ODEs and CTMCs

More Info:

1. “help” in Matlab: `help interp1`
2. “lookfor” in Matlab: `lookfor interpolate`
looks in 1st line of many .m files for that word.
Oops! Doesn't find `interp1`; use “`lookfor interpol`”
3. www.mathworks.com/access/helpdesk/help/helpdesk.html
has details not given in “help”!
4. comp.soft-sys.matlab
5. Matlab File Exchange: www.mathworks.com/matlabcentral/
6. unofficial Matlab style guide: www.datatool.com/prod02.htm
which I don't always follow.

Unix, Windows Tips

- ▶ Probably runs slowest under Windows.
- ▶ Java desktop probably slows things down in either case.
in Unix, start via “matlab -nojvm -nosplash”
- ▶ Infinite loops are a lot easier to stop under Unix.
- ▶ Matlab very slow across network under Windows:
 1. copy it from Y:/lib/matlab7r14sp2 to c:/Program Files
takes about an hour(!) to copy
 2. create a shortcut to “C:/Program
Files/matlab7r14sp2/bin/win32/MATLAB.exe” /nojvm
/nosplash
 3. Tell the shortcut to start in c:/aross/matlab or whatever.
Definitely NOT y:/lib/matlab7/work which is its default.

Style Tips: Variable Names

1. Avoid keywords and existing function names:

```
>> which pvec  
/opt/matlab/7R14/toolbox/lmi/lmictrl/pvec.m  
>> which prvec  
'prvec' not found.
```

2. Usual suspects: mean, std, var
3. If it's a vector, name it (something)vec;
if it's a matrix, name it (something)mat
Example: instead of $A*x \leq b$, use $A_{\text{mat}} * x_{\text{vec}} \leq b_{\text{vec}}$
4. Avoid single-letter names, especially i, j, l, O
 $i, j, = \sqrt{-1}$
the letter l is hard to tell from the number 1, same for O and 0
5. I use an “n” prefix for loop variables: ni, nj, nk, n1, n2, etc.

Style Tips: Page Formats

1. Keep lines under 80 columns.
2. Use ellipses (three dots, ...) to break up lines:

```
yvec = ICantBelieveHowLongThisNameIs(lambda, ...
    mu, servers);
```

3. Especially use them to show parallelism:

```
plot(xvec,upperbound,'-v',xvec,approx,'-+',xvec,true,'-
o',xvec,lowerbound,'-^');
```

versus

```
plot( ...
    xvec, upperbound , '-v', ...
    xvec, approx     , '-+', ...
    xvec, true       , '-o', ...
    xvec, lowerbound , '-^' ...
);
```

Everything I do, I do it for the referees

Odds are, you will have to revisit your work in a year or two.

Perhaps:

- ▶ You will find, 3 hours before your defense, a bug in your code, or
- ▶ A padawan learner you are, and the referees (who are older and wiser than you) will require some changes, or
- ▶ You are the smartest person ever, but those idiot referees still demand some changes.

In some cases, sed 's/referees/advisor/g'

So, structure your work now so you can more easily re-do it later.

Directory Structure

Don't put all your Matlab files in one directory!
Have a central directory that holds generic utilities and downloaded packages.

```
/home/aross/:
matlab/
  birthdeath.m
  exportfig.m
  phlib/
queues/:
estim-capacity/:
  matlab/:
    estim_capacity.m
    play0.m
    do_size.m
    graph_size.m
other-project:
  matlab:/
```

Paths, Startup

Add your main tools directory to the matlab path: put in file startup.m,

```
format compact % suppress extra line-feeds
format short g % e.g. 5 or 5.001 or 5.001e-10
if( isunix )
    addpath /home/aross/matlab
else % assume ‘‘ispc’’ is true--what else is there? :-)
    addpath g:\aross\matlab
end
```

Use Functions!

Files that end in .m have two types: functions, and scripts.

Functions start with “function [outputargs] =
functionname(inputargs)”

Scripts don't.

silly.m:

```
function [yvec] = silly(xvec)
% SILLY does silly stuff to xvec
% function [yvec] = silly(xvec)
yvec = xvec .^ 2 ; % square each component of xvec
return; % not required but nice.
```

myscript.m:

```
xvec = -10:10;
yvec = silly(xvec); % if single output, [] are not required
plot(xvec,yvec);
```

Don't do this:

```
function [yvec] = silly(xvec)
xvec=input('Enter the xvec> ');
yvec = xvec .^ 2 ;
```

The input arguments are how the function gets those values;
no need to ask the user for them.
See previous slide for proper way to do things.

Function Comments

First line of file: `function [mm] = minmax(xvec)`

Second line of file:

```
% MINMAX reports the minimum and maximum of xvec  
That is the line that “lookfor” searches, put key words there!
```

Third line of file: `% function [mm] = minmax(xvec)`
so users can see calling syntax, also good for cut-and-paste.

Fourth, etc:

- ▶ explanation of what input and output arguments mean,
- ▶ things to watch out for,
- ▶ theory of how the computation is done,
- ▶ mention related functions, etc.

Function Arguments

Instead of

```
[mu1,mu2]=myfunc(lambda1,lambda2,pval1,pval2);
```

use

```
muvec=myfunc(lamvec, prvec);
```

That way, if you want to expand to lambda1, ... lambda10 you don't have to re-write the function call.

Many Function Arguments

```
[L,Lq,W,Wq,pb,pd,prvec,prmat,Qmat]=myqueue(lambda,mu,servers,room
```

Ick! Use structures instead:

```
sysinfo.lambda = 9;  
sysinfo.mu = 1;  
sysinfo.servers = 10; % etc.  
perf = myqueue(sysinfo);  
perf.L  
perf.Wq % etc.
```

Also lets you ignore ordering of arguments.

Downside: to do plots, must then put into vectors. Oh well.

How I Do Experiments

I tend to have:

1. Some files that do actual computations (`mph2sr`, `mmsr_abandon`, `sim_once`, etc.)
2. Some “play” files `play0.m`, `play1.m` that are first playing-around attempts.
3. Some computation drivers like `do_svcrate.m`, `do_abanrate.m`, etc.
4. Those files save their data to `.mat` files
5. Some graphing files `graph_svcrate.m`, `graph_abanrate.m` which read from the `.mat` files.

do abanrate.m

```
sysinfo.servers=10;
sysinfo.lambda=9; % etc
avec = 0:.1:2; % various abandonment rates
for ni=1:length(avec)
    arate = avec(ni) % no semicolon
    sysinfo.arate = arate;
    aperf = mmsr_abandon(sysinfo);
    sperf = sim_once(sysinfo);
    a.Lvec(ni) = aperf.L;
    s.Lvec(ni) = sperf.L; % then Wq, etc.
    save do_abanrate
end % for ni
```

Watch out for “save do_abanrate.m”—it will write over your .m script!

graph abanrate.m

```
load do_abanrate

% sometimes we may want to switch:
xvec = avec; % patience rate
% versus
xvec = 1./ avec; % mean patience

plot(xvec, Lvec);
```

This way, you can re-do the plot without re-running the computation.

Plotting

Plot with Publication/Presentation in mind.

Avoid GIF and especially JPG, resolution is too low.
Use EPS or PDF.

Size: IEEE standard is 3.5 inches wide, text at 8 points.

Color? Useful for e-version of journal, and presentations, so yes!
But, be sure it is intelligible in black and white, too.

Could export data to Excel/gnuplot/whatever then graph,
but that's a lot of manual labor, which has to be repeated if you
get new data.

Plotting: How To

Download “exportfig” package from Matlab File Exchange. Then, put into plotsettings.m (or whatever filename you want):

```
opts.Width      = 3.5; % in PaperUnits, usually inches.  
opts.FontSize   = 8  ; % in Points  
opts.LineWidth  = .5; % in Points  
opts.Color='rgb'; % or 'bw'  
% magic:  
opts.LockAxes   = 1;  
opts.FontMode   = 'fixed';  
opts.LineMode   = 'fixed';  
opts.BoundsCode = 'mcode';
```

Now you can do

```
plot(avec,Lvec);  
xlabel('Abandonment Rate (per minute)')  
ylabel('L=E[num. in system]')  
title(sprintf('%d servers',sysinfo.servers));  
  
fname = 'abanrate.eps';  
exportfig(gcf,fname,opts);
```

I then use “epstopdf” in Linux to convert to PDF.

Common Handy Functions: Numeric

- mean** as opposed to Excel's "average"
- var** variance
- std** standard deviation (Excel: stdev)
- sort** sorts a single vector
- sortrows** sorts a whole matrix based on one column
- unique** everything in a vector but no repetitions.
- diff** $x(2) - x(1)$, $x(3) - x(2)$, etc.
- interp1** (that's a "one") Linear or spline interpolation
- regress** Multiple linear least squares
- eig** Eigenvalues and eigenvectors
- chol** Cholesky decomposition
- trapz** Trapezoidal numerical integration
- quad** fancier numerical integration

Plotting

“help graph2d” and “help graph3d”

`plot` ordinary 2-d plots

`semilogy` and `semilogx` and `loglog`

`subplot` multi-panel plots

`hist` histogram

`errorbar` and `boxplot`

`stairs` good for sample paths

`plot3` plots a 3-d curve

`surf` Plot a 3-d surface

`contour` plots contours of a 3-d surface

`spy` sparsity plot of a matrix

also try “help specgraph” for specialized graphs.

Optimization

“help optim” highlights:

`fzero` One-dim $f(x) = 0$

`fsolve` Multi-dim $f(\vec{x}) = \vec{0}$

`fminbnd` One-dim

`linprog` Linear programming

`fminunc` Multi-dim, unconstrained

`fmincon` Multi-dim, constrained

`fminsearch` Nelder-Mead, unconstrained, not so good.

Andrew's Functions

Various stuff I wrote and find useful:

`ratios` like `diff` but with `/` instead of `-`

`cov2corr` turn Covariance matrix into Correlation matrix

`corr2cov` opposite of `cov2corr`

`inm` Incomplete Normal Mean, $\int_a^b x \cdot \phi(x) dx$

`igm` Incomplete Gamma Mean

`minmax` $[\min(x), \max(x)]$

`autotextread` read data from a file into a structure, using column names from first line of file.

Other Tips: Row vs Column vectors

Row vectors:

- ▶ `-10:10`
- ▶ `for ni=1:10; tmp(ni)=sin(ni); end`
- ▶ `tmp(:)'`

Column vectors:

- ▶ `(-10:10)'` or `[-10:10]'` but not `-10:10'`
- ▶ `tmp(:)`

Speed ideas

- ▶ Allocate space before creating a large vector.

```
>> clear
```

```
>> tic; for ni=1:30000; tmp(ni)=1; end ; toc
```

```
Elapsed time is 10.956998 seconds.
```

```
>> clear
```

```
>> tic; tmp=zeros(1,30000);for ni=1:30000; tmp(ni)=1; e
```

```
Elapsed time is 0.090426 seconds.
```

- ▶ Avoid `inv()` for large matrices: for 1000x1000,

```
>> tic; xvec = inv(Amat)*bvec; toc
```

```
Elapsed time is 2.094096 seconds.
```

```
>> tic; xvec = Amatl\bvec; toc
```

```
Elapsed time is 0.863323 seconds.
```

*Unless you will be doing `inv(Amat)*bvec` for many different `b` vectors, in which case consider LU factorization.*

Numerics

- ▶ Don't compare floating point numbers directly.

```
x = 1/3;  
y = 4/3 - 1;  
if( x == y ) % this will fail! Instead, use  
if( abs(x-y) < tol ) % tol is 1e-5, 1e-10, or 1e-15
```

- ▶ The number closest to zero is “realmin”= $2.2e-308$, but the number closest to 1 is “1-eps”, eps is $2e-16$
Who cares? If you want a very small CCDF, calculate it directly instead of doing 1-CDF.
See log1p, expm1

PH Distributions

I just wrote “phlib” which has:

- ▶ PDF, CDF, CCDF, moments, random samples
- ▶ $Z = X + Y$, $Z = \text{mix}(X, Y)$, $Z = \min(X, Y)$,
 $Z = \max(X, Y)$, $\Pr\{X > Y\}$
- ▶ LST, hazard rate, Mean Residual Life
- ▶ Moment Match PH to mean, var, 3rd moment

Pronunciation: “flib” is clearly more fun to say than “p-h-lib”.

This allows us to do:

```
mymean = 1; % avoid using 'mean' as var. name  
mycv = 2;  
myvar = (mymean*mycv)^2;  
PH = ph_mmatch(mymean,myvar,{'coxg','bm'});  
perf = mph2sr(lambda, PH, servers, room);
```

instead of testing if $cv > 1$ or < 1 , then calling Cox, Hyper, or Hypo.

3rd Moment Options

$cv < 1$	$cv > 1$
cox	min3
mg	min3e
hypo	g
	cox
	bm
	bp
#	#

if $cv < \sqrt{2} \approx .707$, forced to use more than 2 phases, only one 3rd-moment option currently.

Steady-State CTMCs

1. Form Q matrix
2. Find steady-state solution
3. Interpret

Form Q matrix

Many interesting CTMCs have a 2-, 3-, or 4-dim. state space, e.g. $(ns, no) = (\# \text{ in system}, \# \text{ in orbit})$ for M/M/s/r+retrials

Must map into 1-dim *and* start at 1 rather than 0.

```
function statenum = told(ns,no,sys_cap,orbit_cap)
%statenum = ns * (orbit_cap +1) + no + 1; % or
statenum = no * (sys_cap +1) + ns + 1;
```

Note: I prefer (ns, no) to be 0-based rather than 1-based.

My philosophy: keep things in intuitive units, convert to Matlab weirdness only when forced to.

Also, keep mapping from (ns, no) to statenum deeply buried, so you can change it easily.

What calls what?

```
do_svcrate.m
```

```
  mmsr_retrials.m
```

```
    mmsr_retrials_mat.m
```

```
      to1d
```

```
        ctmc_stationary_probs.m
```

We would then fill in arrivals like this:

```
% ordinary accepted arrivals:
for ns=0:(sys_cap-1)
    for no=0:orbit_cap
        from = told(ns ,no,sys_cap,orbit_cap);
        to   = told(ns+1,no,sys_cap,orbit_cap);
        Qmat(from,to) = lambda; % see caveat later
    end % for no
end % for ns
% arrivals who get blocked and enter orbit:
ns = sys_cap;
for no=0:(orbit_cap-1)
    from = told(ns,no ,sys_cap,orbit_cap);
    to   = told(ns,no+1,sys_cap,orbit_cap);
    Qmat(from,to) = lambda*pr_retry1;
end % for no
```

and similarly for services, retrials, etc.

Helper vectors

mmsr_retrials_mat.m should also create “helper” vectors:
For a 1-dim system,

$$L = \sum_{n=0}^K n \cdot p_n = \vec{p} \cdot [0, 1, 2, \dots K]'$$

So here $[0, 1, 2, \dots K]'$ is a helper for L, allowing us to do
`L=prvec * helper.L`

For a 2-dim system, helper vec could be $[0,1,2,0,1,2]'$ or
 $[0,0,1,1,2,2]'$ depending on mapping.

```
for ns=0:sys_cap
    for no=0:orbit_cap
        statenum = told(ns,no,sys_cap,orbit_cap);
        helper.Ls(statenum) = ns;
        helper.Lo(statenum) = no;
        helper.L(statenum) = ns+no;
    end % for no
end % for ns
```

Find steady-state solution

Options:

1. set $\pi_1 = 1$ or $\pi_{\text{end}} = 1$, solve, renormalize
2. replace one equation with $[1, 1, \dots, 1]\vec{\pi} = 1$
3. use GTH algorithm (less roundoff error, but slower)
4. iterative methods if CTMC is huge: Gauss-Seidel, SOR, GMRES, etc.

Best for us: option 1. But, can get bad results based on which $\pi_n = 1$.

Put it all together

```
function [Qmat, helpers] = mmsr_retrial_mat(lambda,etc)
nstates = told(sys_cap,orbit_cap,sys_cap,orbit_cap);
Qmat = sparse(nstates,nstates);
% 1. fill in lambdas, mus, etc.
% 2. set diagonal so row sums = 0
% 3. do helpers, as above
% 4. maybe make a state map to help with debugging
return;
function statenum=toold(ns,no,sys_cap,orbit_cap)
statenum = no * (sys_cap + 1) + ns + 1;
return;
```

The “toold” function is inside the `mmsr_retrial_mat.m` file; it is only visible to that function.

Put it all together, part 2

```
function perf = mmsr_retrial(lambda, etc.)
[Qmat, helpers] = mmsr_retrial_mat(lambda, etc.);
prvec = ctmc_stationary_probs_d(Qmat);
perf.prvec = prvec;
perf.L = prvec * helpers.L; % etc.
return;
```

ODEs and CTMCs

Solve the Kolmogorov eqns, $\vec{p}'(t) \equiv \frac{d}{dt}\vec{p}(t) = \vec{p}(t) \cdot Q(t)$

Numerical solvers will pick values of t and $\vec{p}(t)$, will want us to compute $\vec{p}'(t)$

Basic solver: Runge-Kutta of order 4-5, matlab builtin ode45()

do_simple.m:

```
y0 = [1 0 0]'; matlab ode stuff wants column vec.  
tmax = 24;  
[Tvec,Ymat] = ode45(@simple_odefun,[0 tmax],y0);  
% row vector Y(n,:) = prvec( at time T(n) )  
Lt = Ymat * [0 1 2]'; % [0 1 2] is a helper vec.
```

simple_odefun.m:

```
function yprime = simple_odefun(tval,yvec)  
lambda=1.5;  
mu = 1;  
Qmat = [ -lambda,    lambda    , 0        ; ...  
          mu    , -(lambda+mu), lambda    ; ...  
          0      , 2*mu      , -2*mu    ];  
yprime = Qmat' * yvec;  
return;
```

Complications

- ▶ The “odefun” will need lambda, mu, etc.
- ▶ Need to generate new Qmat for each time value.
- ▶ Forming a large matrix is slow, and we'll need to change $\lambda(t)$ at each time step.
- ▶ Initial conditions
- ▶ Stiff systems

Passing system info to odefun

do_simple.m:

```
y0 = [1 0 0]'; matlab ode stuff wants column vec.  
tmax = 24;  
odeopts = []; % use defaults  
[T,Y] =ode45(@simple_odefun,[0 tmax],y0,odeopts,sysinfo);
```

simple_odefun.m:

```
function yprime = simple_odefun(tval,yvec,sysinfo)  
lambda = sysinfo.lambda;  
mu      = sysinfo.mu;  
Qmat = [ -lambda,    lambda    , 0        ; ...  
          mu     , -(lambda+mu), lambda   ; ...  
          0      , 2*mu      , -2*mu    ];  
yprime = Qmat' * yvec;  
return;
```

Dealing with changing rates

Instead of forming

$$Q = \begin{bmatrix} -\lambda & \lambda & 0 \\ \mu & -(\lambda + \mu) & \lambda \\ 0 & 2\mu & -2\mu \end{bmatrix}$$

We will do

$$Q(t) = \lambda(t) \begin{bmatrix} -1 & 1 & 0 \\ 0 & -1 & 1 \\ 0 & 0 & 0 \end{bmatrix} + \mu(t) \begin{bmatrix} 0 & 0 & 0 \\ 1 & -1 & 0 \\ 0 & 2 & -2 \end{bmatrix}$$

```

simple_odefun.m:
function yprime = simple_odefun(tval,yvec,sysinfo)
lambda = sysinfo.lambda;
mu      = sysinfo.mu;
mats.lam = [-1,  1,  0; ...
            0, -1,  1; ...
            0,  0,  0];

mats.mu   = [ 0,  0,  0; ...
            1, -1,  0; ...
            0,  2, -2];

Qmat = mats.lam * lambda + mats.mu * mu;
yprime = Qmat' * yvec;
return;

```

But, we know from our steady-state CTMC stuff that forming the matrices is often difficult (2-dim, etc.)

So, we will break things up:

```
do_phase.m
  ode45.m (or ode15s, later)
    mmsr_retrial_odefun.m
      mmsr_retrial_mats.m
        told
```

The odefun will call the “mats” function once at the start, rather than over and over again.

```
function yprime = mmsr_retrial_odefun(tvals,yvec,sysinfo)
persistent mats;
if( isempty(mats) )
    mats = mmsr_retrial_mats(sysinfo);
end

% 1. Get current lambda, mu, etc.
%   (several ways to do this, not sure which is best.)
% 2. Assemble Q matrix
Qmat = mats.lam * lambda + ...
       mats.lam1 * lambda * pr_retry1 + ...
% plus retrials, etc.
       mats.mu * mu;
yprime = Qmat' * yvec(:);
```

If you change the system size, need to do
clear mmsr_retrial_odefun to tell it to get new matrices.

```
function [mats, helpers] = mmsr_retrial_mats(sysinfo);
% Note: doesn't care what lambda, mu are!
% ordinary accepted arrivals:
lam = sparse(nstates,nstates);
for ns=0:(sys_cap-1)
    for no=0:orbit_cap
        from = told(ns ,no,sys_cap,orbit_cap);
        to   = told(ns+1,no,sys_cap,orbit_cap);
        lam(from,to) = 1;
        lam(from,from) = -1;
    end % for no
end % for ns
mats.lam = lam; % etc. for mu, retrials, etc.

% then, make helpers as before
```

Calculating rates Version 1

Pass parameters, call a specific function

```
(in driver routine:)  
sysinfo.arr.avg = 10;  
sysinfo.arr.freq = 2*pi/24;  
sysinfo.arr.RelAmp = .8;  
sysinfo.arr.phase = 0;
```

```
(then, inside odefun:)  
lambda = ratet(tval, sysinfo.arr);  
mu      = ratet(tval, sysinfo.svc);
```

```
ratet.m:  
function r = ratet(tval, ratestruct)  
r = ratestruct.avg*(1-ratestruct.RelAmp* ...  
    cos(ratestruct.freq*tval + ratestruct.phase));  
return
```

Calculating rates Version 2

Using anonymous function handles

(in driver routine)

```
arr.avg = 10;
```

```
arr.freq = 2*pi/24;
```

```
arr.RelAmp = .8;
```

```
arr.phase = 0;
```

```
sysinfo.arrfunc = ...
```

```
    @(t) arr.avg*(1-arr.RelAmp*cos(arr.freq*t + arr.phase));
```

(then, inside odefun:)

```
lambda = sysinfo.arrfunc(tval);
```

```
mu      = sysinfo.svcfunc(tval);
```

Initial Conditions

Driver routine needs y_0 to give ode45

The y_0 vector depends on the state-space mapping, which we try hard to ignore.

Solution: driver routine will ask "odefun" for a y_0 .

How to ask? Don't give odefun a "yvec".

```
function yprime = mmsr_retrial_odefun(tvals,yvec,sysinfo)
persistent mats;
if( isempty(mats) )
    mats = mmsr_retrial_mats(sysinfo);
end
if( isempty(yvec) ) % a request for a y0 vector.
    % make a y0 vector the same size as the state space
    nstates = size(mats.lam); % or any other matrix
    y0 = zeros(1,nstates);
    y0(1) = 1; % empty system (depends on statemapping)
    yprime = y0; % it's not really yprime, though
    return;
end
```

Could get more advanced, have options for

- ▶ Full system
- ▶ Full servers, no queue
- ▶ Pointwise Stationary Approx
- ▶ user-specified

Just like simulation (IE 404), need to “warm up” from initial conditions.

Often, lull occurs around midnight (hour 0), don't much care about performance then.

Options:

- ▶ Integrate on $[0,48]$ hours but only look at $[24,48]$ hours.
- ▶ Integrate on $[0,24]$ with PSA, empty, full; then compare to show that they're close. Then, use PSA answer.
- ▶ Start with PSA during lull, figure it's good enough.

Stiff systems

Many systems are “stiff”, need a stiff solver like ode15s()

Stiff: some large rates (like 100μ), some small (like 1μ)

Stiff solvers can take much larger time steps than non-stiff solvers.

Warning: once, I had an arrival rate that changed for just 2 hours.
The stiff solver stepped entirely over the change!

Can set a maximum step size.

Stiff solvers benefit from knowing the Jacobian

For our ODE $\vec{p}'(t) \equiv \frac{d}{dt}\vec{p}(t) = \vec{p}(t) \cdot Q(t)$, the Jacobian at time t is just $Q(t)$

Driver changes:

```
odeopts=[]; % defaults
odeopts = odeset(odeopts,'Jacobian','mmsr_retrial_jac');
odeopts = odeset(odeopts,'Vectorized','on');
% Vectorized: yvec may be a collection of vectors
[T,Y] =ode15s(@simple_odefun,[0 tmax],y0,odeopts,sysinfo);
```

odefun changes:

```
function [yprime, Qmat] = mmsr_retrial_odefun ...
    (tvals,yvec,sysinfo)
```

New function to give the Jacobian (very simple!)

```
function [jac] = mmsr_retrial_jac(tvals,yvec,sysinfo)
[discard,Qmat] = handoff0_odefun(t,y,sysinfo);
jac = Qmat'; % that weird row vs. column vector thing.
return
```

Putting it All Together

See the zip file that goes along with these slides for an example.
Read the “readme.txt” file for a brief description of what it contains.

Send me any questions you have via e-mail.