Objective

To provide an introduction to the exciting and frustrating world of MATLAB.

Overview

The ancients created a programming environment to manipulate vectors and matrices. They (Mathworks) named it MATLAB which stands for MATrix LABoratory not to be confused with MATLAV (MATrix LAVatory) a program for flushing matrices down the toilet. MATLAB is a sophisticated numerical software package in the same vein as its competitor Mathematica. It includes advanced numerical algorithms and highly develop visualization tools which makes it very useful in solving and simulating complex systems of equations. In addition, it uses a high-level programming language that allows users to quickly code and manipulate data.

Why use MATLAB

Matrices. Anything that is a vector or matrix is in the wheelhouse of MATLAB. With its built-in graphics capabilities, it is very easy to quickly analyze data and generate results. Other packages often require a bit more effort to manipulate data and visualize it, and many more have copied MATLAB's approach and style.

Common. MATLAB is widely used and thus enjoys a large community of support. The language is also copied by others like Octave and Julia which means that code can easily be converted or shared.

Simulations. Many simulation techniques make use of vector and matrix formulations and so MATLAB is very useful in large-scale simulations.

Ease. The high-level programming language of MATLAB means it is easy to code up an algorithm and fairly transparent to read/understand. There is also little overhead so variables can be defined and modified on a whim, without prior declarations. No need to type "x = Int64[];".

Debugging. MATLAB has two awesome features that assist in debugging and refining code. One are break points which temporarily stop code (even in functions) and allow the user to check the state of various variables. The other is Profiler which computes the time and number of calls of each segment of code within a program. Thus, it is easy to identify what is slowing down a program or what line is being called way too often.

Error messages. Perhaps this is person-specific but the error messages in MATLAB seem much clearer and easier to fix than certain other coding languages.

Why not to use MATLAB

Cost. MATLAB is a commercial product and as such costs money. It is cheaper for students but still has a price tag that many find prohibitive. Adding insult to injury, it typically updates twice a year and these updates must be purchased separately. Fortunately, the basic language is the same but the performance can change.

Updates. As mentioned above, MATLAB frequently updates. Although this results in an overall improvement in performance, it can change the performance of user-written code between releases. It is not uncommon for codes to run slower and require reformatting to run faster– i.e. coding techniques may need to change with different releases.

Text processing. While MATLAB can handle strings and manipulate them, it is not its strong suit. MATLAB is quantitative and prefers to manipulate numbers. Languages like python and perl are better options for this task.

Speed. MATLAB is faster than most languages but it is not the fastest. There is a tradeoff to the user-friendly aspects of MATLAB and performance speed. For example, Julia code can be much faster than MATLAB (one pieces of code was 6X faster) but it can also take longer to optimize Julia code.

Window Structure

MATLAB uses a collection of windows as part of its user interface. Each window contains different types of information and can be removed or highlighted based on user desires.

1. *Current Folder*: shows the contents of the current directory MATLAB is accessing.
2. *Command Window*: where most of the commands will be typed. It should probably be the most prominent window.
3. *Workspace*: lists the variables currently defined and available.
4. *Command History* keeps track of the previous commands typed into the command window.

They can be configured to individual preferences by messing around with the preferences section. Basically, the Command Window and Workspace windows are the most useful/important for regular programming.

Basics of the language

MATLAB is matrix oriented which means that if prefers data in matrix form. A list of numbers, array, vector, etc are all stored as a matrix of one row and many columns (or vice versa depending on how it was entered). So if we want to call a certain list of numbers, "vect1", we type:

```
>> vect1=[1 21 3 .7 -3 0]

vect1 =

    1.0000   21.0000    3.0000    0.7000   -3.0000         0

>>
```

Alternatively, if we did not want to write every element of a vector but just wanted an interval starting at some number a increasing by δt and ending when b is reached, we would write `vect2=[a:`δt`:b]`. For instance:

```
>> vect2=[1:.2:2]

vect2 =

    1.0000    1.2000    1.4000    1.6000    1.8000    2.0000

>> vect2=[1:.3:2]

vect2 =

    1.0000    1.3000    1.6000    1.9000

>> vect2=[2:-.3:1]

vect2 =

    2.0000    1.7000    1.4000    1.1000

>>
```

If we do not want MATLAB displaying the data then we simply put a semicolon at the end of the line. Any line without a semicolon, is effectively asking MATLAB to print out the result of the code. This can be an eyesore and for certain codes can slow down performance. Semicolons tell MATLAB, “Shut up.”

```
>> vect1=[1 21 3 .7 -3 0];
>>
```

To access a certain element, say the third, from `vect1`, we simply type:

```
>> vect1(3)

ans =

     3

>>
```

Note that MATLAB begins counting with 1 which means `vect1(0)` would return an error. Furthermore, the results are stored as `ans` which means if we type `ans+4` we get `7`. The Workspace window should also show the existence of a variable named `ans`. To get the size (dimensionality) of `vect1`, we have two options. The first is using the built-in function `size` and the second is the built-in function `length`.

```
>> size(vect1)

ans =

     1     6

>> size(vect1,1)

ans =

     1

>> size(vect1,2)

ans =

     6

>> length(vect1)

ans =

     6

>>
```

The function `size` displays the number of rows and then columns. It takes in one mandatory input, the item of which the size is requested. The second input, separated by a comma, is optional. It indicates whether one wants the number of rows (`size( ,1)`) or columns (`size( ,2)`). The function `length` reports back the biggest of the two dimensions, either row or column number. Dimensions is particularly important in MATLAB and the cause of most errors. To make notation easier, we will say that a matrix with dimensions Z x Y (Z "by" Y) will have Z rows and Y columns (just like `size` reports). Using linear algebra, anything multiplied by this Z x Y must have Y as its first dimension. Thus, if a vector is to be multiplied by this matrix it must be Y x 1 and will return a vector of Z x 1, i.e. (Z x Y) * (Y x 1) = (Z x 1). The "inner dimensions" must agree, i.e.

(Z x **Y**) * (**Y** x 1). This linear algebra fact forms the basis of MATLAB manipulations. Let's try to square each element in `vect1`.

```
>> vect1*vect1
Error using * Inner matrix dimensions must agree.

>>
```

This command did not work because the inner dimensions do not agree (1 x **Y**) * (**1** x Y), Y$\neq$ 1. One way to make it work is to transpose the second `vect1` by adding a single apostrophe after it `vect1'`. This changes it from a row vector (1 x Y) to a column vector (Y X 1) and gives (1 x **Y**) * (**Y** x 1 = 1 x 1. In other words, this is the dot product. Alternatively, we could transpose the first `vect1` and get...?

```
>> vect1*(vect1')

ans =

  460.4900

>>
```

To actually get a vector where each entry is the square of the corresponding entry in `vect1`, we type:

```
>> vect1.*vect1

ans =

    1.0000  441.0000    9.0000    0.4900    9.0000         0

>>
```

Any time a "." appears before an operation it means "element-wise", ignoring linear algebra rules. So we can also use "." with the exponent notation "^".

```
>> vect1.^2
```

```
ans =

    1.0000  441.0000    9.0000    0.4900    9.0000         0

>>
```

Here, are some other common commands with vectors in MATLAB.

```
>> sum(vect1)

ans =

   22.7000

>> min(vect1)

ans =

    -3

>> max(vect1)

ans =

    21

>>
```

We could try taking the average of `vect1`.

```
>> average(vect1)
Undefined function 'average' for input arguments of type 'double'.

>>
```

This error indicates that `average` is a function which does not exist in MATLAB. Is there another way of doing it based on the functions presented so far? We can check to see if another function exists using `help`.

```
>> help mean
 mean   Average or mean value.
    S = mean(X) is the mean value of the elements in X
    if X is a vector. For matrices, S is a row
    vector containing the mean value of each column.
    For N-D arrays, S is the mean value of the
    elements along the first array dimension whose size
    does not equal 1.
    If X is floating point, that is double or single,
    S has the same class as X. If X is of integer
    class, S has class double.

    mean(X,DIM) takes the mean along the dimension DIM
    of X.

    S = mean(X,'double') and S = mean(X,DIM,'double')
    returns S in double, even if X is single.

    S = mean(X,'native') and S = mean(X,DIM,'native')
    returns S in the same class as X.

    S = mean(X,'default') and S = mean(X,DIM,'default')
    are equivalent to S = mean(X) and S = mean(X,DIM)
    respectively.

    Example: If X = [1 2 3; 3 3 6; 4 6 8; 4 7 7];

    then mean(X,1) is [3.0000 4.5000 6.0000] and
    mean(X,2) is [2.0000 4.0000 6.0000 6.0000].'

    Class support for input X:
       float: double, single
       integer: uint8, int8, uint16, int16, uint32,
                int32, uint64, int64

    See also median, std, min, max, var, cov, mode.

    Overloaded methods:
       fints/mean
       ProbDistUnivParam/mean
       timeseries/mean

    Reference page in Help browser
       doc mean

>> mean(vect1)

ans =

    3.7833
```

```
>>
```

The `help` feature gives important information about a function including its inputs, optional arguments, and returns. One incredibly common return is from `max` in which not only does it return the maximum value but it returns the location. This is particularly useful when one is concerned with retrieving data or manipulating it.

```
>> [u,v]=max(vect1)

u =

    21


v =

     2

>> vect1(v)

ans =

    21

>>
```

Moving on up in the world, we can create matrices in a few ways. One of which is similar to creating vectors in that we list a bunch of numbers and when we want the row to end, we add a semicolon. The other is using ready made vectors. We can also use built-in functions like `ones`,`zeros`,or `rand` to create matrices of all ones, all zeros, or random floats between 0 and 1, respectively.

```
>> mat1=[3 8 0;0 1 9;-3 4 .5]

mat1 =

    3.0000    8.0000         0
         0    1.0000    9.0000
   -3.0000    4.0000    0.5000

>> mat1=[vect1' vect1']
```

```
mat1 =

    1.0000    1.0000
   21.0000   21.0000
    3.0000    3.0000
    0.7000    0.7000
   -3.0000   -3.0000
         0         0



>> mat1=ones(3,3)

mat1 =

     1     1     1
     1     1     1
     1     1     1

>> mat1=ones(3)

mat1 =

     1     1     1
     1     1     1
     1     1     1

>> mat1=[zeros(1,3); zeros(1,3); zeros(1,3)]

mat1 =

     0     0     0
     0     0     0
     0     0     0

>> mat1=rand(3)

mat1 =

    0.8147    0.9134    0.2785
    0.9058    0.6324    0.5469
    0.1270    0.0975    0.9575

>>
```

Accessing elements of matrices is similar to vectors, we use a comma to separate row from column.

```
>> mat1(1,2)

ans =

    0.9134

>> mat1(2,3)

ans =

    0.5469

>> mat1(3,4)
Index exceeds matrix dimensions.
```

The error message came up because `mat1` only has 3 columns not 4. As with vectors if we want to square all entries one by one in a matrix we use `.^2`. However, with square matrices we need to be careful because `mat1*mat1` will return an answer without an error message.

```
>> mat1.^2

ans =

    0.6637    0.8343    0.0776
    0.8205    0.3999    0.2991
    0.0161    0.0095    0.9168

>> mat1*mat1

ans =

    1.5265    1.3489    0.9931
    1.3802    1.2806    1.1218
    0.3134    0.2710    1.0055

>> mat1^2

ans =

    1.5265    1.3489    0.9931
    1.3802    1.2806    1.1218
    0.3134    0.2710    1.0055

>>
```

Loading and saving data

One incredibly nice feature of MATLAB is that there are very easy commands for data management. Functions `csvread` and `csvwrite` allow the import and export of data. Alternatively `save` and `load` all of the saving and loading of previous workspace variables. For example, imagine that there is a csv file called "csvexample.csv". We can import its values and save it something called `d`. We can then create some new variables and then save all of the workspace variables in a file called "allofmystuff.mat" (Note: .mat extension is not necessary to add to the filename using save but will appear automatically in the current working directory). We can clear everything and then use the `load` command to get our data back.

```
>> d=csvread('csvexample.csv')

d =

     4     5     6     7     6     5     4
     3     2     3     4     5     6     7
     1     9     8     4     2     3     4
     3     6     7     4     2     2     1

>> f=mean(d);
>> g=d;
>> g(g>5)=5;
>> save allofmystuff
>> clear
>> load allofmystuff
>>
```

Control flow

MATLAB has the standard assortment of `for` loops, `while` loops, and `if, else` statements. In this section, we present some examples. It should be stated that unlike the programming language python, MATLAB does not care about tabs or spaces. Thus, there is no need to indent apart from readability. Also, semicolons tell MATLAB when the end of a statement is reached which means multiple lines of code can be put on one line with semicolons.

```
>> x=[0:.1:1];
for i1=1:11
x(i1)=2*x(i1);
end
>> x
```

```
x =

  Columns 1 through 6

         0    0.2000    0.4000    0.6000    0.8000    1.0000

  Columns 7 through 11

    1.2000    1.4000    1.6000    1.8000    2.0000

>> for i1=1:11;x(i1)=2*x(i1);end
>> x

x =

  Columns 1 through 6

         0    0.4000    0.8000    1.2000    1.6000    2.0000

  Columns 7 through 11

    2.4000    2.8000    3.2000    3.6000    4.0000

>>
```

The ubiquitous `if else` statements follow a simple structure of: `if` expression, commands, optional `elseif` expression, commands, optional catch all `else` expression, commands, and `end`.

```
>> x=1;
if (x>5)
x=2*x;
elseif (x>2)
x=3*x;
else
x=4*x;
end
>> x

x =

     4

>> if (x>5)
x=2*x;
elseif (x>2)
```

```
x=3*x;
else
x=4*x;
end
>> x

x =

    12

>> if (x>5)
x=2*x;
elseif (x>2)
x=3*x;
else
x=4*x;
end
>> x

x =

    24

>>
```

In true/false evaluated expressions true is the same as the number 1 and 0 represents false. The logical AND is represented by `&` and the OR is represented by —. What is the difference between `&` and `&&`?

```
>> (x>2) & (x<6)

ans =

     1

>> (x>2) & (x>6)

ans =

     0

>> (x>2) | (x>6)

ans =

     1
```

```
>>
```

Using this we can write a `while` loop that adds numbers until either a certain total is reached or the maximum number of iterations is reached.

```
>> num=0;
total=10;
maxnumiter=25;
iter=0;
while (num<total) & (iter<maxnumiter)
iter=iter+1;
num=num+rand(); % this adds a random number between 0 and 1
% oh yeah, this is a comment
% MATLAB has no idea what we are typing
% Sometimes you test my patience, MATLAB
% But I still love you
end
[iter num]

ans =

   25.0000    9.8741

>> num=0;
total=10;
maxnumiter=25;
iter=0;
while (num<total) & (iter<maxnumiter)
iter=iter+1;
num=num+rand(); % this adds a random number between 0 and 1
% oh yeah, this is a comment
% MATLAB has no idea what we are typing
% Sometimes you test my patience, MATLAB
% But I still love you
end
[iter num]

ans =

   19.0000   10.2952

>>
```

Finally, we mention that there is a control flow that is not as common as the other types which is `case`,`switch`,`end`. I would recommend against using this if the code is ever to be put elsewhere but it is incredibly helpful at times within MATLAB coding.

```
>> x=1;
switch x
case 1
x=2*x;
case 2
x='this is a string';
case 'this is not a string'
x=3*x;
end
>> x

x =

     2

>> switch x
case 1
x=2*x;
case 2
x='this is a string';
case 'this is not a string'
x=3*x;
end
>> x

x =

this is a string

>> switch x
case 1
x=2*x;
case 2
x='this is a string';
case 'this is not a string'
x=3*x;
end
>> x

x =

this is a string

>>
```

For more information about any of these, use the `help` feature or similarly use `doc` which brings up a window of MATLAB documentation. So typing `doc case` brings up helpful information about `case`. If your code is risky and takes chances which may result in errors `doc try` is something worthy reading.

Visualizing data, plotting

```
>> x=[.01:.01:10];
>> y=cos(x);
>> plot(x,y);
```

This will result in a window popping up that looks like Figure 7. You can save the Figure to a file with the following command: "`>> print -f1 -dpdf -r300 figcos.pdf`". This will print figure window 1 (`-f1`) to a pdf (`-dpdf`) at a resolution of 300 dpi (`-r300`) with filename "figcos.pdf". If, instead, we had wanted to save some second figure window to an eps file with color in the folder above our current working directory, we would type something like "`>> print -f2 -depsc -r300 ../figcos.eps`".

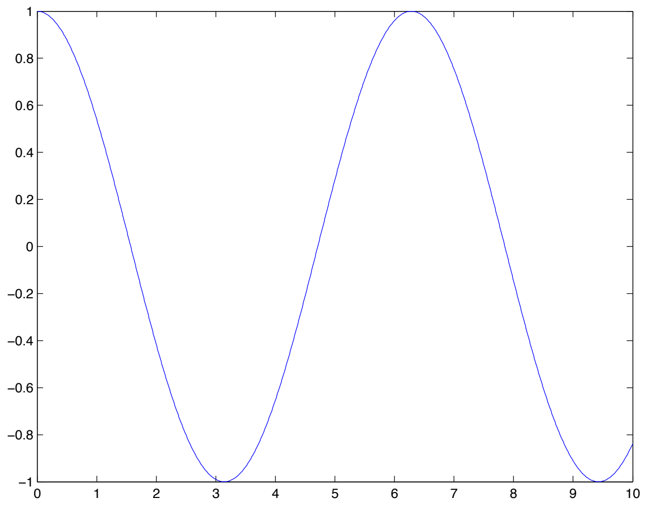


Figure 1. figcos.pdf

For figures that look a bit nicer, there are all sorts of options to change. The `hold on` code allows multiple plots to be put on one figure window without erasing (`hold off` undoes this). Otherwise, MATLAB will overwrite the existing figure. There are also features regarding labels, colors, and line styles. See example below:

```
x=[.01:.01:10];
y=cos(x);
plot(x,y,'--','Color',[1 .5 0],'LineWidth',5);
hold on;
plot(x,.5*sin(2*x),':','Color',[0 .5 0],'LineWidth',3);
set(gca,'TickLength',[.025 .025],'LineWidth',3)
% change features of the graph
xlabel('This is the horizontal axis')
ylabel('This is the vertical axis','FontSize',14)
```

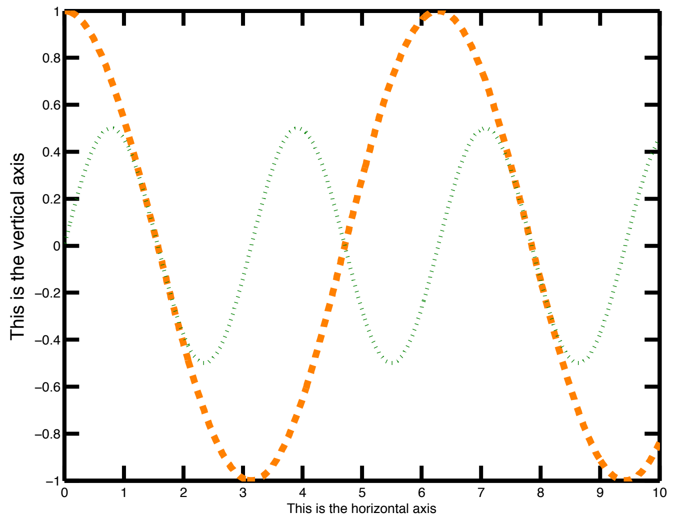


FIGURE 2. figcosadv.pdf

Another type of plot is in log scale which can be done in MATLAB using functions like `semilogx`.

```
close; %closes current figure
figure; % opens new figure window
x=10.^(6*rand(100,1));
y=log10(x); %log alone is ln
semilogx(x,y,'o');
axis([1 10^6 0 7]); % defines the zoom axis for plot
```

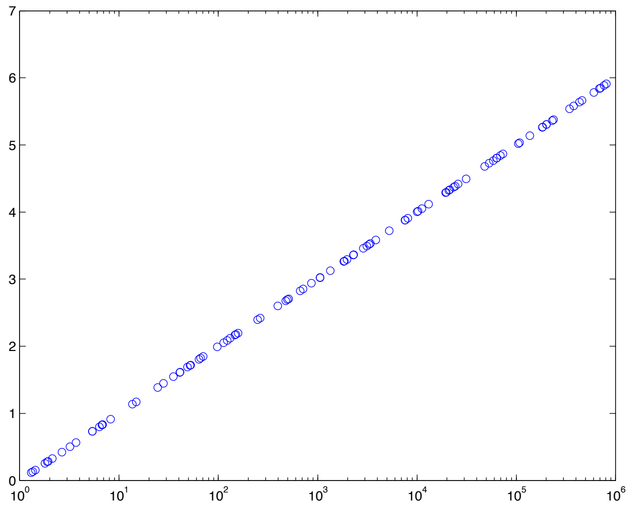


FIGURE 3. semilogplot.pdf

MATLAB has other built in functions for creating standard plots like bar graphs and histograms.

```
x=randn(1000,1); % samples from a normal distribution
hist(x,[-4:.25:4]); % user defined bins
```

SCRIPTS AND FUNCTIONS

Typing `edit` brings up a window where a script or function can be typed. If saved, it will save in the current working directory (displayed as a window in MATLAB). A script is basically just a sequence of commands. It uses the same variables present in the current workspace. Thus, a script may have no reference to some variable `x` but if `x` is defined in the workspace it will run. In contrast, functions only use the variables specifically passed to them. MATLAB is unlike some other programming languages in that variables passed to functions cannot change their values unless specifically returned by the function. For example, let's write a simple function called "countones" which will take in a vector and count all of the ones in it. It will return the total number of ones and just for giggles will change the values of ones to twos.

```
function [otpt1,otpt2] = countones(vect)
% all functions begin with the word function
```

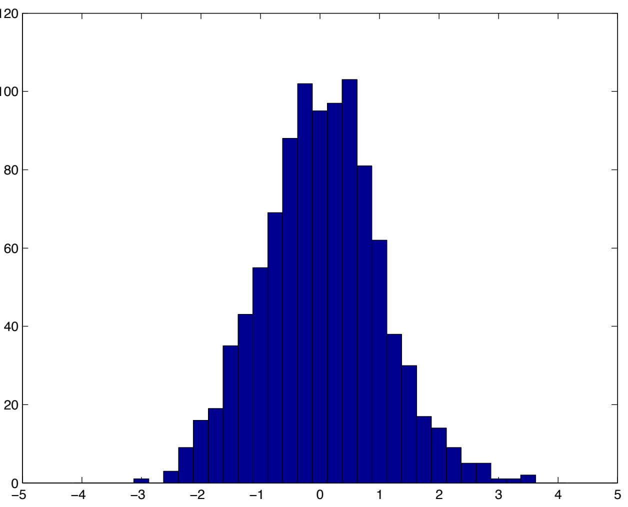


FIGURE 4. histplot.pdf

```
% [otpt1,otpt2] indicates two outputs will be defined
% countones is the name of the function
% vect is function's name for the input
k=find(vect==1); % find returns a list of indices
otpt1=length(k); % counts the number of indices
vect(k)=2;
otpt2=vect;
```

We call this function in the command window and test it out. Notice that the vector z is unscathed through various calls of the function. The function world and command space world are completely separate.

```
>> z=ones(10,1); % vector of ones
>> z(rand(10,1)<.5)=3; % randomly change about 1/2 to 3
>> countones(z)

ans =

     5

>> z

z =

     3
     1
     1
     3
     3
     1
     3
     3
     1
     1

>> a=countones(z)

a =

     5

>> z

z =

     3
     1
     1
     3
     3
     1
     3
     3
     1
     1

>> a,b=countones(z) % this fails because we need brackets

a =

     5


b =

     5

>> [a,b]=countones(z) % this works

a =

     5


b =
       3
       2
       2
       3
       3
       2
       3
       3
       2
       2

>> z

z =

       3
       1
       1
       3
       3
       1
       3
       3
       1
       1

>>
```

An example with ferns

Here, we show an example of MATLAB code designed to generate a Barnsley fern. We begin with a point in the x,y coordinate system. We then apply 1 of 4 transformations to this point. We plot that point and then choose another transformation. While these transformations are chosen randomly they are not equally likely. The details are below:

(1) Transformation 1 chosen 1% of the time:

$$\begin{aligned} x_{t+1} &= 0x_t + 0y_t + 0 \\ y_{t+1} &= 0x_t + 0.16y_t + 0 \end{aligned}$$

(2) Transformation 2 chosen 85% of the time:

$$\begin{aligned} x_{t+1} &= 0.85x_t + 0.04y_t + 0 \\ y_{t+1} &= -0.04x_t + 0.85y_t + 1.6 \end{aligned}$$

(3) Transformation 3 chosen 7% of the time:

$$\begin{aligned} x_{t+1} &= 0.20x_t + -0.26y_t + 0 \\ y_{t+1} &= 0.23x_t + 0.22y_t + 1.6 \end{aligned}$$

(4) Transformation 4 chosen 7% of the time:

$$\begin{aligned} x_{t+1} &= -0.15x_t + 0.28y_t + 0 \\ y_{t+1} &= 0.26x_t + 0.24y_t + 0.44 \end{aligned}$$

One code this that works is:

```
% Barnsley_fern
close all; % close any current graphs
mat1=[0 0;  0   0.16];
vect1=[0    0]';

mat2=[0.85  0.04    ;-0.04  0.85];
vect2=[0    1.6]';

mat3=[0.2   -0.26   ;0.23   0.22];
vect3=[0    1.6]';

mat4=[-0.15 0.28; 0.26 0.24];
vect4=[ 0   0.44]';

p=[.01 .85 .07 .07]'; % probabilities
pt=[0;0]; % starting point
numofpts=50000; % number of points
figure;hold on;
for i=2:numofpts
    temp=rand();
    if temp<p(1)
        pt=mat1*pt+vect1;
    elseif (temp>=p(1)) & (temp<p(1)+p(2))
        pt=mat2*pt+vect2;
    elseif (temp>=p(1)+p(2)) & (temp<p(1)+p(2)+p(3))
        pt=mat3*pt+vect3;
    else
        pt=mat4*pt+vect4;
    end
    plot(pt(1),pt(2),'.','Color',[0 .5 0]);
end
```

The results of this script saved as “barnslow.n” are shown below.

This code, however, is quite slow. We can run and time it using MATLAB’s profiler to find out what is slowing down the code. The trouble seems to be with plotting every

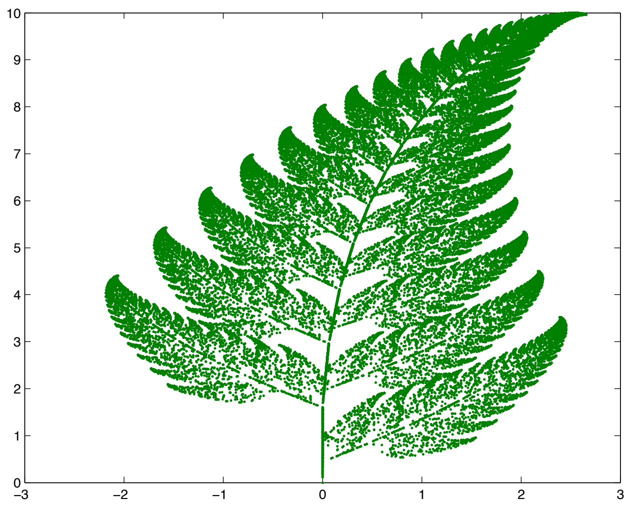


FIGURE 5. fern.pdf

iteration. It is costly to keep adding points to the figure. One way of speeding it up would be to remove this issue.

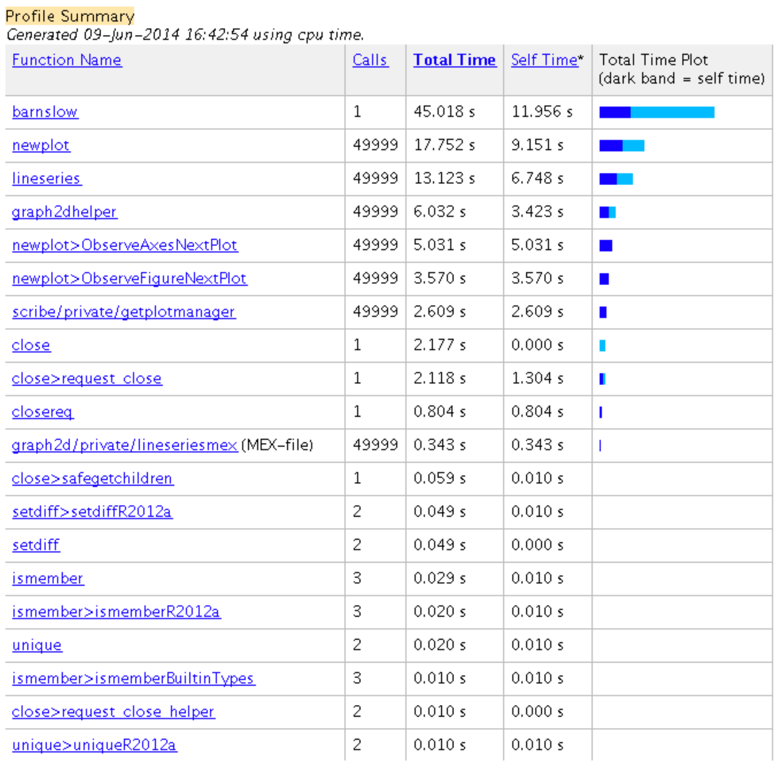
```
% Barnsley_fern
close all;
mat1=[0 0;  0   0.16];
vect1=[0    0]';

mat2=[0.85  0.04    ;-0.04  0.85];
vect2=[0    1.6]';

mat3=[0.2   -0.26   ;0.23   0.22];
vect3=[0    1.6]';

mat4=[-0.15 0.28; 0.26 0.24];
vect4=[ 0   0.44]';

p=[.01 .85 .07 .07]'; % probabilities
pt=[0;0]; % starting point
numofpts=50000; % number of points
figure;hold on;
ptmat=zeros(numofpts,2);
for i=2:numofpts
    temp=rand();
    if temp<p(1)
        pt=mat1*pt+vect1;
    elseif (temp>=p(1)) & (temp<p(1)+p(2))
```

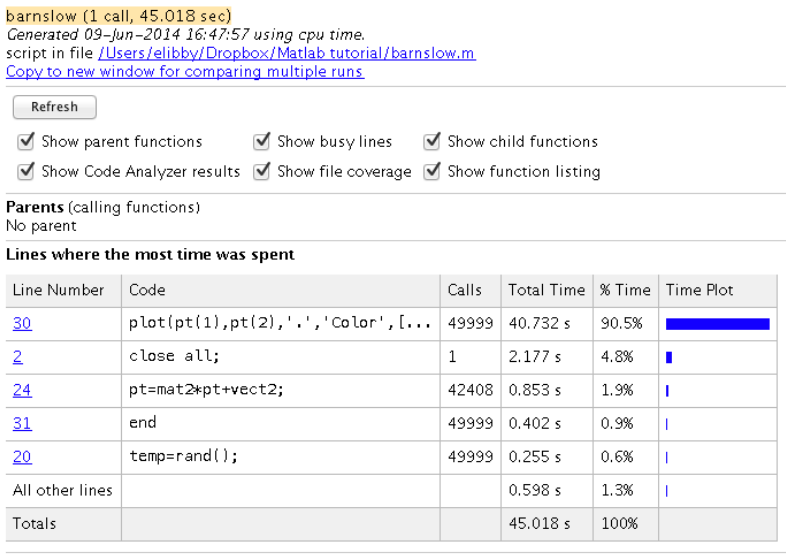


Profile Summary
Generated 09-Jun-2014 16:42:54 using cpu time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
barnslow	1	45.018 s	11.956 s	
newplot	49999	17.752 s	9.151 s	
lineseries	49999	13.123 s	6.748 s	
graph2dhelper	49999	6.032 s	3.423 s	
newplot>ObserveAxesNextPlot	49999	5.031 s	5.031 s	
newplot>ObserveFigureNextPlot	49999	3.570 s	3.570 s	
scribe/private/getplotmanager	49999	2.609 s	2.609 s	
close	1	2.177 s	0.000 s	
close>request_close	1	2.118 s	1.304 s	
closereq	1	0.804 s	0.804 s	
graph2d/private/lineseriesmex (MEX-file)	49999	0.343 s	0.343 s	
close>safegetchildren	1	0.059 s	0.010 s	
setdiff>setdiffR2012a	2	0.049 s	0.010 s	
setdiff	2	0.049 s	0.000 s	
ismember	3	0.029 s	0.010 s	
ismember>ismemberR2012a	3	0.020 s	0.010 s	
unique	2	0.020 s	0.010 s	
ismember>ismemberBuiltinTypes	3	0.010 s	0.010 s	
close>request_close_helper	2	0.010 s	0.000 s	
unique>uniqueR2012a	2	0.010 s	0.010 s	

FIGURE 6. profiler.pdf

```
        pt=mat2*pt+vect2;
    elseif (temp>=p(1)+p(2)) & (temp<p(1)+p(2)+p(3))
        pt=mat3*pt+vect3;
    else
        pt=mat4*pt+vect4;
    end
    ptmat(i,:)=pt';
end
plot(ptmat(:,1),ptmat(:,2),'.','Color',[0 .5 0]);
```



barnslow (1 call, 45.018 sec)
Generated 09-Jun-2014 16:47:57 using cpu time.
script in file /Users/elibby/Dropbox/Matlab tutorial/barnslow.m
Copy to new window for comparing multiple runs

☑ Show parent functions ☑ Show busy lines ☑ Show child functions
☑ Show Code Analyzer results ☑ Show file coverage ☑ Show function listing

Parents (calling functions)
No parent

Lines where the most time was spent

Line Number	Code	Calls	Total Time	% Time	Time Plot
30	`plot(pt(1),pt(2),'.','Color',[...`	49999	40.732 s	90.5%	▬
2	`close all;`	1	2.177 s	4.8%	▮
24	`pt=mat2*pt+vect2;`	42408	0.853 s	1.9%	▮
31	`end`	49999	0.402 s	0.9%	▮
20	`temp=rand();`	49999	0.255 s	0.6%	▮
All other lines			0.598 s	1.3%	▮
Totals			45.018 s	100%	

FIGURE 7. profiler2.pdf

We can test the speed by running `tic()` and `toc()` which times commands.

```
>> tic();barnslow;toc()
Elapsed time is 24.680064 seconds.
>> tic();barnmed;toc()
Elapsed time is 0.318285 seconds.
```

Another way of coding that can show smaller gains in speed is shown below.

```
% Barnsley_fern
close all;
mat{1}=[0   0;  0   0.16]; % this is a cell array
vect{1}=[0  0]';
mat{2}=[0.85    0.04    ;-0.04  0.85];
vect{2}=[0  1.6]';
mat{3}=[0.2 -0.26   ;0.23   0.22];
vect{3}=[0  1.6]';
mat{4}=[-0.15   0.28; 0.26 0.24];
vect{4}=[   0   0.44]';
p=cumsum([0 .01 .85 .07 .07]'); % probabilities
numofpts=50000; % number of points
ptmat=zeros(2,numofpts);
inds=rand(numofpts,1);
inds(inds<p(2))=1;
inds(inds<p(3))=2;
inds(inds<p(4))=3;
inds(inds<p(5))=4;
for i=2:numofpts
    ptmat(:,i)=mat{inds(i)}*ptmat(:,i-1)+vect{inds(i)};
end
plot(ptmat(1,:),ptmat(2,:),'.','Color',[0 .5 0]);
```

An Example with ODE's

Let's imagine that we have a set of differential equations that we would like to solve.

$$\begin{aligned}\frac{dx}{dt} &= 0.750x - 0.100xy\\ \frac{dy}{dt} &= 0.050yx - 0.010y - 0.025yz\\ \frac{dz}{dt} &= 0.025zy - 0.010z\end{aligned}$$

MATLAB has several built-in functions for solving such a set of ordinary differential equations. For help look up `ode45`. The first and last parts are particularly important.

```
>> help ode45
 ode45  Solve non-stiff differential equations, medium order method.
    [TOUT,YOUT] = ode45(ODEFUN,TSPAN,Y0) with TSPAN = [T0 TFINAL]
    integrates the system of differential equations y' = f(t,y)
    from time T0 to TFINAL with initial conditions Y0.
    ODEFUN is a function handle. For a scalar T and  a vector Y,
    ODEFUN(T,Y) must return a column vector corresponding to f(t,y).
    Each row in the solution array YOUT corresponds to a time
    returned in the column vector TOUT.  To obtain solutions at
    specific times T0,T1,...,TFINAL (all increasing or all
    decreasing), use TSPAN = [T0 T1 ...  TFINAL].

    SKIP A BUNCH

       Example
         [t,y]=ode45(@vdp1,[0 20],[2 0]);
         plot(t,y(:,1));
      solves the system y' = vdp1(t,y), using the default relative
      error tolerance 1e-3 and the default absolute tolerance of
      1e-6 for each component, and plots the first component of
      the solution.

    Class support for inputs TSPAN, Y0, and the result of
    ODEFUN(T,Y): float: double, single

    See also ode23, ode113, ode15s, ode23s, ode23t, ode23tb,
             ode15i, odeset, odeplot, odephas2, odephas3, odeprint,
             deval, odeexamples, rigidode, ballode, orbitode,
             function_handle.

    Reference page in Help browser
       doc ode45

>>
```

So `ode45` requires a function that takes in a scalar time t and a vector Y and returns a column vector of the derivative of the vector y. We write a code for this set of differential equations and name it `sampdiff`.

```
function [vp]=sampdiff(t,v)
x=v(1);
y=v(2);
z=v(3);
dxdt=.75*x-.1*x*y;
dydt= .05*y*x-.01*y-.025*y*z;
dzdt= .025*z*y-.01*z;
vp=[dxdt ; dydt ; dzdt];
```

Note that this is a function which takes in a scalar time t but does not use it because the differentiation equations do not explicitly incorporate time. We can then call `ode45` and pass it the name of our function which is `sampdiff`. We solve the equations on the time interval 0 to 10 and with initial conditions $\{x(0) = 10, y(0) = 1, z(0) = .1\}$. The plot is in Figure 8

```
>> [t,y]=ode45(@sampdiff,[0 10],[10 1 .1]);
>> plot(t,y);
```

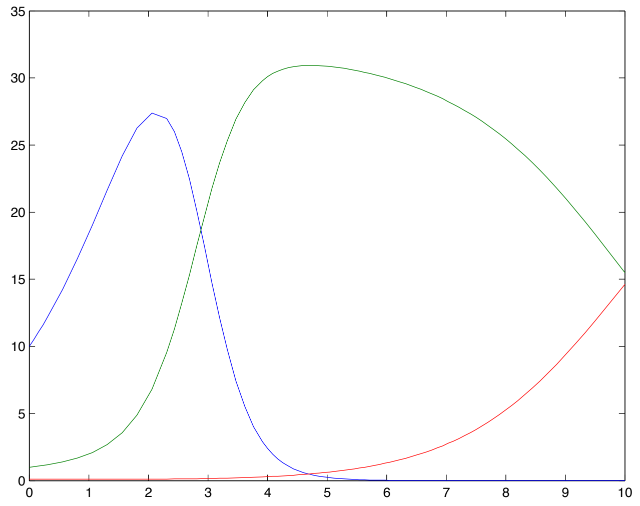


Figure 8. ode45.pdf

It seems more is going on so we extend the solution a couple of times to find the long-term dynamical behavior.

```
>> [t,y]=ode45(@sampdiff,[0 100],[10 1 .1]);
Warning: Failure at t=7.483691e+01.  Unable to meet
integration tolerances without reducing the step size
below the smallest value allowed (2.273737e-13) at time
t.
> In ode45 at 308
>> plot(t,y);
```

So the set of equations eventually explodes (unlimited exponential growth). We can zoom in to find out how this happens.

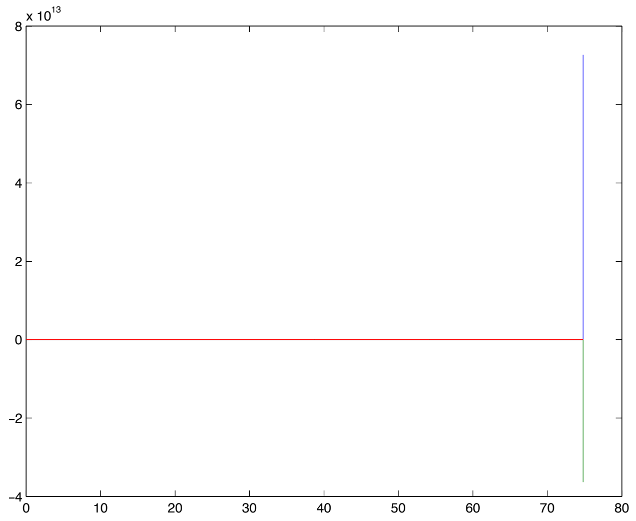


FIGURE 9. ode45b.pdf

```
>> [t,y]=ode45(@sampdiff,[0 65],[10 1 .1]);plot(t,y);
```

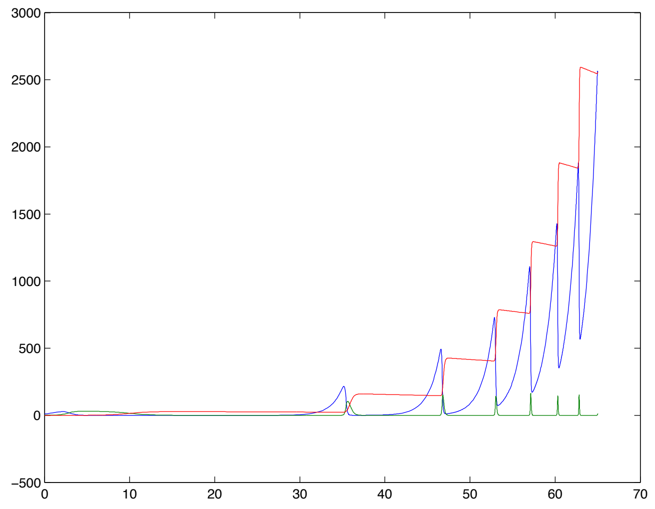


FIGURE 10. ode45c.pdf

Now let's try to stabilize these equations and keep them bounded. We allow an interaction between x and z.

$$\begin{aligned}\frac{dx}{dt} &= 0.750x - 0.100xy - 0.010xz\\ \frac{dy}{dt} &= 0.050yx - 0.010y - 0.025yz\\ \frac{dz}{dt} &= 0.025zy - 0.010z + 0.005zx\end{aligned}$$

```
function [vp]=sampdiff(t,v)
x=v(1);
y=v(2);
z=v(3);
dxdt=.75*x-.1*x*y-.01*x*z;
dydt= .05*y*x-.01*y-.025*y*z;
dzdt= .025*z*y-.01*z+.005*x*z;
vp=[dxdt ; dydt ; dzdt];
```

Try simulating these equations to find something that looks like Fig 11

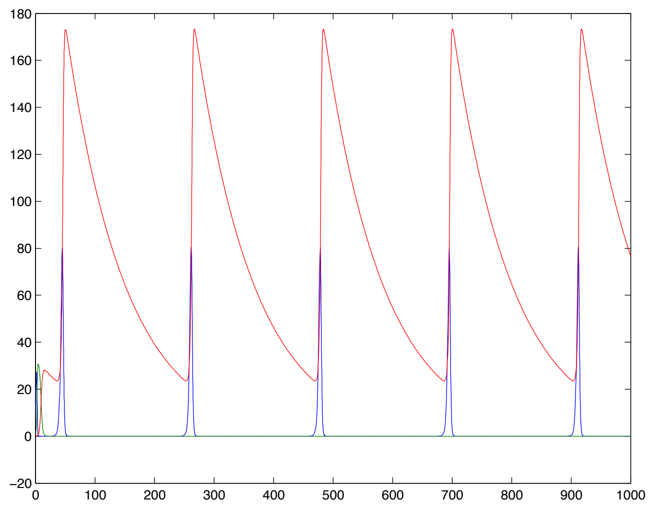


FIGURE 11. ode45rev.pdf

Vector field. We want to create a phase portrait for a grid of x and y and look at the derivative for each combination of x and y for a fixed z... say z=3. To do this, we want to use the built-in function `quiver`. Reading the help documentation, we need to create either a matrix of x and y values or two vectors that describe a uniform grid. This would require calling the `sampdiff` function many many times. Here is one example of how to do this:

```
>> close all % close any current figures
x=[1:1:20]';
y=[1:1:10]';
z=.005;
matx=zeros(length(y),length(x));
maty=zeros(length(y),length(x));
t=0; % t is arbitrary
for i1=1:length(x);
    for i2=1:length(y)
        temp=sampdiff(t,[x(i1);y(i2);z]);
        matx(i2,i1)=temp(1);
        maty(i2,i1)=temp(2);
    end
end
quiver(x,y,matx,maty,'LineWidth',2)
```

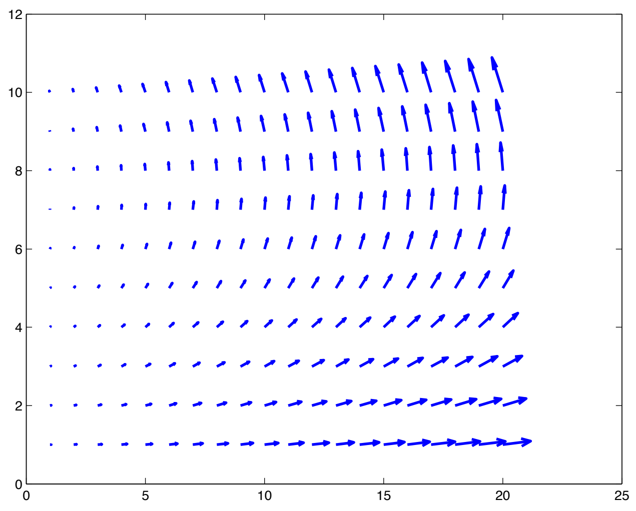


FIGURE 12. vectfield.pdf

Is there another, better way of doing this? Well, nested for loops can take up a lot of time especially if it is a bigger calculation. Another option is to vectorize the code of `sampdiff` so that it takes in a matrix where column 1 is x, column 2 is y, and column 3 is z and returns a matrix of the corresponding derivatives. That way we only need to call `sampdiff` once. We also rename it `samdiffv` to avoid confusion.

```
function [vp]=sampdiffv(t,v)
x=v(:,1);
y=v(:,2);
z=v(:,3);
dxdt=.75*x-.1*x.*y-.01*x.*z;
dydt= .05*y.*x-.01*y-.025*y.*z;
dzdt= .025*z.*y-.01*z+.005*x.*z;
vp=[dxdt dydt dzdt];
```

We then need to give it proper input which is a single matrix of x,y,and z which covers all possible combinations of x and y. This is an example of how to do this.

```
>> close all % close any current figures
x=[1:1:20];
y=[1:1:10];
z=.005;
t=0;
tot=length(x)*length(y);
[x2,y2]=meshgrid(x,y);
v=[reshape(x2,tot,1) reshape(y2,tot,1) z*ones(tot,1)];
vp=sampdiffv(t,v);
quiver(v(:,1),v(:,2),vp(:,1),vp(:,2),'LineWidth',2)
```

Coding Tips

Where the heck is my Figure? MATLAB will only show a Figure if it is newly created. Any additions to an existing Figure will not bring the window forward. Instead of hunting for the figure, simply type `shg` to *sh*ow *g*raph.

Faster typing. The **tab** key will autocomplete commands using variables from the Workspace and previous commands typed. This is particularly helpful if your commands or variable names are long and involved, "all_of_the_stuff_I_ever_needed_is_in_this_vector". Alternatively, using the **up arrow** key will scroll through previous commands.

Common error messages

Let's try a simple task of drawing a random sample from a vector.

```
>> vect=[1:100]; % create a vector of integers 1 to 100
>> ind=rand(1,1)*100; % pick an index
>> vect(ind) % access index
Subscript indices must either be real positive integers or logicals.
```

Right. So the error message seems to suggest there is something wrong with an index. We take a look at the value of the index and find:

```
>> ind

ind =

   31.1215
```

The issue is that we were using a non-integer index, i.e. there is a 31$^{\text{st}}$ and 32$^{\text{nd}}$ element in the list but not a 31.1215$^{\text{stnd}}$ element. We fix this by:

```
>> ind=round(rand(1,1)*100); % pick an index
>> vect(ind) % access index

ans =

    53

>> ind

ind =

    53
```

Our code is “fixed”. We keep running it and then run into:

```
>> ind=round(rand(1,1)*100); % pick an index
>> vect(ind) % access index
Subscript indices must either be real positive integers or logicals.
```

What happened? Checking the value of `ind` again we find:

```
>> ind

ind =

    0
```

MATLAB does not have a zero index and so it returned another error. Just for giggles, we find that strangely MATLAB gives different errors for a similar problem. MATLAB is funny like that.

```
>> vect(101) % taunt MATLAB
Index exceeds matrix dimensions.

>> vect(-1) % taunt MATLAB a second time
Index exceeds matrix dimensions.
```

The error message makes sense for the first one. We tried accessing an element more than the vector has so yes, in fact, our index did exceed the dimensions of the matrix. The second message is puzzling because -1 is not a positive integer and so should return the same error message as `vect(0)` but somehow it doesn't. Now we know.

Coding challenges

Binomial code. We want a function that takes in two variables: the sample size N and the probability p of being chosen and returns y, the number chosen– effectively sampling from a binomial distribution. Imagine that N is very large N, say 10^6, and p is small, say somewhere between 10^{-5} and .5. There are many ways of doing this code but here speed is of the utmost importance as this code will be sampled potentially millions of times. Write as fast a code as you can that still accurately sample from a binomial distribution. Note, MATLAB has a built in command called `binornd` that performs this task. Can you determine what algorithm `binornd` based on clock performance alone? Can you beat `binornd`?

Neutral sequence evolution. There is a population of organisms who each contain a 100 base pair piece of DNA code. We are interested in the neutral evolution of this code over time. The population goes through expansions and contractions so that it starts at 10^6 and then goes through a growth phase of 10 generations of reproduction (growing by a factor of 2^{10}) to $\approx 10^9$. Of this population 10^6 are randomly selected and the process continues. The DNA code of each organism is identical at the starting point and is a vector of 100 entries between 1 and 4 (for A,C,G,T). Each base pair has a probability μ of mutating into another base pair when the organism reproduces. Write a MATLAB code that tracks the evolution of the DNA code of these organisms.

Evolutionary algorithm. Write an evolutionary algorithm that finds the optimum of the function $\exp\left(-\frac{x-200^2}{2*100^2}\right)$. Create a population of binary numbers with 10 bits and allow random mating based on fitness values as well as mutations. A rough outline of an evolutionary algorithm is shown below.

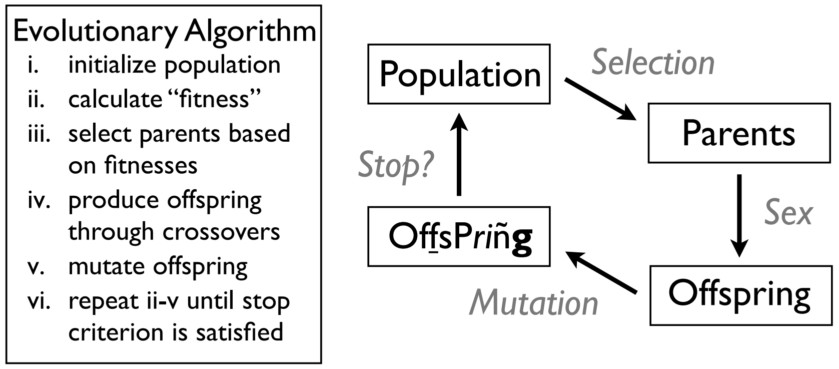


Figure 13. EvoAlg.pdf