MATLAB Tutorial – CCN Course 2012
How to code a neural network simulation

Malte J. Rasch
National Key Laboratory of Cognitive Neuroscience and Learning
Beijing Normal University
China

July 17, 2012
Overview

- Basic introduction to MATLAB
- Learn to code a neural network simulation
- Further exercises with solutions
Why MATLAB?

- **Pro:**
  - Matrix-like numerical operations very fast and easy to use
  - Good plotting capabilities
  - Script language for programming small to medium sized problems in applied mathematics (rapid prototyping)
  - Widely used in the neuroscience community for data analysis as well as computational projects
Why MATLAB?

**Pro:**
- Matrix-like numerical operations very fast and easy to use
- Good plotting capabilities
- Script language for programming small to medium sized problems in applied mathematics (rapid prototyping)
- Widely used in the neuroscience community for data analysis as well as computational projects

**Contra:**
- Support of symbolic/analytic expressions less advanced
  - Mathematica, Maple
- Often not flexible/ fast enough for big projects
  - e.g. Python much more versatile
  - specialized software for detailed/large neural network simulations
    - (NEURON, PCSIM, NEST, GENESIS, ...)
- Very expensive, especially when using on a cluster
  - free alternative: (scientific) python
Starting MATLAB

- Easy: Just click on MATLAB symbol...
Scalar expressions

**Binary operations**: work as expected, use $= + - * / ^$.

Example (compute $y = \frac{a^2 x}{2+a} + b$)

```matlab
>> a = 2;
>> b = 1;
>> x = 0.5;
>> y = a^2*x/(2+a) + b;
>> y
y =
    1.500
```

**Unary operations**: called as functions with $(())$, eg. sin cos tan atan sqrt log gamma

Example (compute $y = \frac{\sqrt{\sin x}}{\ln x}$)

```matlab
>> x = 0.5;
>> y = sqrt(sin(x))/log(x);
```
MATLAB == MATrix LABoratory

Not suprisingly, in MATLAB everything is about matrices.
MATLAB == MATrix LABoratory

Not suprisingly, in MATLAB everything is about matrices.

However, the matrix-like datastructure in MATLAB is better called a **n-dimensional array**, because it can be manipulated in non-algebraic ways.
MATLAB == MatTrix LABoratory

Not suprisingly, in MATLAB everything is about matrices.

However, the matrix-like datastructure in MATLAB is better called a **n-dimensional array**, because it can be manipulated in non-algebraic ways.

- (Almost) all functions will work on arrays as well
  - usually element-wise
- Many MATLAB functions will produce arrays as output
- Array operations *much* faster than for-looped element-wise operation
Arrays: Overview

- How to initialize arrays
- Indexing
- Calculating with arrays
Calculating with arrays is

- straight-forward
- however, carefully check
  - the size of matrices
  - if element-wise or matrix-like operations are intended
  - which matrix dimension to operate on
Calculating with arrays is

- straight-forward
- however, carefully check
  - the size of matrices
  - if element-wise or matrix-like operations are intended
  - which matrix dimension to operate on

Example (compute $y_i = Ax_i + b$ with $b, x_i \in \mathbb{R}^2$)

```matlab
>> A = [1,0.2;0.4,1];
>> b = [1;2] + 0.1;
>> x = 2*randn(2,1);
>> y = A * x + b
y =
    4.5535
    1.4856

>> N = 5;
% same as b = [b,b,b,b,b]
>> bi = repmat(b,[1,N]);
>> xi = 2*randn(2,N);
>> xi(:,1) = x;
>> yi = A * xi + bi
yi =
    4.5535  2.3126  [...]  -0.9021
    1.4856  6.8091  [...]  -0.1080
```
Syntax for initializing arrays

Implicitly, using function returning an array
Syntax for initializing arrays

1. Implicitly, using function returning an array
2. By explicit concatenation
   - Concatenation of **columns** of arrays
     \[ \text{arr1, arr2, ... , arrn} \]
   - Concatenation of **rows** of arrays
     \[ \text{arr1; arr2; ... ; arrn} \]
Syntax for initializing arrays

1. Implicitly, using function returning an array
2. By explicit concatenation
   - Concatenation of **columns** of arrays
     
     [ arr1, arr2, ... , arrn]
   - Concatenation of **rows** of arrays
     
     [ arr1; arr2; ... ; arrn]

**Note:** an scalar is also regarded as an array (of size [1,1]).

**Note 2:** arrays must have matching sizes.
Syntax for initializing arrays

1. Implicitly, using function returning an array
2. By explicit concatenation
   - Concatenation of **columns** of arrays
     
     \[
     [ \text{arr1}, \text{arr2}, \ldots, \text{arrn}]
     \]
   - Concatenation of **rows** of arrays
     
     \[
     [ \text{arr1}; \text{arr2}; \ldots; \text{arrn}]
     \]

**Note:** an scalar is also regarded as an array (of size \([1,1]\)).

**Note 2:** arrays must have matching sizes.

---

**Example (Concatenation)**

```plaintext
>> A = [1,2;3,4]  >> C = [A;A]
A =            C =
1  2           1  2
3  4           3  4

>> B = [A,A]
B =
1  2  1  2
3  4  3  4
```
Syntax for initializing arrays: implicitly

Functions that pre-allocate memory and set each array element to an initial value:

- `zeros` – all zero ND-arrays
- `ones` – all one ND-arrays
- `rand` – random ND-arrays (equally in \([0, 1]\))
Syntax for initializing arrays: implicitly

Functions that pre-allocate memory and set each array element to an initial value:

- **zeros** – all zero ND-arrays
- **ones** – all one ND-arrays
- **rand** – random ND-arrays (equally in \([0, 1]\))
- **colon (:)** – for linear sequences, syntax:
  
  \[
  \text{istart:[]} [\text{step:}] \text{ien}\d
  \]
Syntax for initializing arrays: implicitly

Functions that pre-allocate memory and set each array element to an initial value:

- `zeros` – all zero ND-arrays
- `ones` – all one ND-arrays
- `rand` – random ND-arrays (equally in [0, 1])
- `colon` (:) – for linear sequences, syntax:
  \[ istart: [\text{step:}] iend \]
  - many others, for details type: `help command`
  - `randn`, `linspace`, `logspace`, `ndgrid`, `repmat`,...
Syntax for initializing arrays: implicitly

Functions that pre-allocate memory and set each array element to an initial value:

- **zeros** – all zero ND-arrays
- **ones** – all one ND-arrays
- **rand** – random ND-arrays (equally in [0, 1])
- **colon (:)** – for linear sequences, syntax:
  
  \[ \text{is} \text{start: [ste} \text{p:] i} \text{end} \]
  
  – many others, for details type: `help command`
  
  randn, linspace, logspace, ndgrid, repmat, ...

### Example (Functions initializing arrays)

<table>
<thead>
<tr>
<th>Code</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>A = zeros(3,3);</code></td>
<td><code>ans =</code> 4 4 4</td>
</tr>
<tr>
<td><code>A = ones(4,4,4);</code></td>
<td><code>x =</code> 3.0 2.5 2.0 1.5 1.0</td>
</tr>
<tr>
<td><code>size(A)</code></td>
<td></td>
</tr>
<tr>
<td><code>x = 3:-0.5:1</code></td>
<td><code>A = ones(2)</code></td>
</tr>
<tr>
<td></td>
<td>1 1</td>
</tr>
<tr>
<td></td>
<td>1 1</td>
</tr>
</tbody>
</table>
Indexing arrays

1. **Subscript of a matrix:**
   access the \((i, j)\)-th element of a 2D-matrix \(A\) of dimension \((m, n)\)
   
   `>> A(i,j)`

   **Note:** The first index is always 1 (not 0 as in most other languages)
Indexing arrays

1. **Subscript of a matrix:**
   access the \((i,j)\)-th element of a 2D-matrix \(A\) of dimension \((m,n)\)
   
   \[
   >> A(i,j)
   \]

   **Note:** The first index is always 1 (not 0 as in most other languages)

2. **Linear index of a matrix:**
   access the \((i,j)\)th element of the 2D-matrix \(A\) of dimension \((m,n)\)
   
   \[
   >> \text{linearidx} = i + (j-1)*m;
   \]
   
   \[
   >> A(\text{linearidx})
   \]
Indexing arrays

1. **Subscript of a matrix:**
   access the \((i,j)\)-th element of a 2D-matrix \(A\) of dimension \((m, n)\)
   
   ```
   >> A(i,j)
   ```

   **Note:** The first index is always 1 (*not 0 as in most other languages*)

2. **Linear index of a matrix:**
   access the \((i,j)\)th element of the 2D-matrix \(A\) of dimension \((m, n)\)
   
   ```
   >> linearidx = i + (j-1)*m;
   >> A(linearidx)
   ```

3. **“Slice” indexing with “:”**
   access the \(i\)-th row and \(j\)th column in \(A\), respectively
   
   ```
   >> A(i,:)
   >> A(:,j)
   ```

   get all elements as a concatenated column vector
   
   ```
   >> A(:)
   ```
Multiple indices

vectors of linear indices can be used

```
>> A([1, 4, 5, 6])
```

access the 1st to 4th rows of the 2D-matrix $A$ of dimension $(m, n)$

```
>> A(1:4, :)
```

access the 2nd $(m, n)$-slice of a 3D-matrix $B$ of dimension $(m, n, p)$

```
>> B(:, :, 2)
```
Indexing arrays (2)

4 Multiple indices
vectors of linear indices can be used

>> A([1, 4, 5, 6])
access the 1st to 4th rows of the 2D-matrix A of dimension \((m, n)\)

>> A(1:4, :)
access the 2nd \((m,n)\)-slice of a 3D-matrix B of dimension \((m, n, p)\)

>> B(:, :, 2)

5 Logical indexing
logical matrices of the same size as A can be used as indices

>> A(A>0)
>> A(find(A>0))
Calculating with arrays

1. **Element-wise interpretation**
   - For instance, \( \sin \), \( \cos \), \( \log \) etc.
   - Reserved symbols, \( .* \), \( ./ \), \( .^ \)

2. **“true” matrix interpretation (with dot product)**
   - Symbols \( * \), \( / \), \( ^ \) etc.

3. **Operations on one specified dimensions of the matrix**
   - For instance, \( \text{sum} \), \( \text{mean} \), \( \text{max} \) etc.

4. **Array manipulations**
   - eg. \( \text{reshape} \), \( \text{repmat} \), \( \text{permute} \), \( \text{circshift} \), \( \text{tril} \)

---

**Example (element-wise product and dot product)**

```matlab
>> A = ones(2,2);        >> A*A
>> A.*A
ans =
ans =
    2    2
   1    1
   2    2
   1    1
```
Useful operations on specified dimensions

**Often used functions on array elements include**

- **size** – number of elements in a dimension
- **sum** – sum of elements along an array dimension
- **prod** – product of elements
- **all, any** – logical AND or OR, respectively
- **mean** – mean of elements
- **max** – maximal element of an array dimension

  – and many more, eg. **min var std median diff**

Functions are usually called as

```plaintext
res = func(arr,dim);
```

**Note**: `max` and `min` are exceptions! Here: `res = max(arr,[],dim);`

For details type: `help func`
MATLAB is a script language

**Scripts** are blocks of code which can be called within MATLAB or within another script.

- They are text-files with extensions “.m”.
- They should contain all commands associated with a scientific project.
  (at least to easily reproduce and the calculations)
MATLAB is a script language

**Scripts** are blocks of code which can be called within MATLAB or within another script.

- They are text-files with extensions “.m”.
- They should contain all commands associated with a scientific project.
  (at least to easily reproduce and the calculations)

There are basically two types of m-files

1. **m-file script**
   - A sequential list of MATLAB commands
   - The variable scope is that of the caller.

2. **m-file function**
   - m-file functions accept input parameters and deliver outputs.
   - The variable scope is different from that of the caller.
m-file scripts

How to write an m-file script?
m-file scripts

How to write an m-file script?

1. Open a text-editor of your choice or use the editor provided with MATLAB

   >> edit myscript
m-file scripts

How to write an m-file script?

1. open a text-editor of your choice or use the editor provided with MATLAB
   
   >> edit myscript

2. Write all your calculations in a text-file with extension “.m” (here myscript.m)
m-file scripts

How to write an m-file script?

1. open a text-editor of your choice or use the editor provided with MATLAB
   
   `>> edit myscript`

2. Write all your calculations in a text-file with extension “.m”
   (here `myscript.m`)

3. Save file in the current working directory
   (or `addpath` to search path)
m-file scripts

How to write an m-file script?

1. open a text-editor of your choice or use the editor provided with MATLAB
   
   `>> edit myscript`

2. Write all your calculations in a text-file with extension “.m” (here myscript.m)

3. Save file in the current working directory
   (or addpath to search path)

4. Call your script by calling it from the “Command Window”
   
   `>> myscript;`
   (no compilation needed)
m-file scripts

**How to write an m-file script?**

1. open a text-editor of your choice or use the editor provided with MATLAB
   ```matlab
   >> edit myscript
   ```
2. Write all your calculations in a text-file with extension “.m” (here myscript.m)
3. Save file in the current working directory (or addpath to search path )
4. Call your script by calling it from the “Command Window”
   ```matlab
   >> myscript;
   ```
   (no compilation needed)
5. **Note:** The variable scope is that of the caller. This means that the variables of the script are now present in the workspace
m-file scripts

Excercise #1 (How to write an m-file script)

Write an m-file computing the Stirling approximation

\[ n! \approx \sqrt{2\pi n} \left( \frac{n}{e} \right)^n \]

Hint: \( \pi \) is defined in MATLAB as \( \text{pi} \), and \( e^x \) as \( \text{exp}(x) \).

Solution #1

Reminder:

1. Open a text-editor
   
   `>> edit myscript`

2. Write calculations in text-file with extension “.m”

3. Save file into current working directory

4. Call your script
   
   `>> myscript;`
m-file functions

How to write an m-file function?
m-file functions

How to write an m-file function?
Identical to writing an m-file script, except:
m-file functions

How to write an m-file function?
Identical to writing an m-file script, except:

• In the first line input and output arguments are defined:

  ```matlab
  function outarg = myfun(inarg1, inarg2, ...);
  ```
m-file functions

How to write an m-file function?
Identical to writing an m-file script, except:

- In the first line input and output arguments are defined:
  ```matlab
  function outarg = myfun(inarg1,inarg2,...);
  ```

- Call the function with:
  ```matlab
  >> result = myfun(p1,p2,...);
  ```
m-file functions

**How to write an m-file function?**

Identical to writing an m-file script, *except*:

- In the first line input and output arguments are defined:
  
  ```matlab
  function outarg = myfun(inarg1,inarg2,...);
  ```

- Call the function with
  
  ```matlab
  >> result = myfun(p1,p2,...);
  ```

**Note:** The variable scope is different from that of the caller. That means, that variables defined in the function body are *NOT* present in the workspace after calling the function.
m-file functions

How to write an m-file function?
Identical to writing an m-file script, except:

- In the first line input and output arguments are defined:
  
  ```matlab
  function outarg = myfun(inarg1,inarg2,...);
  ```

- Call the function with
  
  ```matlab
  >> result = myfun(p1,p2,...);
  ```

**Note:** The variable scope is different from that of the caller. That means, that variables defined in the function body are *NOT* present in the workspace after calling the function

**Note 2:** Input arguments are always referenced by value
m-file functions

How to write an m-file function?
Identical to writing an m-file script, except:

- In the first line input and output arguments are defined:

  ```matlab
  function outarg = myfun(inarg1,inarg2,...);
  ```

- Call the function with

  ```matlab
  >> result = myfun(p1,p2,...);
  ```

Note: The variable scope is different from that of the caller. That means, that variables defined in the function body are NOT present in the workspace after calling the function.

Note 2: Input arguments are always referenced by value.

Note 3: When called the m-file file name is used.
m-file functions

Excercise #2 (writing a m-file function)

Write an m-file function computing the Stirling approximation

\[ n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \]

for a given \( n \)

Reminder:

- In the first line input and output arguments are defined:
  
  ```matlab
  function outarg = myfun(inarg1,inarg2,...);
  ```

- Call the function with
  
  ```matlab
  >> result = myfun(p1,p2,...);
  ```
Basic syntax

The basic syntax is similar to other script languages, eg. “python”.

MATLAB has the usual flow control structures, namely

- `if` .. else for decisions
- for-loop
- while-loop
- `switch` .. case for multiple if-else decisions
- `try` .. catch for handling errors
Basic syntax: flow control (1)

if-else block syntax:

1. if scalar_condition
2. expressions
3. else
4. expressions
5. end

Relational operators, eg.: == (equals), || (or), && (and), ~(not)

for details type: help relop
Basic syntax: flow control (1)

if-else block syntax:

if scalar_condition
    expressions
else
    expressions
end

Relational operators, eg.: == (equals), || (or), && (and), ~ (not)

for details type: help relop

Example (if-else)

a = rand(1);
if a == 0.5
    fprintf('you are very lucky!');
end
Programming: basic flow control (2)

for-loop block syntax:

```matlab
for i = array
    % i = array(j) in the j-th loop expressions
end
```

(one can also use break and continue keywords)
Programming: basic flow control (2)

for-loop block syntax:

```
for i = array
  % i==array(j) in the j–th loop expressions
end
```

(one can also use break and continue keywords)

Example (for loop)

```
a=0;
for i = 1:100
  a = a+i;
end
```
Plotting with MATLAB

- Very flexible plotting tools
- Many functions for a variety of plot types and graphics drawing
Overview

Just two most useful plotting commands

plot – plots all kinds of lines, dots, markers in 2D
imagesc – plots an 2-D pseudo image
Overview

Just two most useful plotting commands

plot – plots all kinds of lines, dots, markers in 2D
imagesc – plots an 2-D pseudo image

Note: For an overview of 2D plotting commands: help graph2d
Note 2: For fancy graphs see: help specgraph
How to plot:

1. Open a figure (window) with
   ```
   >> figure;
   ```

2. Create an axes object with
   ```
   >> axes;
   ```

3. Type the plotting command of your choice
   ```
   >> plot(x,y,’b--’,’LineWidth’,2);
   ```
   **Note:** graphics commands draw into the current axes (gca) on the current figure (gcf). They automatically create a new figure and axes if necessary.

4. Make the plot nicer by adding labels and setting limits, eg:
   ```
   >> xlabel(’Time [s]’);
   >> ylabel(’Response [spks/sec]’);
   >> xlim([-1,1]); ylim([-2,2]);
   >> title(’Simulation’)
   ```
plot command

Basic syntax:

```matlab
handle = plot(X,Y,linespec,optname1,val1,...);
```

**X,Y** – x- and y-values to plot. If Y is a 2-D array, all columns are plotted as different lines

**linespec** – a string with a short hand for color and line style and marker type. See `help plot` for an overview. Eg, `linespec = ':ko'` plots dotted (:) black line (k) with a circle at each given coordinate \((x_i, y_i)\)

**optname1** – a string specifying the option name, eg. ’LineWidth’

**val1** – a corresponding value.

**handle** – graphics handle for later reference, eg. with

```matlab
>> set(handle, optname1, val1)
```

**Tip:** To get an overview over all possible options, try `get(handle)`
Plotting example: empirical PDF of Gaussian

```matlab
figure;

% plot Gaussian PDF
x = -4:0.01:4;
y = exp(-0.5*x.^2) / sqrt(2*pi);
plot(x,y,'-k','Linewidth',2);

% empirical pdf
% histogram of 1000 random values
[N,z]=hist(randn(1000,1));
%normalization
p = N/sum(N)/(z(2)-z(1));
hold on; %adds all following graphs
    %to the current axes
plot(z,p,'rx','MarkerSize',10);
hold off;

xlabel('Random variable x')
ylabel('Probability density')
legend('true PDF','empirical PDF')
```
Goal of tutorial

- We will program a neural network simulation together.
We will practice on the way:

- Writing scripts
- Usage of array notation
- How to integrate ODEs
- How to plot results
- How to simulate neurons and synapses
- How to program a quite realistic network simulation
What has to be done in principle:

- $n$ neurons, excitatory and inhibitory, are inter-connected with synapses.
- The each neuron and synapse follows a particular dynamic over time.
- The simulation calculates the dynamic and spiking activity of each neuron for each $t$.
- The network gets some input and reacts in a certain and the response of the network is plotted.
We will proceed in 4 successive steps

1. Simulate a single neuron with current step input
2. Simulate a single neuron with Poisson input
3. Simulate 1000 neurons (no recurrent connections)
4. Simulate complete network (conductance based)
Which neuron model to use?

Biophysical model (i.e. Hodgkin-Huxley model)

\[ C_m \frac{dV_m}{dt} = -\frac{1}{R_m} (V_m - V_L) - \sum_i g_i(t)(V_m - E_i) + I_{syn} + I_{app} \]

Including non-linear dynamics of many channels in \( g_i(t) \)
Which neuron model to use?

Biophysical model (i.e. Hodgkin-Huxley model)

\[ C_m \frac{dV_m}{dt} = -\frac{1}{R_m}(V_m - V_L) - \sum_i g_i(t)(V_m - E_i) + l_{syn} + l_{app} \]

Including non-linear dynamics of many channels in \( g_i(t) \)

Mathematical simplification (Izhikevich, book chapter 8)

\[ \dot{v} = (0.04v + 5)v + 150 - u - \sum_j w_j g_j(v - E_j) + l_{app} \]

\[ \dot{u} = a(bv - u) \]

\[ v \leftarrow c, \quad u \leftarrow u + d, \quad \text{if} \quad v \geq 35 \]

with \( \dot{g}_j = -g_j/\tau_g \) and \( g_j \leftarrow g_j + 1 \) if \( v_j \geq 35 \). It is \( b = 0.2, c = -65, \) and \( d = 8, a = 0.02 \) for exc. neurons and \( d = 2, a = 0.1 \) for inh. neurons.
**Neuron model**

\[
v' = 0.04v^2 + 5v + 140 - u + I \\
u' = a(bv - u)
\]

*If* \( v = 30 \text{ mV} \),
*then* \( v - c, \ u - u + d \)

**regular spiking (RS)**

**intrinsically bursting (IB)**

**chattering (CH)**

**fast spiking (FS)**

**thalamo-cortical (TC)**

**thalamo-cortical (TC)**

**resonator (RZ)**

**low-threshold spiking (LTS)**
Step 1: Simulate a single neuron with injected current

Exercise 1

Simulate the neuron model for 1000ms and plot the resulting voltage trace. Apply a current step ($I_{app} = 7 \text{pA}$) between time 200ms and 700ms.
Step 1: Simulate a single neuron with injected current

Exercise 1

Simulate the neuron model for 1000ms and plot the resulting voltage trace. Apply a current step ($I_{app} = 7pA$) between time 200ms and 700ms.

Neuron model:

\[ \begin{align*}
\dot{v} &= (0.04v + 5)v + 140 - u + I_{app} \\
\dot{u} &= a(bv - u) \\
v &\leftarrow c, \ u \leftarrow u + d, \ \text{if} \ v \geq 35
\end{align*} \]

with $d = 8$, $a = 0.02$, $b = 0.2$, $c = -65$ for an excitatory neuron.
Step 1 in detail:

Open MATLAB and create a new file (script) that will simulate the neuron. If you do not know how to use a command get some help.

Proceed as follows:

1. Initialize parameter values ($\Delta t = 0.5\text{ms}$, $a = 0.02$, $d = 8$, $\cdots$)
Step 1 in detail:

Open MATLAB and create a new file (script) that will simulate the neuron. If you do not know how to use a command get some help.

Proceed as follows:

1. Initialize parameter values ($\Delta t = 0.5\text{ms}$, $a = 0.02$, $d = 8$, …)

2. Reserve memory for voltage trace $v$ and $u$ (of length $T = 1000/\Delta t$) and set first element to $-70$ and $-14$, respectively.
Step 1 in detail:

Open MATLAB and create a new file (script) that will simulate the neuron. If you do not know how to use a command get some help.

Proceed as follows:

1. Initialize parameter values ($\Delta t = 0.5\text{ms}$, $a = 0.02$, $d = 8$, $\cdots$)

2. Reserve memory for voltage trace $v$ and $u$ (of length $T = 1000/\Delta t$) and set first element to $-70$ and $-14$, respectively.

3. Loop over $T - 1$ time steps and do for each step $t$.
**Step 1 in detail:**

Open MATLAB and create a new file (script) that will simulate the neuron. If you do not know how to use a command get some help.

**Proceed as follows:**

1. Initialize parameter values ($\Delta t = 0.5\text{ms}$, $a = 0.02$, $d = 8$, $\cdots$)

2. Reserve memory for voltage trace $v$ and $u$ (of length $T = 1000/\Delta t$) and set first element to $-70$ and $-14$, respectively.

3. Loop over $T - 1$ time steps and do for each step $t$:
   - set $I_{\text{app}} \leftarrow 7$ if $t\Delta t$ is between 200 and 700 (otherwise 0)
Step 1 in detail:

Open MATLAB and create a new file (script) that will simulate the neuron. If you do not know how to use a command get some help.

Proceed as follows:

1. Initialize parameter values ($\Delta t = 0.5\text{ms}$, $a = 0.02$, $d = 8$, …)

2. Reserve memory for voltage trace $v$ and $u$ (of length $T = 1000/\Delta t$) and set first element to $-70$ and $-14$, respectively.

3. Loop over $T - 1$ time steps and do for each step $t$

   1. set $I_{\text{app}} \leftarrow 7$ if $t\Delta t$ is between 200 and 700 (otherwise 0)

   2. if $v_t < 35$: update element $t + 1$ of $v$ and $u$ according to $v_{t+1} \leftarrow v_t + \Delta t \{0.04 v_t + 5\} v_t - u_t + 140 + I_{\text{app}}$

   $u_{t+1} \leftarrow u_t + \Delta t a (b v_t - u_t)$
Step 1 in detail:

Open MATLAB and create a new file (script) that will simulate the neuron. If you do not know how to use a command get some help.

Proceed as follows:

1. Initialize parameter values ($\Delta t = 0.5\text{ms}$, $a = 0.02$, $d = 8$, $\cdots$)

2. Reserve memory for voltage trace $v$ and $u$ (of length $T = 1000/\Delta t$) and set first element to $-70$ and $-14$, respectively.

3. Loop over $T - 1$ time steps and do for each step $t$

   1. set $I_{\text{app}} \leftarrow 7$ if $t\Delta t$ is between 200 and 700 (otherwise 0)

   2. if $v_t < 35$: update element $t + 1$ of $v$ and $u$ according to:

      $$v_{t+1} \leftarrow v_t + \Delta t \left\{ (0.04 v_t + 5) v_t - u_t + 140 + I_{\text{app}} \right\}$$

      $$u_{t+1} \leftarrow u_t + \Delta t \ a (b v_t - u_t)$$

   3. if $v_t \geq 35$: set $v_{t+1} \leftarrow c$ and $u_{t+1} \leftarrow u_t + d$ (and optional $v_t \leftarrow 35$)
Step 1 in detail:

Open MATLAB and create a new file (script) that will simulate the neuron. If you do not know how to use a command get some help.

Proceed as follows:

1. Initialize parameter values ($\Delta t = 0.5\text{ms}$, $a = 0.02$, $d = 8$, \ldots)

2. Reserve memory for voltage trace $v$ and $u$ (of length $T = 1000/\Delta t$) and set first element to $-70$ and $-14$, respectively.

3. Loop over $T-1$ time steps and do for each step $t$

   1. set $I_{\text{app}} \leftarrow 7$ if $t\Delta t$ is between 200 and 700 (otherwise 0)

   2. if $v_t < 35$: update element $t+1$ of $v$ and $u$ according to

      $$v_{t+1} \leftarrow v_t + \Delta t \left\{ (0.04 v_t + 5) v_t - u_t + 140 + I_{\text{app}} \right\}$$

      $$u_{t+1} \leftarrow u_t + \Delta t a (b v_t - u_t)$$

   3. if $v_t \geq 35$: set $v_{t+1} \leftarrow c$ and $u_{t+1} \leftarrow u_t + d$ (and optional $v_t \leftarrow 35$)

4. Plot the voltage trace $v$ versus $t$. 
Solution to step 1

1  % 1)  initialize  parameters
2  dt = 0.5;
3  d = 8;
4  a = 0.02;
5  c = -65;
6  b = 0.2;
7  T = ceil(1000/dt);
8
9  % 2)  reserve  memory
10  v = zeros(T, 1);
11  u = zeros(T, 1);
12  v(1) = -70;  %  resting  potential
13  u(1) = -14;  %  steady  state
14
15  %3)  for-loop  over  time
16  for  t = 1:T-1;
17      %3.1)  get  input
18      if  t*dt>200  &&  t*dt<700
19          lapp = 7;
20      else
21          lapp = 0;
22          end
23
24      if  v(t)<35
25          %3.2)  update  ODE
26              dv = (0.04*v(t)+5)*v(t)+140-u(t);
27              v(t+1) = v(t) + (dv+lapp)*dt;
28              du = a*(b*v(t)-u(t));
29              u(t+1) = u(t) + dt*du;
30      else
31          %3.3)  spike  !
32              v(t) = 35;
33              v(t+1) = c;
34              u(t+1) = u(t)+d;
35          end
36  end
37
38  % 4)  plot  voltage  trace
39  plot((0:T-1)*dt,v,'b');
40  xlabel('Time [ms]');
41  ylabel('Membrane voltage [mV]');
Step 2: Single neuron with synaptic input

Exercise 2

Simulate the neuron model for 1000ms and plot the resulting voltage trace. Assume that 100 synapses are attached to the neuron, with each pre-synaptic neuron firing with a Poisson process of rate $f_{\text{rate}} = 2$ Hz between time 200ms and 700ms.
Step 2: Single neuron with synaptic input

Exercise 2

Simulate the neuron model for 1000ms and plot the resulting voltage trace. Assume that 100 synapses are attached to the neuron, with each pre-synaptic neuron firing with a Poisson process of rate $f_{\text{rate}} = 2$ Hz between time 200ms and 700ms.

Synaptic input model:

$$l_{\text{syn}} = \sum_j w_j^{\text{in}} g_j^{\text{in}}(t)(E_j^{\text{in}} - v(t))$$

$$\dot{g}_j^{\text{in}} = g_j^{\text{in}} / \tau_g$$

$$g_j^{\text{in}} \leftarrow g_j^{\text{in}} + 1, \text{ if } r_j(t) < f_{\text{rate}} \Delta t$$

with $\tau_g = 10$ms, weights $w_j^{\text{in}} = 0.07$, $E_j = 0$, $r_j(t) \in [0, 1]$ uniform random numbers drawn for each step $t$, and $j = 1 \ldots 100$. 
Step 2 in detail:

Use the last script, save it under a new file name, and add the necessary lines.

Proceed as follows:

1. Initialize new parameter values \( \tau_g = 10, \ f_{rate} = 0.002\text{ms}^{-1} \)
Step 2 in detail:

Use the last script, save it under a new file name, and add the necessary lines.

**Proceed as follows:**

1. Initialize new parameter values ($\tau_g = 10$, $f_{rate} = 0.002\text{ms}^{-1}$)
2. Reserve memory and initialize $\mathbf{g}^{in} = (g_j^{in})$, $\mathbf{w}^{in} = (w_j^{in})$, and $\mathbf{E} = (E_j)$ (vectors of length $n_{in} = 100$) with constant elements
Step 2 in detail:

Use the last script, save it under a new file name, and add the necessary lines.

**Proceed as follows:**

1. Initialize new parameter values ($\tau_g = 10$, $f_{rate} = 0.002\text{ms}^{-1}$)
2. Reserve memory and initialize $\mathbf{g}^{\text{in}} = (g_j^{\text{in}})$, $\mathbf{w}^{\text{in}} = (w_j^{\text{in}})$, and $\mathbf{E} = (E_j)$ (vectors of length $n_\text{in} = 100$) with constant elements
3. Inside the for-loop change/add the following:
Step 2 in detail:

Use the last script, save it under a new file name, and add the necessary lines.

**Proceed as follows:**

1. Initialize new parameter values ($\tau_g = 10$, $f_{rate} = 0.002\text{ms}^{-1}$)
2. Reserve memory and initialize $\mathbf{g}^{in} = (g_j^{in})$, $\mathbf{w}^{in} = (w_j^{in})$, and $\mathbf{E} = (E_j)$ (vectors of length $n_{in} = 100$) with constant elements
3. Inside the for-loop change/add the following:
   - set $p_j = 1$ if $r \leq f_{rate}\Delta t$ (otherwise 0) in the case when $i\Delta t$ is between 200 and 700 (otherwise 0). $\mathbf{r} = (r_j)$ is a vector of uniform random numbers of length $n_{in}$
Step 2 in detail:

Use the last script, save it under a new file name, and add the necessary lines.

**Procede as follows:**

1. Initialize new parameter values ($\tau_g = 10$, $f_{\text{rate}} = 0.002\text{ms}^{-1}$)

2. Reserve memory and initialize $g^{\text{in}} = (g_j^{\text{in}})$, $w^{\text{in}} = (w_j^{\text{in}})$, and $E = (E_j)$ (vectors of length $n_{\text{in}} = 100$) with constant elements.

3. Inside the for-loop change/add the following:
   
   1. set $p_j = 1$ if $r \leq f_{\text{rate}}\Delta t$ (otherwise 0) in the case when $i\Delta t$ is between 200 and 700 (otherwise 0). $r = (r_j)$ is a vector of uniform random numbers of length $n_{\text{in}}$

   2. before the $v_t$ update: implement the conductance dynamics $g$ and set $I_{\text{app}}$ (using array notation carefully): 

      $$
      g_j^{\text{in}} \leftarrow g_j^{\text{in}} + p_j \\
      I_{\text{app}} \leftarrow w^{\text{in}} \cdot (g^{\text{in}} \odot E^{\text{in}}) - (w^{\text{in}} \cdot g^{\text{in}}) \odot v_t \\
      g_j^{\text{in}} \leftarrow (1 - \Delta t/\tau_g) g_j^{\text{in}}
      $$


Solution to step 2

1 %initialize parameters
2 dt = 0.5;
3 d = 8;
4 a = 0.02;
5 c = -65;
6 b = 0.2

7 %reserve memory
8 T = ceil(1000/dt);
9 v = zeros(T,1);
10 u = zeros(T,1);
11 v(1) = -70;
12 u(1) = -14;
13 n_in = 100;
14 rate = 2*1e-3;%to [ms]
15 tau_g = 10;
16 g_in = zeros(n_in,1);
17 E_in = zeros(n_in,1);
18 w_in = 0.07*ones(1,n_in);
19
20 %for-loop over time
21 for t = 1:T-1;
22     %Poisson input
23     if t*dt>200 && t*dt<700
24         p = rand(n_in,1)<rate*dt;%NEW
25     else
26         p = 0;
27     end
28     %conductance update of g_in
29     g_in = g_in + p;
30     lapp = w_in*(g_in.*E_in);
31     lapp = lapp - (w_in*g_in).*v(t);
32     g_in = (1 - dt/tau_g)*g_in;
33
34     if v(t)<35
35         %update ODE
36         dv = (0.04*v(t)+5)*v(t)+140-u(t);
37         v(t+1) = v(t) + (dv+lapp)*dt;
38         du = a*(b*v(t)-u(t));
39         u(t+1) = u(t) + dt*du;
40     else
41         %spikes
42         v(t) = 35;
43         v(t+1) = c;
44         u(t+1) = u(t)+d;
45     end
46
47 end

48 %plot voltage trace
49 plot((0:T-1)*dt,v,'b');
50 xlabel('Time [ms]');
51 ylabel('Membrane voltage [mV]');
Step 3: Simulate 1000 neurons (not inter-connected)

Exercise 3

Simulate 1000 neurons for 1000 ms and plot the resulting spikes. Assume that each neuron receives (random) 10% of the 100 Poisson spike trains of rate $f_{\text{rate}} = 2$ Hz between time 200 ms and 700 ms. Note that the neurons are not yet inter-connected.
Step 3: Simulate 1000 neurons (not inter-connected)

Exercise 3

Simulate 1000 neurons for 1000 ms and plot the resulting spikes. Assume that each neuron receives (random) 10% of the 100 Poisson spike trains of rate $f_{\text{rate}} = 2$ Hz between time 200 ms and 700 ms. Note that the neurons are not yet inter-connected.

Excitatory and inhibitory neurons:

A neuron is, with probability $p_I = 0.2$, a (fast-spiking) inhibitory neuron ($a = 0.1$, $d = 2$), others are (regular spiking) excitatory neurons ($a = 0.02$ and $d = 8$). Weights of the input synapse $j$ to inhibitory neuron $i$ is $w^{\text{in}} = 0.07$ if connected (otherwise 0).
Step 3 in detail:

Modify the last script (after saving it under new name).

**Proceed as follows:**

1. Initialize new parameter values \((n = 1000)\)
Step 3 in detail:

Modify the last script (after saving it under new name).

Proceed as follows:

1. Initialize new parameter values \( (n = 1000) \)

2. Initialize 2 logical vectors (for indexing \( k_{\text{inh}} \) and \( k_{\text{exc}} \) of length \( n \), where \( k_{\text{inh}} \) has a 1 in element \( i \) with probability \( p = 0.2 \) (marking an inhibitory neuron) and 0 otherwise. \( k_{\text{exc}} = \neg k_{\text{inh}} \).
Step 3 in detail:

Modify the last script (after saving it under new name).

**Proceed as follows:**

1. Initialize new parameter values ($n = 1000$)

2. Initialize 2 logical vectors (for indexing) $k_{inh}$ and $k_{exc}$ of length $n$, where $k_{inh}$ has a 1 in element $i$ with probability $p = 0.2$ (marking an inhibitory neuron) and 0 otherwise. $k_{exc} = \neg k_{inh}$.

3. Reserve memory and initialize $v_{ij}$, $u_{ij}$ (now being $n \times T$ matrices). Set parameters $a_i$ and $d_i$ according to $k_{exc}$ and $k_{inh}$. 
Step 3 in detail:

Modify the last script (after saving it under new name).

Proceed as follows:

1. Initialize new parameter values \( n = 1000 \)

2. Initialize 2 logical vectors (for indexing \( k_{\text{inh}} \) and \( k_{\text{exc}} \) of length \( n \), where \( k_{\text{inh}} \) has a 1 in element \( i \) with probability \( p = 0.2 \) (marking an inhibitory neuron) and 0 otherwise. \( k_{\text{exc}} = \neg k_{\text{inh}} \).

3. Reserve memory and initialize \( v_{ij}, u_{ij} \) (now being \( n \times T \) matrices). Set parameters \( a_i \) and \( d_i \) according to \( k_{\text{exc}} \) and \( k_{\text{inh}} \).

4. The weights \( w_{ij}^{\text{in}} = 0.07 \) now form a \( n \times n_{\text{in}} \) matrix. Set 90 % random elements to 0 to account for the connection probability.
Step 3 in detail:

Modify the last script (after saving it under new name).

Proceed as follows:

1. Initialize new parameter values \((n = 1000)\)

2. Initialize 2 logical vectors (for indexing) \(k_{\text{inh}}\) and \(k_{\text{exc}}\) of length \(n\), where \(k_{\text{inh}}\) has a 1 in element \(i\) with probability \(p = 0.2\) (marking an inhibitory neuron) and 0 otherwise. \(k_{\text{exc}} = \neg k_{\text{inh}}\).

3. Reserve memory and initialize \(v_{ij}\), \(u_{ij}\) (now being \(n \times T\) matrices). Set parameters \(a_i\) and \(d_i\) according to \(k_{\text{exc}}\) and \(k_{\text{inh}}\).

4. The weights \(w_{ij}^{\text{in}} = 0.07\) now form a \(n \times n_{\text{in}}\) matrix. Set 90% random elements to 0 to account for the connection probability.

5. Inside the for-loop change/add the following:

   1. Same update equations (for \(v_{i,t+1}\) and \(u_{i,t+1}\)) but use array notation to update all \(i\) neurons simultaneously.
Step 3 in detail:

Modify the last script (after saving it under new name).

**Proceed as follows:**

1. Initialize new parameter values \( (n = 1000) \)

2. Initialize 2 logical vectors (for indexing) \( k_{\text{inh}} \) and \( k_{\text{exc}} \) of length \( n \), where \( k_{\text{inh}} \) has a 1 in element \( i \) with probability \( p = 0.2 \) (marking an inhibitory neuron) and 0 otherwise. \( k_{\text{exc}} = \neg k_{\text{inh}} \).

3. Reserve memory and initialize \( v_{ij} \), \( u_{ij} \) (now being \( n \times T \) matrices). Set parameters \( a_i \) and \( d_i \) according to \( k_{\text{exc}} \) and \( k_{\text{inh}} \).

4. The weights \( w_{ij}^{\text{in}} = 0.07 \) now form a \( n \times n_{\text{in}} \) matrix. Set 90 % random elements to 0 to account for the connection probability.

5. Inside the for-loop change/add the following:
   
   1. Same update equations (for \( v_{i,t+1} \) and \( u_{i,t+1} \)) but use array notation to update all \( i \) neurons simultaneously.

6. Plot the spike raster. Plot black dots at \( \{(t, i) | v_{it} \geq 35\} \) for excitatory neuron \( i \). Use red dots for inhibitory neurons.
Solution to step 3

1 % initialize parameters
2 dt = 0.5;
3 n = 1000; % no of neurons
4 inh = rand(n,1)<0.2; % kinh
5 exc = ~inh; % others exc
6 d = 8*exc + 2*inh; % for exc/inh
7 a = 0.02*exc + 0.1*inh;
8
9 % reserve memory
10 T = ceil(1000/dt);
11 v = zeros(n,T); % now: n x T
12 u = zeros(n,T); % now: n x T
13 v(:,1) = -70; % vectors
14 u(:,1) = -14;
15
16 n_in = 100;
17 rate = 2*1e-3;
18 tau_g = 10;
19 g_in = zeros(n_in,1);
20 E_in = zeros(n_in,1);
21 W_in = 0.07*ones(n,n_in); % n x n_in
22 W_in(rand(n, n_in)>0.1) = 0; %conn. prob
23
24 % for-loop over time
25 for t = 1:T-1;
26    if t*dt>200 && t*dt<700
27       p = rand(n_in,1)<rate*dt;
28    else
29       p = 0;
30    end
31    g_in = g_in + p;
32 lapp = w_in*(g_in.*E_in);
33 lapp = lapp -(W_in*g_in).*v(:, t); %NEW
34 g_in = (1 - dt/tau_g)*g_in;
35
36 %update vectorized ODE
37 dv = (0.04*v(:, t)+5).*v(:, t)+140-u(:, t);
38 v(:, t+1) = v(:, t) + (dv+lapp)*dt;
39 du = a.*(0.2*v(:, t)-u(:, t));
40 u(:, t+1) = u(:, t) + dt*du;
41
42 %handle spikes (reset v,u)
43 fired = v(:, t)>=35; % neurons fired?
44 v(fired ,t) = 35;
45 v(fired ,t+1) = -65;
46 u(fired ,t+1) = u(fired ,t)+d(fired);
47 end
48
49 %plot spike raster
50 spks = double(v==35);
51 spks(inh,:) = 2*spks(inh,:);
52 clf, hold on;
53 [X,Y] = meshgrid((0:T-1)*dt,1:n);
54 col = 'kr';
55 for k = 1:2 % exc/inh
56    idx = find(spks==k);
57    plot(X(idx),Y(idx),[col(k) '.']);
58 end
59 xlim([0,T*dt]); ylim([0,n])
60 xlabel('Time [ms]')
61 ylabel('Unit #')
Step 4: Simulate recurrent network

Exercise 4

Simulate 1000 neurons as before but with added recurrent connections.
Step 4: Simulate recurrent network

Exercise 4

Simulate 1000 neurons as before but with added recurrent connections.

Recurrent synaptic activations

A neuron $i$ is sparsely (with probability $p_{rc} = 0.1$) connected to a neuron $j$. Thus neuron $i$ receives an additional current $I_{i}^{\text{syn}}$ of the form:

$$I_{i}^{\text{syn}} = \sum_{j=1}^{n} w_{ij}g_{j}(t)(E_{j} - v_{i}(t))$$

Weights are Gamma distributed (scale 0.003, shape 2). Inh. to exc. connections are twice as strong.
Step 4 in detail:

Modify the last script (after saving it under new name).

Proceed as follows:

1. Initialize and allocate memory for the new variables \((g = (g_j), E_j)\).
   Set \(E_j = -85\) if \(j\) is an inhibitory neuron (otherwise 0).
Step 4 in detail:

Modify the last script (after saving it under new name).

**Proceed as follows:**

1. Initialize and allocate memory for the new variables \( g = (g_j), E_j \). Set \( E_j = -85 \) if \( j \) is an inhibitory neuron (otherwise 0).

2. Reserve memory and initialize weights \( W = (w_{ij}) \) to zero. Randomly choose 10% of the matrix elements.
Step 4 in detail:

Modify the last script (after saving it under new name).

**Proceed as follows:**

1. Initialize and allocate memory for the new variables \( \mathbf{g} = (g_j), E_j \). Set \( E_j = -85 \) if \( j \) is an inhibitory neuron (otherwise 0).

2. Reserve memory and initialize weights \( \mathbf{W} = (w_{ij}) \) to zero. Randomly choose 10% of the matrix elements.

3. Set the chosen weight matrix elements to values drawn from a Gamma distribution of scale 0.003 and shape 2.
Step 4 in detail:

Modify the last script (after saving it under new name).

Proceed as follows:

1. Initialize and allocate memory for the new variables \( g = (g_j), E_j \). Set \( E_j = -85 \) if \( j \) is an inhibitory neuron (otherwise 0).

2. Reserve memory and initialize weights \( W = (w_{ij}) \) to zero. Randomly choose 10% of the matrix elements.

3. Set the chosen weight matrix elements to values drawn from a Gamma distribution of scale 0.003 and shape 2.

4. Make the weight matrix “sparse”.
Step 4 in detail:

Modify the last script (after saving it under new name).

Proceed as follows:

1. Initialize and allocate memory for the new variables \( \mathbf{g} = (g_j), E_j \). Set \( E_j = -85 \) if \( j \) is an inhibitory neuron (otherwise 0).

2. Reserve memory and initialize weights \( \mathbf{W} = (w_{ij}) \) to zero. Randomly choose 10\% of the matrix elements.

3. Set the chosen weight matrix elements to values drawn from a Gamma distribution of scale 0.003 and shape 2.

4. Make the weight matrix “sparse”.

5. Scale weights from inh. to exc. neurons by the factor of 2.
Step 4 in detail:

Modify the last script (after saving it under new name).

**Proceed as follows:**

1. Initialize and allocate memory for the new variables \((g = (g_j), E_j)\). Set \(E_j = -85\) if \(j\) is an inhibitory neuron (otherwise 0).

2. Reserve memory and initialize weights \(W = (w_{ij})\) to zero. Randomly choose 10% of the matrix elements.

3. Set the chosen weight matrix elements to values drawn from a Gamma distribution of scale 0.003 and shape 2.

4. Make the weight matrix “sparse”.

5. Scale weights from inh. to exc. neurons by the factor of 2.

6. Inside the for-loop change/add the following:

   1. add the equations for recurrent conductances \(g_j\)

      \[
      g_j \leftarrow g_j + 1, \quad \text{if } v_j(t - 1) \geq 35 \\
      I^{\text{syn}} \leftarrow W \cdot (g \odot E) - (W \cdot g) \odot v \\
      g_j \leftarrow (1 - \Delta t/\tau_g) g_j
      \]
Solution to step 4

1. % initialize parameters
2. dt = 0.5;
3. n = 1000;
4. inh = rand(n,1) < 0.2;
5. exc = ~inh;
6. d = 8 * exc + 2 * inh;
7. a = 0.02 * exc + 0.1 * inh;

8. % reserve memory
9. T = ceil(1000/dt);
10. v = zeros(n,T);
11. u = zeros(n,T);
12. v(:,1) = -70;
13. u(:,1) = -14;
14. n_in = 100;
15. rate = 2 * 1e-3;
16. tau_g = 10;
17. g_in = zeros(n_in,1);
18. E_in = zeros(n_in,1);
19. W_in = 0.07 * ones(n,n_in);
20. W_in(rand(n,n_in) > 0.1) = 0;
21. g = zeros(n,1); % NEW: init RC
22. E = zeros(n,1);
23. E(inh) = -85;
24. W = zeros(n,n); % init W with 0
25. idx = find(rand(n,n) < 0.1); % choose ele.
26. W(idx) = gamrnd(2,.003,length(idx),1);
27. W(exc,inh) = 2 * W(exc,inh); % scale 1->E
28. W = sparse(W); % faster..
29. fired = 0;
30. % for-loop over time
31. for t = 1:T-1;
32.   % Poisson input
33.   if t*dt > 200 && t*dt < 700
34.     p = rand(n_in,1) < rate*dt;
35.   else
36.     p = 0;
37.   end
38.   % input conductances
39.   g_in = g_in + p;
40.   lapp = W_in*(g_in.*E_in);
41.   lapp = lapp - (W_in* g_in).*v(:,t);
42.   g_in = (1 - dt/tau_g)*g_in;
43. % NEW recurrent conductances
44.   g = g + fired; % spikes have arrived
45.   lsyn = W*(g.*E) - (W*g).*v(:,t);
46.   lapp = lapp + lsyn;
47.   g = (1 - dt/tau_g)*g;
48. % update vectorized ODE
49.   dv = (0.04*v(:,t) + 5).*v(:,t) + 140 - u(:,t);
50.   v(:,t+1) = v(:,t) + (dv+lapp)*dt;
51.   du = a.*(0.2*v(:,t)-u(:,t));
52.   u(:,t+1) = u(:,t) + dt*du;
53. % handle spikes (reset v,u)
54.   fired = v(:,t) >= 35; % neurons fired?
55.   v(fired,t) = 35;
56.   v(fired,t+1) = -65;
57.   u(fired,t+1) = u(fired,t)+d(fired);
58. end
59. % plot spike raster: as before
CONGRATULATION!

You have just coded and simulated a quite realistic network model!
Optional exercises
Exercises

Exercise (p-series)

Calculate the p-series (generalization of the Harmonic Series) for a given $p$ up to a given $m$

$$\mu_m = \sum_{i=1}^{m} \frac{1}{n^p}$$

Use array notation.

Advise: Use array notation and avoid for-loops wherever you can!
Exercise (blob movie)

1. Generate a two arrays with, $x$ and $y$, ranging from -2 to 2 (and about 100 elements)

2. Generate two 100 by 100 grid-matrices, $X$ and $Y$ using `meshgrid` with $x$ and $y$ as input (look at $X$ and $Y$ to understand what `meshgrid` is doing).

3. Calculate a matrix $Z$ of the same size as $X$ and $Y$ where each element is given by $z_i = e^{-(x_i^2+y_i^2)}$.

4. Write a loop with $t$ ranging from 0 to 1000 where you
   1. plot the matrix $Z$ (using `imagesc`)
   2. circular shift the matrix $Z$ by 1 element (using `circshift`)
   3. force the plot to be drawn (using `drawnow`)
Exercises

Exercise (Logical indexing and basic plotting)

1. Generate a 100 by 2 matrix $M$ of Gaussian random values
2. Plot a point cloud (first vs. second column of $M$) using blue circles
3. Calculate how many values of $M$ are positive
4. Erase all rows in $M$ which have at least one negative value
5. Plot the points of the modified array $M$ using red crosses in the same graph as above.
6. Set both axes limits to $-3$ to $3$ and plot dotted gray lines on the coordinate axes (where $y = 0$ or $x = 0$).
7. Label the axes and write a legend

Solution
Exercise (Poisson spike trains)

Write a function that generates a homogeneous Poisson spike train of rate $\lambda$ having exactly $N$ spikes. Use array notations ($\text{cumsum}$).

**Hint:** Poisson spike intervals $t$ are exponentially distributed. They can be generated by *inverse transform sampling*: Given uniform random variables $u$ in 0 to 1 valid inter-spike intervals can be calculated as

$$t = -\frac{\log u}{\lambda}$$

**Advise:** Use array notation and avoid for-loops wherever you can!
Exercise (More on spike trains)

Generate a long Poisson spike train (with the function from the last exercise). Compute the mean and standard deviation of the inter-spike interval distribution.

Further, write a function that counts the number of spike times falling into time-bins of length $\Delta t$.

**Hint:** Use `diff` to get the intervals from spike times.

**Hint 2:** Use `histc` to bin the spike-times
Exercise (Plot 2-D Gaussian point cloud)

Write a function which plots a point cloud of $n$ random samples drawn from a 2D Normal distribution (with 0 mean) and variance $\Sigma = (RD)^T (RD)$, where the rotation matrix is defined as

$$R = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix}$$

and $D$ is a diagonal matrix of the standard deviation in the principal directions

$$D = \begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{pmatrix}$$

The function should accept parameters $\sigma_1$, $\sigma_2$, and $\theta$.

**Hint:** use `randn` to produce independent Normal-distributed random vectors $x_i$ and transform them according to $y_i = RDx_i$. 

Solution
Advanced exercise

Exercise (Optimizing MATLAB code)

Optimize “bad” MATLAB code for generating self-organizing maps.

See provided code and description in som_exercise.zip.
Solution: P-series

```matlab
function mu = pseries(n,p);
% PSERIES(N,P) computes the P-series up to N

iarr = (1:n).^p;
mu = sum(1./iarr,2);
```

(back to text)
Solution: Blob movie

```matlab
x = linspace(-2, 2, 100);
y = linspace(-2, 2, 100);

[X,Y] = meshgrid(x,y);

Z = exp(-X.^2 - Y.^2);

for t = 1:1000
    imagesc(Z);
    Z = circshift(Z,[0,1]);
    drawnow;
end
```
Solution: Logical indexing and basic plotting

1. \%1.
2. \texttt{M = randn(100,2);}
3. 
4. \%2.
5. \texttt{plot(M(:,1),M(:,2),'bo');}
6. 
7. \%3.
8. \texttt{npos = sum(M(,:) > 0)}
9. 
10. \%4.
11. \texttt{M(any(M<0,2),:) = [];}
12. 
13. \%5.
14. \texttt{hold on;}
15. \texttt{plot(M(:,1),M(:,2),'rx');}
16. 
17. \%6.
18. \texttt{limits = [-3,3];}
19. \texttt{xlim(limits); ylim(limits)}
20. \texttt{plot(limits,[0,0],'--','Color',[0.5,0.5,0.5])}
21. \texttt{plot([0,0],limits,'--','Color',[0.5,0.5,0.5])}
22. 
23. \%7.
24. \texttt{xlabel('X')}
25. \texttt{ylabel('Y')}
26. \texttt{legend({'Gaussian random samples','Points with X>0 and Y>0'})}
Solution: Poisson spike trains

```matlab
function spkt = poissonspikes(lambda,N)

% SPKT = POISSONSPIKES(LAMBDA,N) generates a
% Poisson spike train with rate LAMBDA
% and exactly N spikes.

isis = -log(rand(N,1))/lambda;
spkt = cumsum(isis);
```

back to text
Solution: More on spike trains

1. \( N = 1000; \) \% number of spikes
2. \( \lambda = 10; \) \% spike rate

3. \%
Poisson spikes
4. \( \text{spkt} = \text{cumsum}(-\log(\text{rand}(N,1))/\lambda); \)

5. \%
mean of the ISI
6. \( \text{misi} = \text{mean}(\text{diff}(\text{spkt})); \)

7. \%
stddev of the ISI
8. \( \text{sisi} = \text{std}(\text{diff}(\text{spkt})); \)

9. \%

10. \textbf{function} \( S = \text{spikebinning}(\text{spkt}, \text{twin}, \text{dt}) \)
11. \%
\( S = \text{SPIKEBINNING}(\text{SPKT}, \text{TWIN}, \text{DT}) \) counts the spike times
12. \%
occurring in each bin of width DT in the time window
13. \%
from TWIN(1) and TWIN(2)
14. \%
\( S = \text{histc}(\text{spkt}, \text{twin}(1):\text{dt}:\text{twin}(2)); \)
Solution: Plot 2-D Gaussian point cloud

```matlab
function plot2DGAussian(sig1,sig2,theta);
% PLOT2DGAUSSIAN(SIG1,SIG2,THETA) plots a Gaussian point cloud with
% standard deviations SIG1 and SIG2 and orientation THETA.

n = 250;
r = randn(2,n);
D = [sig1,0;0,sig2];
R = [cos(theta),-sin(theta);sin(theta),cos(theta)];

x = R*D*r;

%plotting
figure; subplot(2,2,1);
plot(x(1,:),x(2,:),'b','MarkerSize',6);

%resize the axes
mxlen = 1.1*max(abs(x(:)));
xlim([-mxlen,mxlen])
ylim([-mxlen,mxlen])

%plot the 0-coordinates
hold on;
plot([-mxlen,mxlen],[0,0],'--k')
plot([0,0],[-mxlen,mxlen],'--k')
hold off;
```
How to start **MATLAB**

- Easy: Just click on **MATLAB** symbol...
How to get help

- Each MATLAB function has a header which describes its usage.
- Just type:
  ```
  >> help command
  ```
  or
  ```
  >> doc command
  ```

- Alternatively:
  ```
  http://www.mathworks.com/access/helpdesk/help/techdoc/
  ```
How to plot in general

How to plot:

1. Open a figure (window) with
   ```
   >> figure;
   ```

2. Issue plotting command of your choice
   ```
   >> plot(x,y,'b--','LineWidth',2);
   ```

   ![HowTo](Note: graphics commands draw into the current axes (gca) on the current figure (gcf). They automatically create a new figure and axes if necessary.

3. Make the plot nicer by adding labels and setting limits, eg:
   ```
   >> xlabel('Time [s]');
   >> ylabel('Response [spks/sec]');
   >> xlim([-1,1]); ylim([-2,2]);
   >> title('Simulation')
   ```
How to use the plot command

Basic syntax:

\[ handle = \text{plot}(X,Y,\text{linespec},\text{optname1},\text{val1},\ldots); \]

- **X,Y** – x- and y-values to plot. If Y is a 2-D array, all columns are plotted as different lines
- **linespec** – a string with a short hand for color and line style and marker type. See `help plot` for an overview. Eg, `linespec = ':ko'` plots dotted (:) black line (k) with a circle at each given coordinate \((x_i, y_i)\)
- **optname1** – a string specifying the option name, eg. `'LineWidth'`
- **val1** – a corresponding value.
- **handle** – graphics handle for later reference, eg. with `>> \text{set}(\text{handle}, \text{optname1}, \text{val1})`

Tip: To get an overview over all possible options, try `get(handle)`
How to initialize parameters

Just use the syntax

```matlab
>> parname = value;
```

Example

```matlab
>> a = 2;
>> vreset = 0;
>> tau = 0.02
```

```
tau =
    0.02
```
How to use scalar expressions

**Binary operations**: work as expected, use $= + - * / ^$

Example (compute $y = \frac{a^2 x}{2+a} + b$)

```matlab
>> a = 2;
>> b = 1;
>> x = 0.5;
>> y = a^2*x/(2+a) + b;
>> y
y =
    1.500
```

back
# How to initialize arrays

1. Implicitly, using function returning an array
2. By explicit concatenation
   - Concatenation of **columns** of arrays
     
     \[
     \begin{bmatrix}
     \text{arr1}, \text{arr2}, \ldots, \text{arrn}
     \end{bmatrix}
     \]
   - Concatenation of **rows** of arrays
     
     \[
     \begin{bmatrix}
     \text{arr1}; \text{arr2}; \ldots; \text{arrn}
     \end{bmatrix}
     \]

**Note:** an scalar is also regarded as an array (of size \([1,1]\)).

**Note 2:** arrays must have matching sizes.

## Example (Concatenation)

\[
\begin{align*}
\text{\texttt{>> A}} &= \begin{bmatrix} 1,2;3,4 \end{bmatrix} \\
\text{A} &= \\
&= \begin{bmatrix}
1 & 2 \\
3 & 4
\end{bmatrix} \\
\text{\texttt{>> B}} &= \begin{bmatrix} \text{A};\text{A} \end{bmatrix} \\
\text{B} &= \\
&= \begin{bmatrix}
1 & 2 & 1 & 2 \\
3 & 4 & 3 & 4
\end{bmatrix} \\
\text{\texttt{>> C}} &= \begin{bmatrix} \text{A};\text{A} \end{bmatrix} \\
\text{C} &= \\
&= \begin{bmatrix}
1 & 2 \\
3 & 4
\end{bmatrix}
\end{align*}
\]
How to pre-allocate memory

Functions for pre-allocating memory include:

**colon (:)** – for linear sequences

**zeros** – all zero array of given size

**ones** – all one array of given size

**rand** – random array of given size (equally in $[0, 1]$)

To improve performance arrays should always be pre-allocated!

Example (Functions initializing arrays)

```plaintext
>> A = zeros(3,3);  >> x = 3:-0.5:1
>> A = ones(4,4,4);   x =
>> size(A)           3.0  2.5  2.0  1.5  1.0
ans =               >> A = ones(2)
    4   4   4       1   1
               1   1
```
How to write an m-file script?

1. open a text-editor of your choice or use the editor provided with Matlab
   >> edit myscript
2. Write all your calculations in a text-file with extension “.m” (here myscript.m)
3. Save file in the current working directory (or addpath to search path)
4. Call your script by calling it from the “Command Window”
   >> myscript;

Example (myscript.m)

```matlab
%m this is my first script. It displays a random number
random_number = rand(1);
fprintf('A random number: %1.4f: \n', random_number);
```
How to use basic syntax: if-clause

if-else block syntax:

```plaintext
if scalar_condition
    expressions
else
    expressions
end
```

Relational operators, eg.: `==` (equals), `||` (or), `&&` (and), `~` (not)

for details type: `help relop`

Example (if-else)

```plaintext
a = rand(1);
if a == 0.5
    fprintf('you are very lucky!');
end
```
How to use a for-loop

for-loop block syntax:

```matlab
for i = array
    % i==array(j) in the j-th loop
    expressions
end
```

(one can also use break and continue keywords)

Example (for loop)

```matlab
a=0;
for i = 1:100
    a = a+i;
end
```
How to index arrays

1. **Subscript of a matrix:**
   access the \((i,j)\)-th element of a 2D-matrix \(W\) of dimension \((m, n)\)
   \[\text{>> } W(i,j) = 1\]

   Note: The first index is always 1 (not 0 as in most other languages)

2. **Linear index of a matrix:**
   access the \((i,j)\)th element of the 2D-matrix \(W\) of dimension \((m, n)\)
   \[\text{>> } \text{linearidx} = i + (j-1) * m;\]
   \[\text{>> } W(\text{linearidx})\]

3. **“Slice” indexing with “:”**
   access the \(i\)-th row and \(j\)th column in \(W\), respectively
   \[\text{>> } w_i = W(i,:);\]
   \[\text{>> } w_j = W(:,j);\]

   get all elements as a concatenated column vector
   \[\text{>> } W(:)\]
How to index arrays (2)

4 Multiple indices
vectors of linear indices can be used

```matlab
>> W([1, 4, 5, 6])
```
access the 1st to 4th rows of the 2D-matrix $W$ of dimension $(m, n)$

```matlab
>> W(1:4,:)
```
access the 2nd $(m,n)$-slice of a 3D-matrix $V$ of dimension $(m, n, p)$

```matlab
>> W(:,:,2)
```

5 Logical indexing
logical matrices of the same size as $W$ can be used as index (very useful)

```matlab
>> W(W>0)
>> W(find(W>0)) = 1
```
Calculating with arrays

1. Element-wise interpretation
   - For instance, sin cos log etc.
   - Reserved symbols, .* ./ .^ etc.

2. “true” matrix interpretation (with dot product)
   - Symbols * / ^ etc.

3. Operations on one specified dimensions of the matrix
   - For instance, sum mean max etc.

4. Array manipulations
   - eg. reshape repmat permute circshift tril

Example (element-wise product and dot product)

```matlab
>> A = ones(2,2);
>> A.*A
ans =
   1  1
   1  1

>> A*A
ans =
   2  2
   2  2
```
Calculating with arrays is

- straightforward
- however, carefully check
  - the size of matrices
  - if element-wise or matrix-like operations are intended
  - which matrix dimension to operate on

Example (compute $y_i = Wx_i + b$ with $b, x_i \in \mathbb{R}^2$)

```matlab
>> W = [1,0.2;0.4,1];
>> b = [1;2] + 0.1;
>> x = 2*randn(2,1);
>> y = W * x + b
y =
    4.5535
    1.4856
```

```matlab
>> N = 5;
% same as b = [b,b,b,b,b]
>> bi = repmat(b,[1,N]);
>> xi = 2*randn(2,N);
>> xi(:,1) = x;
>> yi = W * xi + bi
yi =
    4.5535  2.3126  [...] -0.9021
    1.4856  6.8091  [...] -0.1080
```
The Gamma probability density function is defined as

\[ p(x|k, \theta) = \frac{1}{\Gamma(k)\theta^k} x^{k-1} e^{-\frac{x}{\theta}} \]

with shape \( k \) and scale \( \theta \) and \( x \in [0, \infty) \).

Example (Gamma random numbers)

```matlab
>> shape = 2;
>> scale = 0.003;
>> n = 1e7;
>> r = gamrnd(shape,scale,n,1);
>> size(r)
ans =
    1000000    1
>> hist(r,1000)
```
How to use sparse matrices

In MATLAB it is often more efficient (faster) to use sparse matrices instead of regular matrices if the majority of matrix elements are 0.

To generate a sparse matrix:

```matlab
>> W = sparse(W);
```

In general, the syntax for using sparse matrices is the same as for regular matrices.

**Example (Sparse matrix)**

```matlab
>> W = double(rand(100,100)>0.9);
>> Ws = sparse(W);
>> y = Ws*x; % the same as y=W*x but faster
```
Solution to Exercise #1

1. \%compute the Stirling Formula

2. \n = 50;

3. \n \_f a c t o r i a l = \text{sqrt}(2 * \pi * n) \times (n/\text{exp}(1))^n
Solution to Exercise #2

1. `function n_factorial = stirfac(n);`
2. `%compute the Stirling Formula`
3. `n_factorial = sqrt(2*pi*n)*(n/exp(1))^n;`