An open source Python code for evolving Schrödinger systems

Edgar Figueiras, David Olivieri
Computer Science Department, Universidade de Vigo. As Lagoas s/n, Ourense, ES-32004 Spain.

Ángel Paredes, and Humberto Michinel
Applied Physics Department, School of Aerospace Engineering, Universidade de Vigo. As Lagoas s/n, Ourense, ES-32004 Spain.

(Dated: October 11, 2017)

This note provides practical help on getting up and running with the Python code for simulating Schrödinger systems with the beam propagation method (BPM). We give detailed information concerning the command-line syntax for executing our code examples of physical systems discussed in the main text. Also, we describe the organization of the source code as well as key parameters so that users can write functions representing other physical systems of interest. Finally, for those researchers uninitiated in Python, we provide a brief FAQ and a section about installing Python through Anaconda on all popular operating system platforms. For producing animations of the simulations, we also provide instructions for installing the `mencoder` utility.

I. GETTING STARTED

This material is part of a paper submitted to AJP; please refer to it for the context of the examples and if you use this software in a publication. In the main text of the article, several usage examples of the beam propagation method (BPM) simulation code for teaching and research in different areas of physics have been described. The goal of this brief note is to explain how the files are organized and how the BPM code should be run.
Availability: the source code is available at https://github.com/pyNLSE/bpm

We have also prepared video tutorials that accompany this note. The videos can be found at the following links:

- Installing on MacOS: `https://youtu.be/ErNEuouwrrg`
- Installing on Windows7: `https://youtu.be/0-1xoPWXZE0`
- Installing on Linux: `https://youtu.be/lTjbssbg_0A`

A. Prerequisites

This software code was written in Python 3, and uses a number of standard modules (numpy, os, matplotlib, importlib, and other system related modules). The software runs on major operating systems - Linux, MacOS, and Windows - and only requires a minimum setup effort described in this note. Also, while there are no minimum memory or processor requirements, the computational speed does depend upon the problem size and the discretization.

In order to generate the video outputs, the `mencoder` utility and associated libraries are used. This utility is available on all operating system platforms and must be installed separately (details of the installation process are described in section VI of this note).

B. Input and output files

The main file is `bpm.py` and is executed from the command-line (in the same directory where the bpm.py script is found) with two required options:

```
python3 bpm.py <example> <dimension>
```

or (if python is configured to point to the python3 binary)

```
python bpm.py <example> <dimension>
```

where the `<example>` is one of the physical situations specified within a given example file (described in the text), and `<dimension>` specifies the problem is either one-dimensional (1D) or two-dimensional (2D) (called by functions stored in `1D.py` or `2D.py`). For example, to simulate the one-dimensional *double well* example, the command would be:

```
python bpm.py Double_Well_1D  1D
```

(executed in the same directory where the `bpm.py` script is found) and where `Double_Well_1D` indicates the name of the example file (found in the `examples1D`) that will be loaded. All other examples are found in their respective directories `examples1D` or `examples2D`, and are loaded in a similar way as above through the `<example>` command-line option.

A quick comment about the organization of the files in terms of directories is helpful. The files `bpm.py`, `1D.py`, and `2D.py` must be in the same top-level directory, together with subdirectories `examples1D` and `examples2D`. Depending upon the dimension, the example files must be inside one of these subdirectories. All the output for a particular simulation example is deposited in a subdirectory with the same name as the example file name.

From the examples distributed with our BPM software, the valid options for `<example>` in 1D are the following:

```
Diffraction_Slit_1D
Double_Well_1D
Interference_Gaussians_1D
Rectangular_Barrier_1D
Sech2_Pot_1D
Soliton_Emission_A_1D
Soliton_Emission_B_1D
Solitons_in_phase_1D
Solitons_phase_opp_1D
Thomas_Fermi_1D
```

while the valid options for `<example>` in 2D are:

```
Collapse_2D
Diffraction_Circle_2D
Filamentation_2D
Gaussian_Beam_2D
Gaussian_Vortex_interf_2D
Liquid_Droplet_2D
Vortex_2D
Vortex_Breaking_2D
Vortices_Pattern_2D
Vortex_Precession_2D
```

During the computation, graphical output is generated by the `matplotlib` library. Depending on the dimension, the output is as follows:

- In one dimension, a plot of $|\psi(x)|^2$ is generated for each value of t, stored as a `.png` file. When the computation finishes, a video is created from the collection of `.png` files using the `mencoder` utility and a contour plot for $|\psi(x,t)|^2$ is also created.
- In two dimensions, a contour plot of $|\psi(x,y)|^2$ and a plot of $|\psi(x,y=0)|^2$ are generated for each values of t, stored in separate `.png` files. Upon completion of the calculation, two videos are created from the stored `.png` files. A contour plot for $|\psi(x,t)|^2_{y=0}$ is also created. Note that the $y=0$ cuts are not always useful but they have been included because in some examples they can be illustrative.

When the computation is done, the final value of ψ is written to the `<example>.npy` file. This is useful if the user wants to continue a computation for values of t beyond the one initially specified. In that case, the user can just load the `.npy` as the initial condition for the subsequent computation.

II. GENERATING AN EXAMPLE

New and different physical examples, beyond those provided, can be constructed by the user without modifying the `bpm.py`, `1D.py` and `2D.py` files. In this case, the user would construct a python script file similar to those found in the example directories, by specifying computational parameters. The twenty cases provided within the supplementary material cover a large set of possibilities and serve as examples for setting up other physical situations.

A. Computational parameters

There is a relatively small set of input parameters and functions needed to specify a simulation of a physical system.

- `Nx, Ny`: These are the number of grid points for the spatial discretization in each direction. Some caveats should be noted. First, while `Ny` is not used in the one-dimensional case, it must be assigned a default value. Next, although larger `Nx, Ny` values would produce more precise calculations (at the cost of computational speed), in some cases, large values can produce numerical instabilities due to specifics of the algorithm. From a practical computational perspective, very large `Nx, Ny` can lead to stack overflow failures due to computer memory limitations (or at the least, a deleterious situation leading to *thrashing*, where virtual pages are constantly swapped between memory and disk).
- `dt`: This is the value of t corresponding to the fundamental time-step of the algorithm. Typically, a smaller dt value would produce higher precision, however it will also make the computations slower. There is a point of diminishing returns, since values that are too small could degrade accuracy due to the numerical machine precision.
- `tmax`: This is the maximum time, t, that the simulation will run before the computation finalizes.
- `xmax, ymax`: These specify the spatial window of the computation: $x \in$ `[-xmax, xmax)`, $y \in$ `[-ymax, ymax)` (notice that `ymax` is not used in the one-dimensional case but it must be assigned a default value).
- `absorb_coeff`: This controls the boundary condition needed by the Fourier method. By default (`absorb_coeff`=0), the Fourier method automatically implements periodic boundary conditions and the wave traversing a boundary of the computational boundary enters through the opposite one. In some situations, this is inconvenient because it introduces spurious effects in the physical description. In order to avoid this problem, it is customary to introduce an absorbing potential at the edge of the boundary that (approximately) cancels out the energy approaching the borders. The value of the `absorb_coeff` parameter is the strength of this *sponge*.

B. Output parameters

These parameters determine various aspects of the graphical output:

- `images`: This parameter determines the number of plots to be output. The plots are created with uniform time intervals between 0 and `tmax`. The total number of plots is `images` + 1.
- `output_choice`: This parameter determines whether images should be displayed or not. When `output_choice`=1, the images are displayed on-screen, if (`output_choice`=2, the images are stored in the output directory. For both storing and displaying images on screen, the user should select `output_choice`=3.
- `fixmaximum`: This parameter fixes the maximum amplitude scale of plots for $|\psi|^2$. If it is a positive value, the plot range of $|\psi|^2$ is between 0 and `fixmaximum`. Fixing the scale is necessary when plots of $|\psi|^2$ at different t are combined (for instance, in a video). If `fixmaximum` = 0, the range of $|\psi|^2$ is determined automatically for each plot.

C. Physical input

The physical input for a particular problem consists of the initial condition $\psi(\mathbf{x}, t = 0)$, as a function of the spatial coordinate(s) and the potential, which can depend on the spatial coordinate(s), on time, and for nonlinear cases, on the wave function itself. Within the example file, these are specified in the functions `def psi0(x,y):` and `def  V(x,y,t,psi):`. For the one-dimensional cases, the y coordinate is inconsequential.

III. THE BPM.PY FILE

This is the main file that controls the computation. The code is organized in four parts:

- Preliminaries: First, the required modules are imported, the global variables are set, and the directories are managed (creating the output directory if it doesn't already exist). Also, if any files exist from a previous run, then a command eliminates the existing `.png` files in the output directory in order to avoid possible confusion.
- Initialization of the computation: The example file (specified by the `<example>` option) is imported and used to generate all the objects needed in the computation, namely the spatial grid, the initial condition, the Laplacian operator, etc.
- Main computational loop: At each step of the loop, $\psi(t)$ evolves in time to $\psi(t+dt)$. At the appropriate steps of the computations, the suitable functions of the `1D.py/2D.py` are called to generate the output. When t reaches `tmax`, the loop completes.
- Final operations: The final value of ψ is saved and the final output is generated.

This file has been kept as simple and minimal as possible for clarity. Intentionally, advanced features of Python (such as classes, inheritance, and more efficient numerical libraries) have been avoided. This decision was made because we assumed minimal Python programming exposure by the students and we want to focus on the algorithms themselves without other programming concepts. Also, considerable effort has been made to make the same code cross-compatible across Unix and Windows systems. There are certainly many interesting features that can be added, depending on the preferences and needs of the user.

IV. THE 1D.PY AND 2D.PY FILES

These files contain some functions that are called from the main file. They are required for the implementation of the algorithm and the generation of the output.

Both `1D.py` and `2D.py` include the same content, adapted in each case to the dimension of the computation. In fact, both could be merged in a single file, but they have been kept separate to make their understanding and editing simpler for the interested user. Notice that the `1D.py` still specifies the y-coordinate, even if it is immaterial.

A brief explanation of the scope of the functions ensues:

- `grid`: Defines the spatial grid used in the computation.
- `L`: The Laplacian operator in Fourier space, used to compute the evolution associated to the kinetic term.
- `absorb`: Generates the absorbing term near the borders of the computational window.
- `savepsi`: In order to produce the final contour plot, some intermediate data must be kept for certain values of t. The stored data are $|\psi(x)|^2$ and $|\psi(x, y = 0)|^2$ for the one and two-dimensional cases, respectively.
- `output`: Generates the plots at intermediate steps of the computation.
- `final_output`: When the algorithm has ended, it generates the final contour plot, the .avi animations, and a data file with the final value of ψ.
- `movie`: It is called from `final_output` to create the movies.

V. FREQUENTLY ASKED QUESTIONS

Users of other scientific languages might initially find some of the details of Python puzzling when first getting started. We provide a brief list of answers to some some common questions that may arise.

a. Q1: I am new to Python. Is it similar to Matlab/Octave? Where can I get some basic information about the Python language? Also, what are the advantages of using Python for scientific computing?

- For Matlab/Octave users, transitioning to Python is quite easy. First, Python is interpreted (meaning that no compilation is required) and functions are organized into scripts. Numerical operations and functions are also quite similar, both through the numerical package, `numpy` or the mathematics package `math`. A useful resource for comparing differences between Matlab/Octave array operations can be found at: https://docs.scipy.org/doc/numpy-dev/user/numpy-for-matlab-users.html

- There is extensive online documentation for Python (https://www.python.org/doc/). It is also worth mentioning that there is a very active community of Python users and developers so that answers to many questions can be found in online forums.
- Python has several advantages over other languages for computational problems. Foremost, it is a full-featured language with a large and growing list of open-source add-on modules. Python has the advantage of being interpreted for rapid development, but retains powerful features of core languages (such as C/C++), such as object-oriented, inheritance, and flexible but extensive data types. For raw speed, critical sections can be coded and compiled into modules with the C-language Python API.

b. Q2: How are functions in Python called? How are functions from Python modules called?

- Defining and calling functions in Python is *as easy as pie.* Use the keyword `def` and the name of the function you want to call (e.g., `def func()`. This function can either return a value (using the keyword `return`) or not (in which case it acts as a procedure. When calling the function, we simply call it by name (e.g., for the function above, that returns a single value, we might write: `x = func()`). A far more extensive documentations can be found at the main Python page, or other online material.
- The Python language is designed to be modular and extensible. For this reason, the Python core interpreter incorporates very few functions for basic operability. The core functionality is extended via modules, which can be other python scripts or compiled binaries (in the form of shared objects). To install a module for an interactive session or for running a script, modules are imported (e.g., `sys` and `os` for system and operating system commands, and `numpy` for numerical functions). The namespace of the module follows a simple calling procedure with a dot invocation, i.e., when calling a function, the name of the module is written, followed by a dot and the name of the function (e.g., when calling the absolute value function from the numpy module, we would write `numpy.abs()`).

c. Q3: I notice that Python syntax does not have control brackets. How does one define a loop or other control structure in Python?

- The structure of Python codes relies on the indentation for the organization of nested functions, loops, etc. Thus, it is fundamental to keep the indentation correct. In particular, all indents should be a single tabulator or four spaces. Notice that both options can be used, but in a consistent way: it is not possible to combine them within a code. If a TabError occurs, it is probably because of this issue.

d. Q4: An important part of my numerical code is the array index ordering. How is this handled in Python? Is it similar to Matlab or C?

- The indexing within vectors and arrays starts in 0, as opposed to some other languages where it starts with 1. For instance, if `a=[5,8]` is defined, `a[0]` is 5, `a[1]` is 8 and `a[2]` produces an "IndexError".

e. Q5 :Why are there different versions of Python? Will both versions work with the BPM code?

- The differences between the two versions python2.7 and python3.X have to do with library support. For practical purposes of a beginner, the syntax is nearly identical. There is plenty of online information about the differences between these two versions, however this blog article succinctly summarizes key differences: `http://sebastianraschka.com/Articles/2014_python_2_3_key_diff.html`
- Our BPM code was written for Python 3.6. Nonetheless, with minor changes (mostly to the print function), this code would equally work with Python2.7.

VI. STEP-BY-STEP TUTORIAL FOR GETTING PYTHON UP-AND-RUNNING

This tutorial describes installing Python with Anaconda, a popular distribution that allows multiple versions to be installed on the same machine without collision or interference from other system wide libraries. The distribution is available in the Linux, MacOS, and Windows. Anaconda provides a turn-key Python system with most of the standard libraries already configured and installed. Apart from this, it allows for scalability and virtual containers for managing many different versions of packages. Anaconda also provides a convenient text editor for Python files called Spyder.

A. Getting and installing Anaconda in Linux

The great thing about installing Anaconda is that it does not require root privilege. This means that it is completely portable and can be installed to the home directory on any machine, making it ideal for a student laboratory setting. The steps for installing Anaconda are the following.

1. Go to the download page `https://www.anaconda.com/download/` and get the appropriate version. The `bpm` code uses version 3.x, so you will need to *download* the version 3.x (now 3.6 at time of writing). Also, you should download the appropriate pre-compiled binaries corresponding to your architecture: whether your machine is 64bit or 32bit. To find out what your architecture is, execute the following command:

   ```
   uname -a
   ```

   which will either give `i686` or `i386` if system is 32-bit, or 64-bit (`x86_64`).

2. Execute the installer from the command-line. Open an x-terminal and change directories to the downloaded file. The file that you downloaded is a bash script that contains the install commands as well as binary packed Python packages. Executing this binary will search your system's configuration and libraries in order to check dependencies, unpack the files within its own file, and download additional packages or unsatisfied dependencies.

   To execute, you could either execute it with the `bash` command:

   ```
   bash Anaconda3-6.0.0-Linux-x86.sh
   ```

   or, you could convert it to executable and then run it directly:

   ```
   chmod +x Anaconda3-6.0.0-Linux-x86.sh
   ./Anaconda3-6.0.0-Linux-x86.sh
   ```

   After approving the license agreement and setting the install path, the installation should continue to the end. It will finally ask you whether you want to modify your `.bashrc` file. A good idea is to say `yes`, since a bash command will be inserted that modifies your default `PATH` variable in order to include the route to the anaconda bin directory. Now, the next time you open an x-terminal, this `PATH` variable is automatically set.

3. First execution: In the x-terminal that you used to install, you must *source* the `.bashrc` file (found in your *home* directory). For this, simply type:

   ```
   source .bashrc
   ```

   Instead of doing this, you could also just open a new x-terminal and the `PATH` will automatically read the `.bashrc` file to set the `PATH` to the route of the Anaconda bin directory. After the install is complete, you can test to make sure you are running Anaconda by typing `python` at the command line.

   ```
   python --version
   Python 3.6.0 :: Anaconda custom
   ```

4. Install and execution of BPM software. Once you have installed Python with Anaconda, the BPM software should be unpacked into a directory of your choice and the examples are executed in an x-terminal from the same directory where the `bpm.py` is found. An example x-terminal session is as follows:

   ```
   $ cd Desktop
   $ unzip bpm_code.zip
   $ cd bpm_code/
   $ ls
      1D.py  2D.py  bpm.py  examples1D  examples2D
   $ python bpm.py Thomas_Fermi_1D 1D
   ```

   As indicated in the above example, all the code examples are executed from the top level `bpm_code` directory.

B. Getting and installing Anaconda in MacOS

For MacOS, the same procedure as described above for installing Anaconda on Linux using the command-line can be followed.

There is also a graphical installer that guides the entire process. Detailed notes can be found here: `https://docs.anaconda.com/anaconda/install/mac-os#macos-graphical-install`

Once the installation is complete, follow the instructions above for creating a directory for the `bpm` code. Make sure that when you open an x-terminal, the version of python is 3.X from the Anaconda directory.

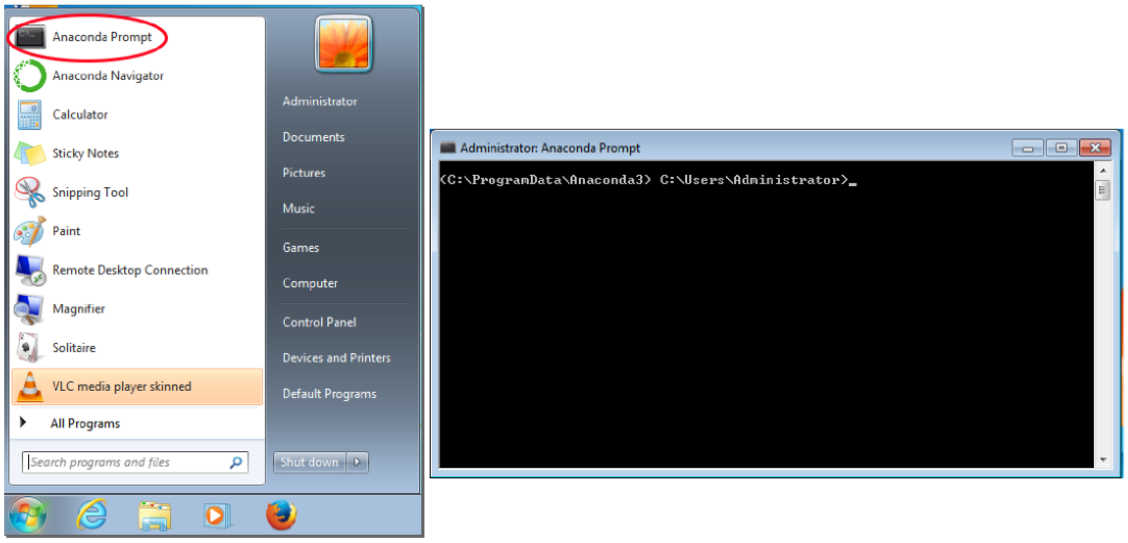
C. Getting and installing Anaconda in Windows

The steps for installing Anaconda in Windows are slightly different. A word of caution, the Anaconda environment only works on Windows versions 7 or higher.

1. Go to the download page `https://www.anaconda.com/download/` Once again, download the version 3.x (now 3.6 at time of writing) corresponding to your machine architecture: either 64bit or 32bit.

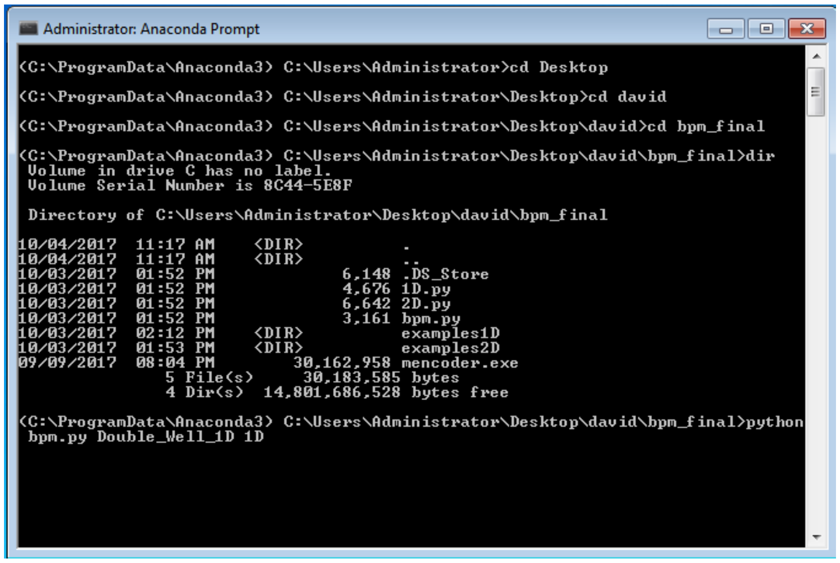
2. Execute the binary to install the Anaconda distribution on your system. After the installation, your main Windows menu should have entries for the Anaconda Navigator.

3. The command line tool. The Anaconda application provides an x-terminal, called the Anaconda Prompt, similar to what one would use on Linux. This terminal is convenient for running the `bpm.py` software using the format specified throughout this note and the main article. The terminal should look like this when you select the Anaconda Prompt from the Windows menu.



```
(C:\ProgramData\Anaconda3) C:\Users\Administrator>_
```

4. Install and execution of BPM software in Windows with Anaconda. Once the you have installed Python with Anaconda, the BPM software should be unpacked into a directory of your choice. Next, open the Anaconda Promt (found from your main menu), as shown above. From the Anaconda Prompt terminal, change the directory to the same directory where the `bpm.py` is found. Here is a sample session from the Anaconda Prompt window:



```
(C:\ProgramData\Anaconda3) C:\Users\Administrator>cd Desktop

(C:\ProgramData\Anaconda3) C:\Users\Administrator\Desktop>cd david

(C:\ProgramData\Anaconda3) C:\Users\Administrator\Desktop\david>cd bpm_final

(C:\ProgramData\Anaconda3) C:\Users\Administrator\Desktop\david\bpm_final>dir
 Volume in drive C has no label.
 Volume Serial Number is 8C44-5E8F

 Directory of C:\Users\Administrator\Desktop\david\bpm_final

10/04/2017  11:17 AM    <DIR>          .
10/04/2017  11:17 AM    <DIR>          ..
10/03/2017  01:52 PM             6,148 .DS_Store
10/03/2017  01:52 PM             4,676 1D.py
10/03/2017  01:52 PM             6,642 2D.py
10/03/2017  01:52 PM             3,161 bpm.py
10/03/2017  02:12 PM    <DIR>          examples1D
10/03/2017  01:53 PM    <DIR>          examples2D
09/09/2017  08:04 PM        30,162,958 mencoder.exe
               5 File(s)     30,183,585 bytes
               4 Dir(s)  14,801,686,528 bytes free

(C:\ProgramData\Anaconda3) C:\Users\Administrator\Desktop\david\bpm_final>python
 bpm.py Double_Well_1D 1D
```

Once in the bpm.py directory, you can execute the commands as indicated in previous sections. Note, when executing the code, you must always be in the main bpm_code directory (where the bpm.py file is located).

D. Making movies from outputs in Linux, MacOS, and Windows

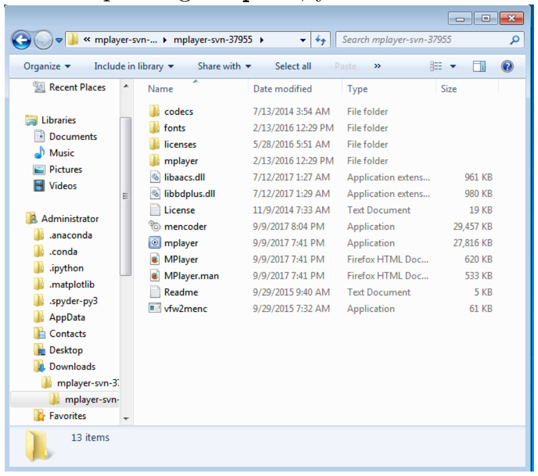
Animations are created by combining the .png files using the mencoder utility, which is part of the MPlayer utility. In Linux or MacOS, these are obtained as binaries from the standard repositories. For example,

- in Linux, the MPlayer utility can be installed with apt-get: with the command: `apt-get install mplayer`; the mencoder utility and library dependencies are installed as well.
- in MacOS, the MPlayer utility can be installed via Homebrew (`brew install mplayer`), where once again, the mencoder binary with dependencies are also installed.

In Windows, there are some additional steps that must be done. First, obtain the version of the mencoder corresponding to your architecture at:

`http://mplayerwin.sourceforge.net/downloads.html`

After unpacking .zip file, you should have a directory that looks like the following:



Name	Date modified	Type	Size
codecs	7/13/2014 3:54 AM	File folder	
fonts	2/13/2016 12:29 PM	File folder	
licenses	5/28/2016 5:51 AM	File folder	
mplayer	2/13/2016 12:29 PM	File folder	
libaacs.dll	7/12/2017 1:27 AM	Application extens...	961 KB
libbdplus.dll	7/12/2017 1:29 AM	Application extens...	980 KB
License	11/9/2014 7:33 AM	Text Document	19 KB
mencoder	9/9/2017 8:04 PM	Application	29,457 KB
mplayer	9/9/2017 7:41 PM	Application	27,816 KB
MPlayer	9/9/2017 7:41 PM	Firefox HTML Doc...	620 KB
MPlayer.man	9/9/2017 7:41 PM	Firefox HTML Doc...	533 KB
Readme	9/29/2015 9:40 AM	Text Document	5 KB
vfw2menc	9/29/2015 7:32 AM	Application	61 KB

Next, copy the file mencoder.exe to the directory where the software bpm.py is located. That is all! The bpm.py software will take care of everything else for producing the animations with mencoder.