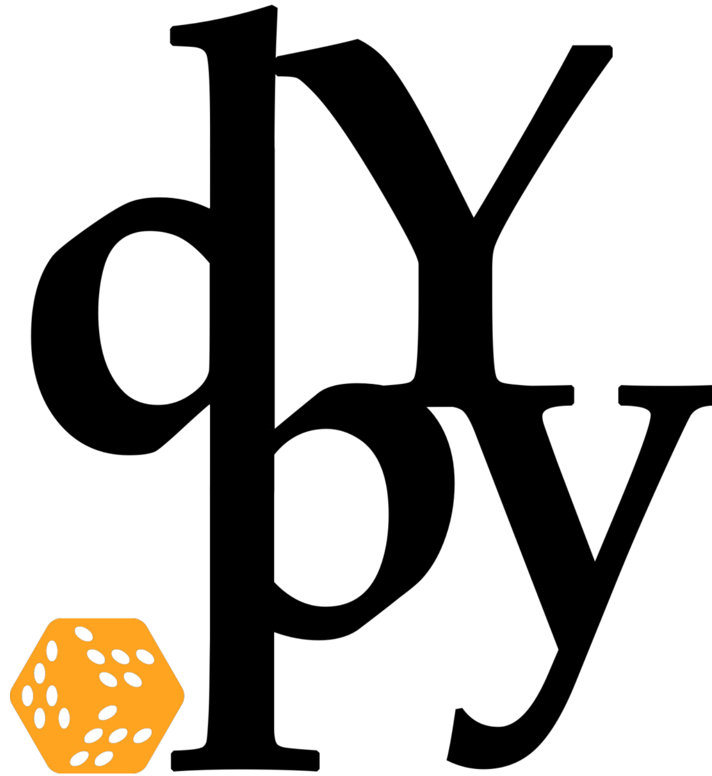

dypy: Dynamical Systems in Python

PHY250 • Project Report • Spring 2008

SOPHIE ENGLE AND SEAN WHALEN
Department of Computer Science
{sjengle,shwhalen}@ucdavis.edu

Abstract: This project introduces **dypy** – an extensible Python framework for visually exploring dynamical systems. The framework allows for new dynamical systems to be added to the tool, as well as new methods for visualizing those systems. It additionally provides a powerful gui component, allowing on-the-fly changes to the visualization parameters. While still under construction, **dypy** already contains over 1600 lines of code, with over 800 lines dedicated to the gui.

1 Introduction

In this course, we have seen and created numerous tools for visualizing dynamical systems. Many of these tools illustrate the power of visualization and simulation. However, these tools are implemented on a wide array of platforms and programming languages and sometimes lack graphical user interfaces.

Our goal is to integrate some of these ideas and tools we've seen in class into a single, extensible cross-platform framework. We chose Python since it is cross-platform and used by many scientists in the field. We additionally chose to use OpenGL to allow for hardware-accelerated three-dimensional visualizations. Ideally, this will allow anyone with minimal Python experience the ability to extend the framework to new systems and visualization tools without having to do extensive gui programming. This allows the focus to be on the exploration of new systems.

The result of our work is **dypy** (named for **dy**namical systems in **py**thon), which is both a tool and a custom Python package. While it is a work in progress, **dypy** supports dynamically loaded systems and visualization tools, and an interactive gui allowing for visualization parameters to be changed in real-time.

2 Background

The visualization tools we chose to first integrate into **dypy** focus on a Monte-Carlo animation approach to traditional visualizations such as orbit diagrams or phase portraits. Instead of presenting a static picture, we animate the iteration of these systems based on a randomized set of initial states.

Our first implementation of these ideas was in Processing, an open source programming language and environment based on Java. This implementation can be found at:

`http://www.node99.org/projects/bifurcation/` *or*
`http://www.phien.org/wiki/index.php?title=Logistic_Map_Bifurcation`

OpenGL is the industry standard environment for developing 2D and 3D graphics applications. OpenGL has standard language bindings for languages such as C++ and Java, and can be used in Python through the Pyglet package.

For gui programming, the package wxPython provides a Python interface to the open-source, cross-platform wxWidgets C++ class library, which provides gui applications with a native look and feel.

3 Methods

We used Python v2.5, Pyglet v1.0.1, wxPython v2.8.4, and Numpy v1.0.4 to implement our framework. We developed the code under Windows and Macintosh systems.

We created a `dypy` Python package with separate subpackages/folders for systems, demos, visualization tools, and the gui components. The package structure is as follows:

```
dypy            <- main package
    demos       <- subpackage containing demos named SystemName__Demo#.py
    gui         <- subpackage containing general gui components
    images      <- internal folder containing toolbar and logo images
    systems     <- subpackage containing systems named SystemName.py
    tools       <- subpackage containing tools named ToolName.py and ToolNameGUI.py
```

To start **dypy** we execute the following Python code:

```
import dypy
dypy.show()
```

To allow for new systems, demos, and visualization tools to be added to the **dypy** framework, we had to support dynamic loading of modules. Most of this code is in the file `utils.py` in the `gui` folder of the `dypy` package. For example, to add a new system to the package, create a `Map` or `ODE` class for that system and place the module file in the `dypy/systems` directory. Example code for the dynamic loading can be found in figure 1.

Each tool is made of two components, and hence two different Python modules. The first module must extend the `DynamicsTool` class. Each `DyanmicsTool` contains a `DynamicsWindow` class which automatically handles setting up the OpenGL visualization window in Pyglet. The second module must contain a `wx.Panel` class which provides a gui interface to that tool. The **dypy** gui loads this panel when necessary.

The gui code is separated into 17 different modules, and handles displaying and updating all the necessary system and visualization parameter settings. The major gui classes include:

Module	**Description**
`MainPanel.py`	Displays widgets allowing user to select a system, demo, and tool dynamically loaded from the package.
`SystemPanel.py`	Displays widgets allowing user to select the state (x, y, z) ranges and parameter ranges to explore, adjusting to the system dimension as necessary.
`MainWindow.py`	Displays the primary **dypy** window.

We discuss the implementation of systems in the next section.

4 Dynamical Systems

We implemented 11 dynamical systems in our framework. This includes:

1. Cosine Map
2. Cubic Map
3. Cusp Map
4. Exponential Map
5. Henon Map
6. Logistic Map
7. Lorenz ODEs
8. Neuron Map
9. Rossler ODEs
10. Standard Map
11. Tent Map

To implement these systems, we created two classes: a `Map` class and a `ODE` class. Every system implemented inherits one of these two classes. The `Map` class includes functions to iterate the map, return the derivative, and get the state and parameter ranges and names for the map. The `ODE` class is almost identical, except for a Runga-Kutta integrator. Both classes use arrays to allow for systems of any dimension.

5 Results

The resulting framework consists of over 1600 lines of code, with over 800 lines of code dedicated to just the gui. We implemented 11 systems and two visualization tools in the gui. The gui allows users to configure system and visualization parameters before and during visualization, and automatically adjusts to the necessary number of dimensions for the system.

However, the framework is still incomplete. The toolbar, allowing the user to load a system, demo, or tool from a Python file outside the **dypy** package is not functional. The demo concept is not yet implemented, some usability improvements can be made to the gui, and some code optimization is possible.

We found the implementation of wxPython to vary across systems – making the gui almost unusable on some Mac systems. This creates an obstacle to making **dypy** a true cross-platform framework. Additionally, moving from the Java Processing environment to Python, we found some visualization tools ran significantly slower under Python and Pyglet.

Figure 2, 3, and 4 demonstrate the **dypy** gui and visualization tools.

6 Conclusion

dypy is an extensible Python package allowing for users with Python experience to easily add new systems to explore. With slightly more experience, users can even add new visualization tools to **dypy** .

However, the interface between Python and the various C++ based packages was cumbersome. This especially affected the gui programming. Trying to the gui interface in wxPython to behave identically across different platforms has been a challenge.

Having a gui interface increases interactivity with the visualization, allowing the user greater flexibility when exploring new dynamical systems. However, the amount of code required by the gui interface in relation to the total amount of code for all of **dypy** illustrates the need for a framework which allows users to avoid messy gui programming. While we were unable to fully complete the gui for **dypy** , we have shown such a framework is possible in Python.

7 Bibliography

None.

8 Figures

```
from systems/__init__.py:
    import dypy, os, os.path

    def getall():
        path = os.path.join(dypy.__path__[0], "systems")
        excludes = [ "__init__", "Map", "ODE" ]
        names = [f[0:-3] for f in os.listdir(path) if f[-3:] == ".py"]

        for e in excludes:
            if e in names:
                names.remove(e)

        return names

    __all__ = getall()

from gui/utils.py:
    import dypy.systems

    def get_classes(modules):
        classes = []

        for module in modules:
            class_name = module.__name__.split('.')[-1]
            exec "current = %s.%s()" % (module.__name__, class_name)
            classes.append(current)

        return classes

    def get_systems():
        names = dypy.systems.__all__
        names = ["dypy.systems." + name for name in names]
        names.sort()

        return get_classes(get_modules(names))
```

Figure 1: Example Python code which allows for systems to be dynamically loaded at startup.

Figure 2: A screenshot showing both the dypy gui and the visualization window, displaying a Monte-Carlo animated orbit diagram for the Logistic Map with x ranging from 0 to 1 and r ranging from 3.5 to 4.0.

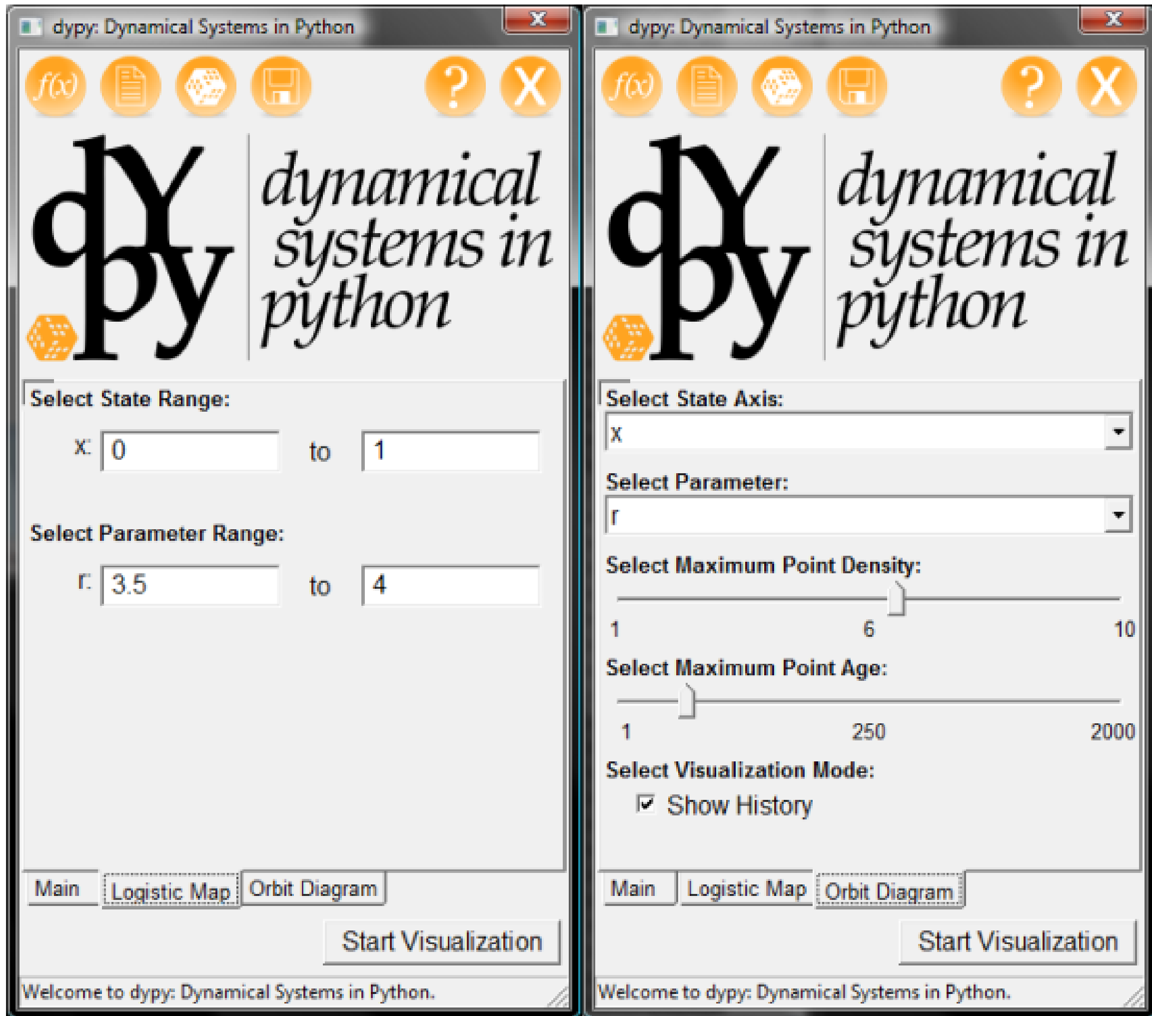

Figure 3: A screenshot showing the other tabs of the dypi gui. The "Logistic Map" tab is automatically generated based on the system selected in the "Main" tab. The "Orbit Diagram" tab is automatically generated based on the visualization tool selected in the "Main" tab. Ideally, the demo selected would automatically populate the parameter values in the "Logistic Map" tab. However, demos have not yet been implemented in dypy.

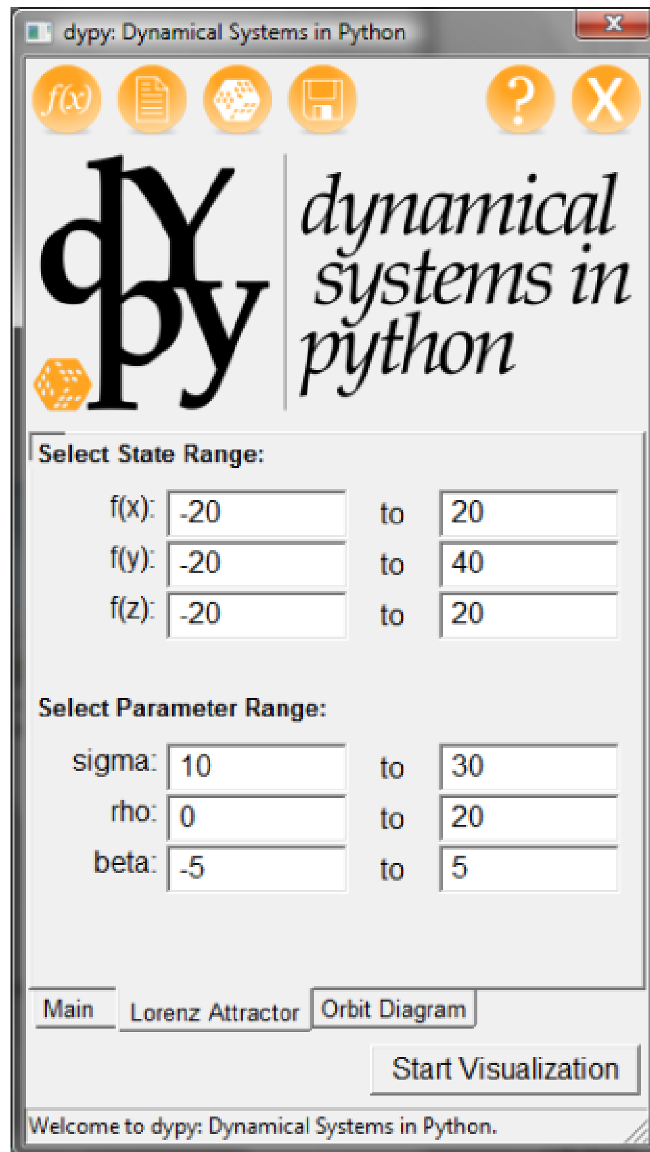

Figure 4: dypy is designed to allow for systems with any number of dimensions. In this screenshot, we show how the dypy gui automatically adjusted to the 3-dimensional Lorenz odes.