Designing interactive webapplications for mathematical visualization

Benjamin Rüth
Technische Universität München
Email: benjamin.rueth@tum.de

July 30, 2015

We developed a flexible open–source based workflow with the specific application of webdriven, platform independent, interactive applications for mathematical visualization in the context of university education. Our main goal is to realize mathematical visualization with no requirements of programming skills or program installation on the user–side. The Python plotting library Bokeh plays a crucial role in our approach.

1. Introduction

We wanted to setup interactive webapplications for the visualization of math content, specially in the context of the math lecture for mechanical engineering students. The goal was to have an environment where as few user interaction as possible is necessary to get the application running and with as less visible code as possible. Both restrictions are in our opinion very important, because a non smooth start–up of the application often frustrates the user and especially undergraduate students often do not want or are not able to program (even few lines of code), sometimes even seeing code results in the user immediatelly leaving the webpage. Therefore we put up the following constraints to our application:

- no installation or log–in required
- no visible code and no necessity to access or modify code
- use of open source software
- interactive modification of math visualizations
- high flexibility with respect to plotting and interaction

Before starting this project we have used videos generated in MATLAB for visualizing mathematical content. At some point we found out that many topics cannot be visualized properly by creating a video. Often it has a much higher effect, if the user is able to play around with parameters and try out different combinations of parameters on ones own.

Our MATLAB videos were — obviously — not able to fulfil our constraints. Also other environments like "Wolfram CDF Player" (huge installation, not open source), "IPython Notebook" (too much code interaction) or "Geogebra" (only basic scripting language, not flexible) do not fulfil our criteria and turned out to be not appropriate for that specific task.

The Python framework is in general very flexible, open–source and also extendible, since many scientific applications base on Python. After some research we discovered the plotting library Bokeh, which is similar to matplotlib, but also supports interactive manipulation of the plots with different kinds of widgets as well as running these interactive plots on a server. In the following we want to describe our workflow for setting up an interactive webapp using Bokeh as well as the other used tools.

2. Installation

The Framework Anaconda turned out to be a good choice for the installation of all the necessary python modules. The installation of Anaconda is described at

http://continuum.io/downloads

After the installation of Anaconda one can install Bokeh with the following commands:

```
$ sudo conda install bokeh
```

If installation fails due to access rights create environment.

```
$ conda create -n my_root
$ sudo activate my_root
```

Exit environment with

```
$ source deactivate
```

For development one can use the IDE spyder. Running Spyder from `sudo` often causes problems, don't do this! If problems with the libraries occur run

```
$ spyder --reset
$ spyder
```

3. The idea

Our goal was to implement an interactive Fourier–Series app. The Fourier–Series of a 2π–periodic function $f(x)$ is defined in the following way:

$$f(x) = \frac{a_0}{2} + \sum_{k=1}^{\infty} \left[a_k \cos(kx) + b_k \sin(kx)\right],$$

with

$$a_k = \frac{1}{\pi} \int_0^{2\pi} f(x) \cos(kx)\,\mathrm{d}x \quad \text{and} \quad b_k = \frac{1}{\pi} \int_0^{2\pi} f(x) \sin(kx)\,\mathrm{d}x.$$

This representation is exact, if one uses infinitely many coefficients a_k, b_k. If one truncuates the series after n coefficients, such that we set $a_k, b_k = 0 \forall k > n$ we do not get an exact representation of $f(x)$, but only an approximation:

$$f(x) \approx \frac{a_0}{2} + \sum_{k=1}^{n} \left[a_k \cos(kx) + b_k \sin(kx)\right].$$

We now wanted to implement a app, where the user has the possibility to choose the value of n and can therefore interactively control the accuracy of the approximation of $f(x)$. An additional goal is to visualize the analytical representation of the fourier series at the same time, such that the user can see different phenomena:

- The higher we choose n the longer our analytical expression is.
- Even functions (symmetrical to the y–axis) produce only cos terms in the series expansion, odd functions (point–symmetrical to the origin) produce onyl sin terms in the series expansion.
- The coefficients are decreasing along the series.

Of course one can also proof all these properties, but seeing them "in action" often has a bigger effect.

4. Implementation

The mathematical functionality has been implemented in the file `fourierFunctions.py`; these functions just supply us with some example functions $f(x)$ and functions for the calculation of the coefficients and the evaluation of the fourier series. We will not explain these in detail.

The dynamic generation of LaTeX strings depending on the degree of the fourier series and $f(x)$ is done in `fourierTex.py`.

The most important part of our implementation is in `fourierApp.py`. This file sets up the web app and uses the previously mentioned files to correctly handle user input.

These functions are only able to publish our app directly on the Bokeh plotting server. For more elaborate functionality (like the dynamic update of LaTeX strings) we feed the output of the Bokeh plotting server to a Flask server(which is able to handle to LaTeX strings sent via an request). The LaTeX strings are finally embedded into the html file via MathJax. All source files can be found in the appendix as well as on GitHub: https://github.com/BenjaminRueth/Visualization

5. Running the App

For running the app one firstly has to start the Bokeh plotting server via

```
$ bokeh-server -m --backend=memory
```

Then one has to run the script starting the App on the plotting server

```
$ python fourierApp.py
```

One can now access the App at the plotting server. For enabling displaying of dynamically generatex LaTeX strings one has to run

```
$ python fourierFlask.py
```

for starting the Flask server. The FourierApp can now be accessed from any Browser.

6. Conclusions

With Python and Bokeh we have the possibility to realize a large field of visualizations. Additionaly to sliders, Bokeh also supplies many other Widgets like buttons, checkboxes, dropdown–lists etc. as well as many different kinds of 2D plots. Embedding These Bokeh plots in a Flask environment gives us additionaly the possibility to modify the html content directly from our app. Currently we are working on running this app on a server as well as developing more interactive visualizations. Another already completed app is the visualization of different ODE solvers where the user is able to control parameters as step size and initial value.

A. Listings

A.1. fourierApp.py

```
#=============================================================================
# This file demonstrates a bokeh applet. The applet has been designed at TUM
# for educational purposes. The structure of the following code bases to large
# part on the work published on
# https://github.com/bokeh/bokeh/tree/master/examples/app/sliders_applet
#=============================================================================
```

```
import urllib, time
import numpy as np
import fourierFunctions as ff
import fourierTex as ft

class FourierApp(object):
    extra_generated_classes = [["FourierApp", "FourierApp", "VBox"]]
    start = 0; # start of plotting range
    end = 2*np.pi; # end of plotting range
    xRes = 200; #number of plotted points

    def __init__(self):
#=============================================================================
#         Initializes FourierApp object with all important properties
#=============================================================================
        from bokeh.document import Document
        from bokeh.session import Session
        from bokeh.models import ColumnDataSource

        self.document = Document()
        self.session = Session()
        self.session.use_doc('fourier')
        self.session.load_document(self.document)
        #here the data is stored
        self.source = ColumnDataSource(data=dict(x=[], y=[], y_series=[]))
        #finally renders the app
        self.render();

    def render(self):
#=============================================================================
#         renders the App: Sets up layout, initializes plots...
#=============================================================================
        self.init_curves()
        self.create_layout()
        self.document.add(self.layout)
        self.init_data()

    def create_layout(self):
#=============================================================================
#         Initializes layout of the interactive bokeh plot
#=============================================================================
        from bokeh.models.widgets import VBox, Slider, VBoxForm

        #slider controlling degree of the fourier series
        self.degree = Slider(
            title="degree", name='degree',
            value=5.0, start=0, end=20.0, step=1
        )
        #add behaviour to slider: slider change calls function input_change
        self.degree.on_change('value', self,'input_change')

        #lists all the controls in our plot
        self.controls = VBoxForm(
            children=[self.degree]);
        #put plot and slider in a vertical box (VBox)
        self.layout = VBox(self.controls,self.plot);

    def init_curves(self):
#=============================================================================
#         Initializes the plots of our App.
#=============================================================================
```

```
        from bokeh.plotting import figure

        toolset = "crosshair,pan,reset,resize,save,wheel_zoom"
        # Generate a figure container
        self.plot = figure(title_text_font_size="12pt",
                           plot_height=400,
                           plot_width=400,
                           tools=toolset,
                           title="fourier",#obj.text.value,
                           x_range=[0, 2*np.pi],
                           y_range=[-2.5, 2.5]
        )
        # Plot the line by the x,y values in the source property
        self.plot.line('x', 'y', source=self.source,
                       line_width=3,
                       line_alpha=0.6,
                       color='red'
        )
        self.plot.line('x','y_series',source=self.source,
                       color='green',
                       line_width=3,
                       line_alpha=0.6
        )


    def input_change(self, obj, attrname, old, new):
#==========================================================================
#         Executes whenever the input form changes.
#         It is responsible for updating the plot, or anything else you want.
#         Args:
#             obj : the object that changed
#             attrname : the attr that changed
#             old : old value of attr
#             new : new value of attr
#==========================================================================
        TeX_string = self.update_data() #update data and get new TeX
        self.send_request(TeX_string)   #send new TeX to server

    def init_data(self):
#==========================================================================
#         Called for initializing data of the plots.
#==========================================================================
        #function f(x) which will be approximated
        x = np.linspace(self.start,self.end,self.xRes)
        y = np.empty(len(x))
        for i in range(0,len(x)):
            y[i] = ff.f(x[i])

        #saving data to plot
        self.source.data = dict(x=x, y=y)
        self.update_data();
        self.session.store_document(self.document)

    def update_data(self):
#==========================================================================
#         Called each time that any watched property changes.
#
#         This updates the fourier series expansion with the most recent values
#         of the slider. The new fourier series y data is stored as a numpy
#         arrays in a dict into the app's data source property.
#==========================================================================
        x = self.source.data.get('x') # get data of f(x)
```

```
        y = self.source.data.get('y')
        N = int(round((self.degree.value))) # Get the current slider values

        # Generate Fourier series
        T = self.end - self.start #length of one period of the function
        a,b = ff.coeff(ff.f,self.start,self.end,N) #calculate coefficients
        y_series = np.empty(len(x))

        for i in range(0,len(x)): # evaluate fourier series
            y_series[i] = ff.fourier_series(a,b,T,x[i])

        #saving data to plot
        self.source.data = dict(x=x,y=y,y_series=y_series)
        self.session.store_document(self.document)

        #generate new TeX string
        TeX_string = ft.generate_tex(a,b,T)

        return TeX_string

    def send_request(self,TeX_string):
#-------------------------------------------------------------------------
#         Sends the TeX String via a request to the flask server. This directly
#         triggers the update of the html page.
#-------------------------------------------------------------------------
        print "sending request..."
        urllib.urlopen("http://localhost:5001/publish?TEX="+TeX_string)

#-------------------------------------------------------------------------
# main function
#-------------------------------------------------------------------------
import bokeh.embed as embed
appBokeh = FourierApp();

tag = embed.autoload_server(appBokeh.layout, appBokeh.session)
print("""\n use the following tag in your flask code: %s """ % tag)

link = appBokeh.session.object_link(appBokeh.document.context)
print("""You can also go to %s to see the plots on the Bokeh server directly"""
% link)

print("""Bokeh server is now running the fourier app!""")

# saves the tag which identifies the app to a file, the Flask server later
# generates a html using this tag.
tag_f = open('current_tag.tmp', 'w')
tag_f.write(tag)
tag_f.close()

# run app.
try:
    while True:
        appBokeh.session.load_document(appBokeh.document)
        time.sleep(0.1)
except KeyboardInterrupt:
    print()
```

A.2. fourierFunctions.py

```
# -*- coding: utf-8 -*-
"""
Created on Thu Jul 30 18:03:09 2015
```

```
@author: benjamin
"""

import numpy as np
import math
from scipy.integrate import quad

#==========================================================================
# Here the function, of which we calculate the fourier series, is hardcoded if
# one wants to use a different function, this has to be changed here.
#==========================================================================
def f( x ):
    return saw(x);

#==========================================================================
# The hat function
#==========================================================================
def hat(x):
    if x < np.pi:
        y = x/np.pi
    else:
        y = (2*np.pi-x)/np.pi
    return y

#==========================================================================
# The step function
#==========================================================================
def step(x):
    if x < np.pi:
        y = -1.0
    else:
        y = 1.0
    return y

#==========================================================================
# The sawtooth function
#==========================================================================
def saw(x):
    if x < np.pi:
        y = x-np.pi/2
    else:
        y = x-3*np.pi/2
    return y

#==========================================================================
# This function computes the coefficients of the fourier series representation
# of the function f, which is periodic on the interval [start,end] up to the
# degree N.
#==========================================================================
def coeff(f,start,end,N):
    T=end-start
    a = (N+1) * [0]
    b = (N+1) * [0]

    for k in range(0,N+1):
        tmp=quad(lambda x: 2/T*f(x)*math.cos(2*math.pi*k*x/T),start,end)
        a[k]=tmp[0]
        tmp=quad(lambda x: 2/T*f(x)*math.sin(2*math.pi*k*x/T),start,end)
        b[k]=tmp[0]

    a[0]=a[0]/2
```

```
    return [a,b]

#=============================================================================
# This function evaluates the fourier series of degree N with the coefficient
# vectors a and b and the period length T at the points in the array x.
#=============================================================================
def fourier_series(a,b,T,x):
    N = len(a)-1
    y = 0
    for k in range(0,N+1):
        y += a[k]*math.cos(2*math.pi*k*x/T)+b[k]*math.sin(2*math.pi*k*x/T)

    return y
```

A.3. fourierTex.py

```
# -*- coding: utf-8 -*-
"""
Created on Thu Jul 30 18:05:34 2015

@author: benjamin
"""
#=============================================================================
# Returns the correct leading sign, depending on coeff, the sign is encoded as
# a html request, since the TeX string will be sent via request (important for
# plus sign!). Additionaly the coeff is rounded to 2 digits.
#=============================================================================
def selective_str(coeff):
    if(round(coeff,3)==0):
        return ""
    elif(round(coeff,3)==1):
        return "%2B" # encoding for a plus sign in html request
    elif(round(coeff,3)==-1):
        return ""
    elif(coeff < 0):
        return ""+format(coeff,'.2f')
    else:
        return "%2B"+format(coeff,'.2f')

#=============================================================================
# Returns the value of k as a string. If k is equal to 1 an empty string is
# returned. This is particularly useful for expressions like k*something, where
# one does not want to show factors equal to 1.
#=============================================================================
def k_str(k):
    if(k==1):
        return ""
    else:
        return str(k)

#=============================================================================
# Returns a string of the format coeff*cos(k*x), if k is equal to 1 this factor
# is not displayed. The same holds for the coefficient. If the coefficient is
# small, then we omit its contribution and return an empty string.
#=============================================================================
def selective_str_cos(coeff,T,k):
    if(abs(coeff)<.01):
        return ""
    else:
        return selective_str(coeff)+"\\cos\\left("+k_str(k)+"x "+"\\frac{2\\pi}{"+str(round(T,3))+'}
```

```
#=========================================================================
# Returns a string of the format coeff*sin(k*x), if k is equal to 1 this factor
# is not displayed. The same holds for the coefficient. If the coefficient is
# small, then we omit its contribution and return an empty string.
#=========================================================================
def selective_str_sin(coeff,T,k):
    if(abs(coeff)<.01):
        return ""
    else:
        return selective_str(coeff)+"\sin\\left("+k_str(k)+"x "+"\\frac{2\\pi}{"+str(round(T,3))+'}\

#=========================================================================
# Uses the functions from above to generate a TeX string of the fourier series
# representation with given coefficients a and b and the periodicity T.
#=========================================================================
def generate_tex(a,b,T):
    TeX_string = ""
    N = len(a)-1
    print "generating TeX string for N = "+str(N)

    for k in range(0,N+1):
        if(k == 0):
            if(round(a[k],3) != 0):
                TeX_string += str(a[k])
        else:
            TeX_string += selective_str_cos(a[k],T,k)+selective_str_sin(b[k],T,k)

    if(TeX_string[0:3]=='%2B'):
        TeX_string = TeX_string[3:len(TeX_string)] #remove leading plus if there is one

    if(TeX_string==""):
        TeX_string="0"

    TeX_string = "    f(x)="+TeX_string+"    "

    print "sending the following TeX string: "+TeX_string
    return TeX_string
```

A.4. fourierFlask.py

```
import gevent
from gevent.wsgi import WSGIServer
from gevent.queue import Queue

from flask import Flask, Response, request

# SSE "protocol" is described here: http://mzl.la/UPFyxY
class ServerSentEvent(object):

    def __init__(self, data):
        self.data = data
        self.event = None
        self.id = None
        self.desc_map = {
            self.data : "data",
            self.event : "event",
            self.id : "id"
        }

    def encode(self):
        if not self.data:
            return ""
```

```
        lines = ["%s: %s" % (v, k)
                 for k, v in self.desc_map.iteritems() if k]

        return "%s\n\n" % "\n".join(lines)

appFlask = Flask(__name__)
subscriptions = []

# Client code consumes like this.
@appFlask.route("/")
def index():
    tag_f = open('current_tag.tmp','r')
    tag = tag_f.read()

    debug_template = """
<!DOCTYPE html>
<html>
<head>
<title>Fourier Transform</title>

<script type="text/javascript"
  src="https://cdn.mathjax.org/mathjax/latest/MathJax.js?config=TeX-AMS-MML_HTMLorMML">
</script>

</head>
<body>

<script>
  //
  //  Use a closure to hide the local variables from the
  //  global namespace
  //
  (function () {
    var QUEUE = MathJax.Hub.queue;  // shorthand for the queue
    var math = null, box = null;    // the element jax for the math output, and the box it's in

    //
    //  Hide and show the box (so it doesn't flicker as much)
    //
    var HIDEBOX = function () {box.style.visibility = "hidden"}
    var SHOWBOX = function () {box.style.visibility = "visible"}
    var i = 1;

    //
    //  Get the element jax when MathJax has produced it.
    //
    QUEUE.Push(function () {
      math = MathJax.Hub.getAllJax("MathOutput")[0];
      box = document.getElementById("box");
      i = i+1;
      SHOWBOX(); // box is initially hidden so the braces don't show
    });

    //
    //  The onchange event handler that typesets the math entered
    //  by the user.  Hide the box, then typeset, then show it again
    //  so we don't see a flash as the math is cleared and replaced.
    //
    UpdateMath = function (TEX) {
        console.log("running UpdateMath()");
        math.innerHTML = TEX;
        QUEUE.Push(HIDEBOX,["Text",math,TEX],SHOWBOX);
```

```
        console.log("updated TeX string!");
    }
    //
    // Flask server sent event handling
    //
    var evtSrc = new EventSource("/subscribe");

    evtSrc.onmessage = function(e) {
        console.log("received Signal from Server:");
        console.log(e.data);
        UpdateMath(e.data);
    };

  })();
</script>

<div style="height:100px" onclick="UpdateMath()" class="box" id="box" style="visibility:hidden">
    <div id="MathOutput" class="output">\(      \)</div>
</div>

<p>
%s
</p>

</body>
</html>
""" % tag
    return(debug_template)

@appFlask.route("/debug")
def debug():
    return "Currently %d subscriptions" % len(subscriptions)

@appFlask.route("/publish")
def publish():
    TEX = request.args.get("TEX")
    #Dummy data - pick up from request for real data
    def notify():
        for sub in subscriptions[:]:
            sub.put(TEX)

    gevent.spawn(notify)

    return "OK"

@appFlask.route("/subscribe")
def subscribe():
    def gen():
        q = Queue()
        subscriptions.append(q)
        try:
            while True:
                result = q.get()
                ev = ServerSentEvent(str(result))
                yield ev.encode()
        except GeneratorExit: # Or maybe use flask signals
            subscriptions.remove(q)

    return Response(gen(), mimetype="text/event-stream")

if __name__ == "__main__":
    port=5001;
```

```
print "Flask server is running!"
print "visit http://localhost:"+str(port)+" for using the app.\n"
appFlask.debug = True
server = WSGIServer(("", port), appFlask)
server.serve_forever()
```