vanderpol

February 18, 2019

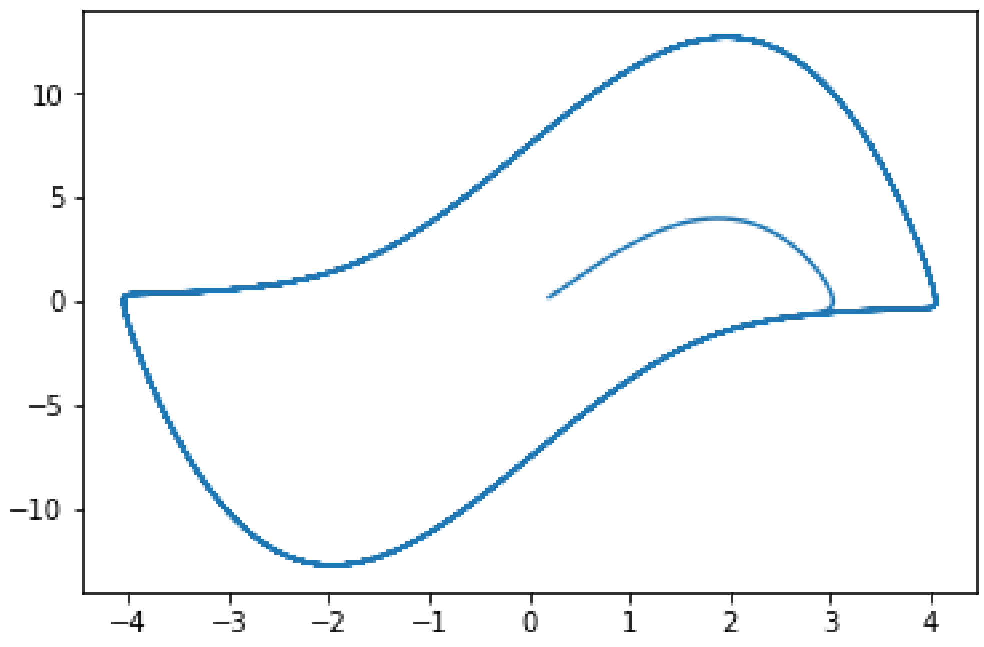
1 Training ANNs to learn the Van Der Pol equation

Lets create and run a system per the 1993 paper "Identification of Continuous-Time Dynamical Systems: Neural Network Based Algorithms and Parallel Implementation" which can be found at https://arxiv.org/abs/comp-gas/9305001

Lets see what Python shows for the Van Der Pol equation.

```
In [18]: #Settings used in https://arxiv.org/pdf/comp-gas/9305001.pdf
         mu = 1.
         delta = 4.0
         omega = 1.0

         #Settings used in https://arxiv.org/pdf/comp-gas/9305001.pdf
         #nSamples=500
         nSamples=10000

         #mu = 0.2
         #delta = 1.0
         #omega = 1.0
```

```
In [19]: # %load viewPredict.py
         import numpy as np
         from pylab import *
         from scipy.integrate import odeint

         def van_der_pol_oscillator_deriv(x, t):
             nx0 = x[1]
             nx1 = -mu * (x[0] ** 2.0 - delta) * x[1] - omega * x[0]
             res = np.array([nx0, nx1])
             return res

         ts = np.linspace(0.0, 50.0, nSamples)

         xs = odeint(van_der_pol_oscillator_deriv, [0.2, 0.2], ts)
         plt.plot(xs[:,0], xs[:,1])

         plt.show()
```

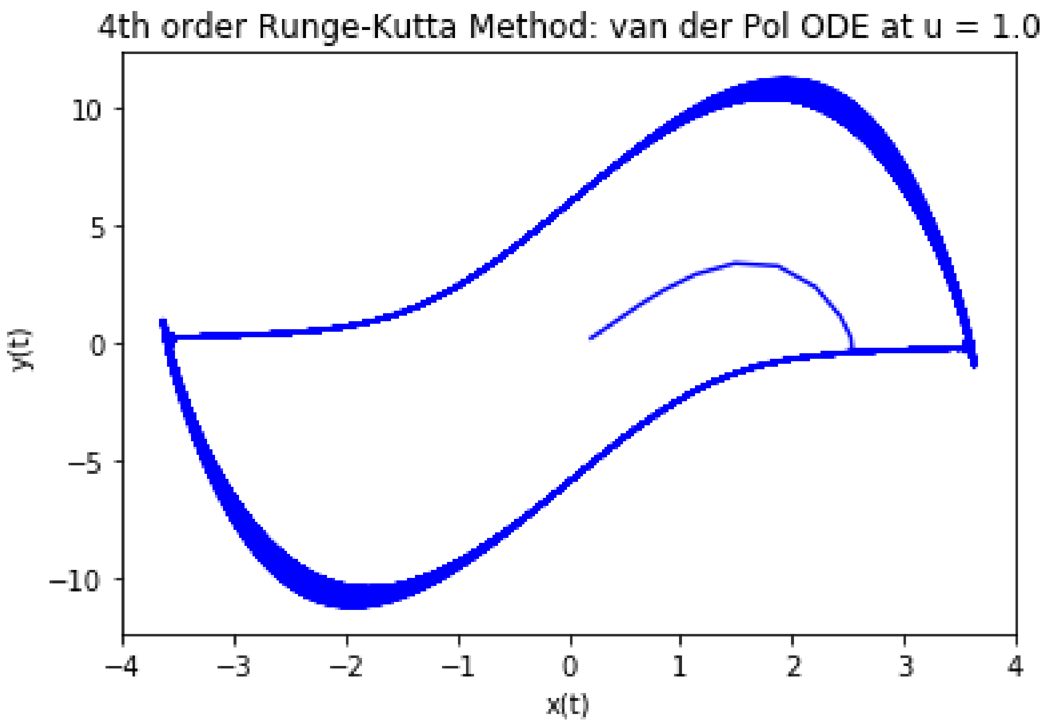


We want to train using a 4th order Runga-Kutta integrator like the following which generates the file train.dat. Note we change the Van Der Pol paramters to be those in the paper.

```
In [20]: # RK2D.py: Plot out time series of integration steps of a 2D ODE
         #      to illustrate the fourth-order Runge-Kutta method.
         #
         # For a 2D ODE
         #     dx/dt = f(x,y)
         #     dy/dt = g(x,y)
         # See RKTwoD() below for how the fourth-order Rungle-Kutta method integrates.
         #

         # Import plotting routines
         from pylab import *
         import numpy as np
         import struct


         # 2D Fourth-Order Runge-Kutta Integrator
         def RKTwoD(x, f, t):
             x = np.array(x)
             k1 = dt * f(x,t)
             k2 = dt * f(x + x / 2.0,t)
             k3 = dt * f(x + k2 / 2.0,t)
             k4 = dt * f(x + k3,t)
             x = x + ( k1 + 2.0 * k2 + 2.0 * k3 + k4 ) / 6.0
```

```
    return x

# Simulation parameters
# Integration time step
dt = 0.1

# Time
t  = [ 0.0]

# The main loop that generates the orbit, storing the states
xs= np.empty((nSamples+1,2))
xs[0]=[0.2,0.2]

for i in range(0,nSamples):
  # at each time step calculate new x(t) and y(t)
  xs[i+1]=(RKTwoD(xs[i],van_der_pol_oscillator_deriv,dt))
  t.append(t[i] + dt)

print(nSamples, len(xs))

#convert this to binary
fn = open('train.dat','wb')
fn.write(bytearray(struct.pack('i',int(2))))
fn.write(bytearray(struct.pack('i',int(2))))
fn.write(bytearray(struct.pack('i',len(xs))))

for i in range(0,nSamples):
    # write input
    fn.write(bytearray(struct.pack('f',xs[i][0])))
    fn.write(bytearray(struct.pack('f',xs[i][1])))
    #write output
    fn.write(bytearray(struct.pack('f',xs[i+1][0])))
    fn.write(bytearray(struct.pack('f',xs[i+1][1])))
fn.close()

# Setup the parametric plot
xlabel('x(t)') # set x-axis label
ylabel('y(t)') # set y-axis label
title('4th order Runge-Kutta Method: van der Pol ODE at u = ' + str(mu)) # set plot t
#axis('equal')

# Plot the trajectory in the phase plane
plot(xs[:,0],xs[:,1],'b')
show()
```

```
10000 10001
```



We train the neural network with the following. The maximum runtime is 60 seconds, but it only takes around 40 seconds to run to completion. The file param.dat is deleted below so we start from random initial conditions.

The following shows the basic configuration for this network

```
In [21]: f=open("FcnOfInterest_config.h","w")
         f.write("#define GFCN Tanh_G\n")
         f.write('#define RK4_H 0.1\n')
         f.write('#define PREDFCN rhs\n')
         f.write('#define EXPLICIT_RK4\n')
         f.close()
```

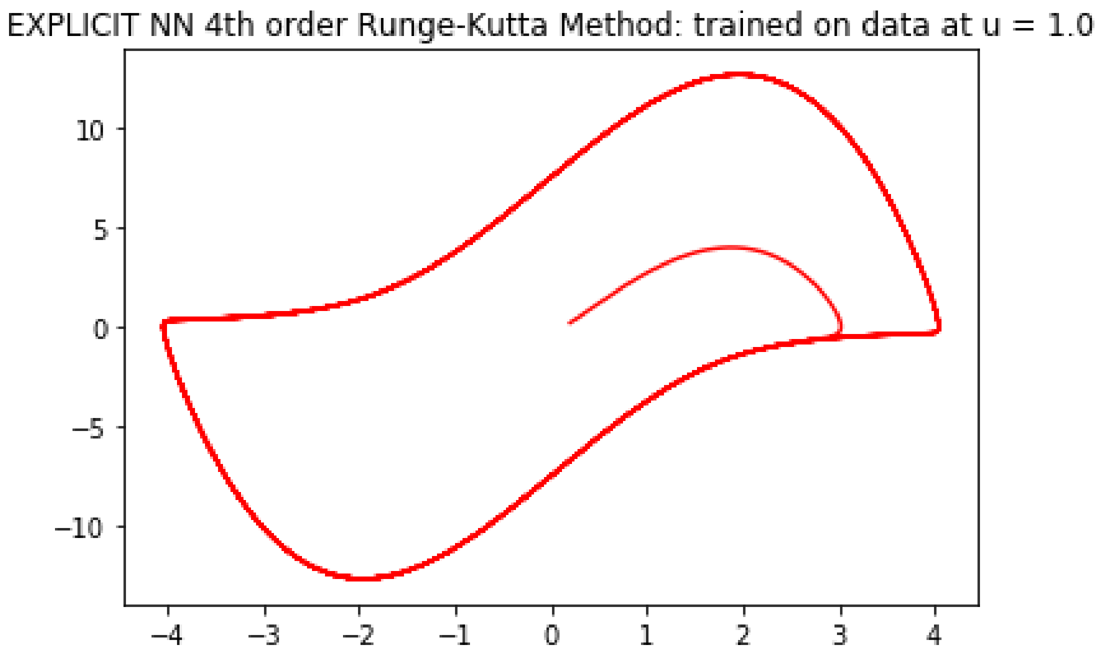
```
In [22]: import subprocess
         output = subprocess.getoutput(['make -j'])
         output = subprocess.getoutput(['sh BUILD.python.module.sh'])

         output = subprocess.getoutput(['rm -f param.dat'])
         output = subprocess.getoutput(['./nloptTrain.x -p param.dat -d train.dat -t 60'])
         print(output)
```

```
training data in: train.dat
using and/or writing params to: param.dat
OMP_NUM_THREADS 24
nInput 2 nOutput 2 nExamples 10001 in datafile (train.dat)
*******************
```

```
Objective Function: Least Means Squared
Function of Interest: EXPLICIT RK4 twolayer 2x5x5x2
Citation: https://arxiv.org/pdf/comp-gas/9305001.pdf
RK4_H=0.100000 with G() tanh()
Number params 57
Max Runtime is 60 seconds
Using NLOPT_LD_LBFGS
        Optimization Time 1.56654
        nlopt failed! ret -1
RUNTIME Info (10001 examples)
        DataLoadtime 0.00783649 seconds
        AveObjTime 0.000717605, countObjFunc 118, totalObjTime 0.0839598
        Estimated Flops myFunc 217,average GFlop/s 3.02425 nFuncCalls 118
        Estimated maximum GFlop/s 4.42998, minimum GFLop/s 0.53253
        AveGradTime 0.0125205, nGradCalls 118, totalGradTime 1.47742
        nFuncCalls/nGradCalls 1.00
```

This shows we trained with the EXPLICIT RK4 implementation described in the paper. Let's look at the output by calling the C++ prediction function using the model parameters from the training run. We can then integrate to see if we learned the Van Der Pol equation.

```
In [23]: # Import plotting routines
         from pylab import *
         import numpy as np
         import struct
         from farbopt import PyPredFcn

         from scipy.integrate import odeint

         ts = np.linspace(0.0, 50.0, nSamples)

         xs = odeint(van_der_pol_oscillator_deriv, [0.2, 0.2], ts)

         title('EXPLICIT NN 4th order Runge-Kutta Method: trained on data at u = ' + str(mu))

         plt.plot(xs[:,0], xs[:,1],'r')
         #plt.gca().set_aspect('equal')
         #plt.savefig('vanderpol_oscillator.png')
         plt.show()
```



2 Implicit implementation

Looks good! Now lets try the implicit integrator using ANN recursion. See paper for more details

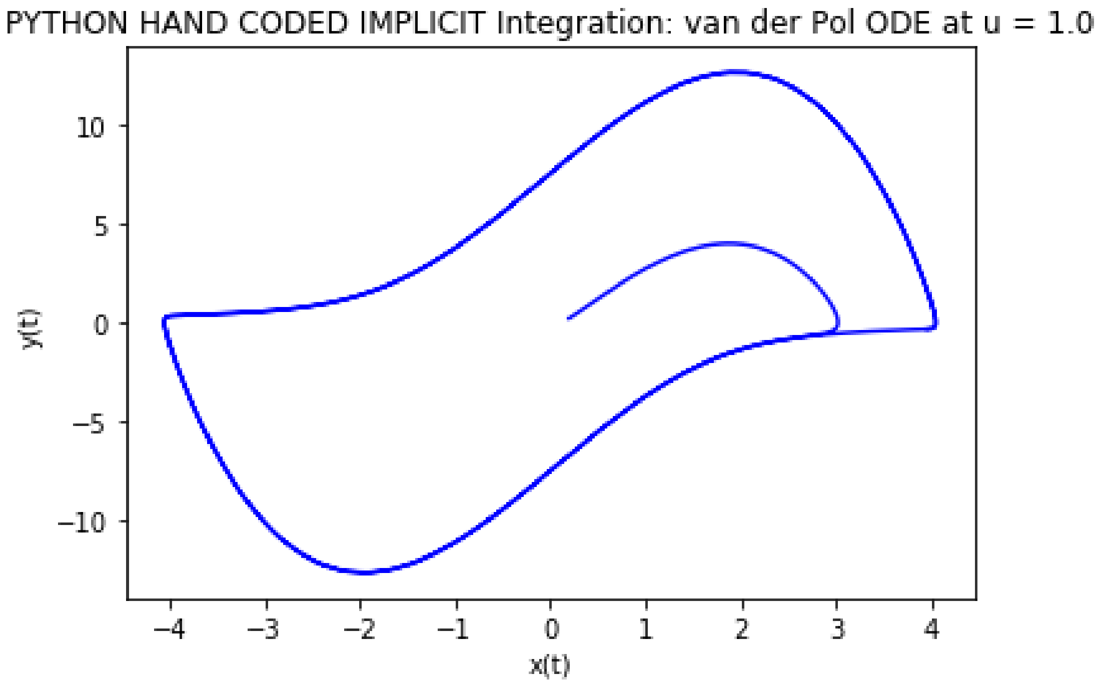
In [24]:
```
# Simulation parameters
# Integration time step
# The number of time steps to integrate over

dt_step = 0.02
nRecurIters=int(dt/dt_step) # we iterate to the previous dt
writeImplicitData=1 # keep the previous data set

def ImplicitMethod(x,f,t):
    x = np.array(x)
    Yn_1 = np.array(f(x,t))
    t1 = np.array(f(x,t))
    for i in range(nRecurIters):
        Yn_1 = x + dt_step/2. * (t1 + np.array(f(Yn_1,t)))
    return Yn_1
```

In [25]:
```
#Implicit integrator

# Import plotting routines
from pylab import *
import numpy as np
```

```python
import struct

nSamples = 1000
nInitialPointSamples=100
nRandomInitial = 100
RandomRange=1.
RandomRangeNegSkew=0.

# Time
t  = [ 0.0]

# The main loop that generates the orbit, storing the states
xs= np.empty((nSamples+1,2))
xs[0]=[0.2,0.2]
for i in range(0,nSamples):
  # at each time step calculate new x(t) and y(t)
  xs[i+1]=(ImplicitMethod(xs[i],van_der_pol_oscillator_deriv,dt))
  #t.append(t[i] + dt)

initialPoints = []
for i in range(0,nRandomInitial):
    initialPoints.append(RandomRange * rand(1,2) - RandomRangeNegSkew)
#print(initialPoints)
trainData=[]

xs1= np.empty((nInitialPointSamples+1,2))

for l in initialPoints:
    xs1[0]=np.array(l)
    for i in range(0,nInitialPointSamples):
        # at each time step calculate new x(t) and y(t)
        xs1[i+1]=(ImplicitMethod(xs1[i],van_der_pol_oscillator_deriv,dt))
        trainData.append(list(xs1[i]))
        trainData.append(list(xs1[i+1]))

for i in range(0,nSamples):
    trainData.append(xs[i])
    trainData.append(xs[i+1])

nExamples = int(len(trainData)/2)
print("trainData", len(trainData), nExamples)

if(writeImplicitData != 0):
    fn = open('train.dat','wb')
    fn.write(bytearray(struct.pack('i',int(2))))
    fn.write(bytearray(struct.pack('i',int(2))))
    fn.write(bytearray(struct.pack('i',nExamples)))
```

```
            for t in trainData:
                # write values
                fn.write(bytearray(struct.pack('f',t[0])))
                fn.write(bytearray(struct.pack('f',t[1])))
            fn.close()

        #
        # Setup the parametric plot
        xlabel('x(t)') # set x-axis label
        ylabel('y(t)') # set y-axis label
        title('PYTHON HAND CODED IMPLICIT Integration: van der Pol ODE at u = ' + str(mu)) #
        #axis('equal')

        # Plot the trajectory in the phase plane
        plot(xs[:,0],xs[:,1],'b')
        show()
```

```
trainData 22000 11000
```



Looks good so let's train the network using the implicit integrator. Note this uses a form of recurrence connection. See paper for details.

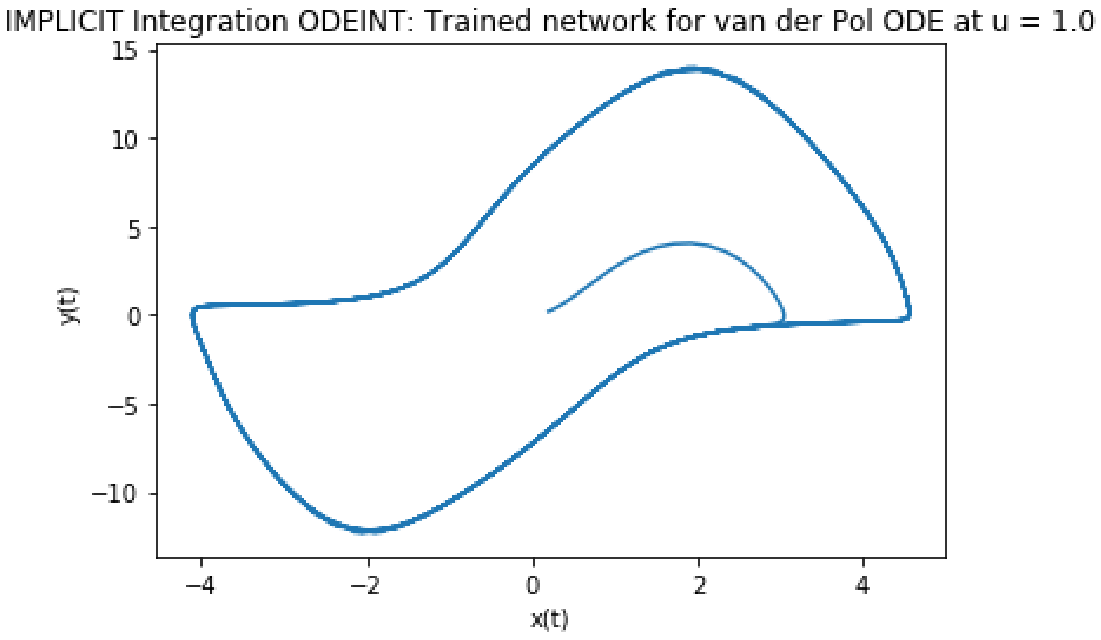
```
In [26]: f=open("FcnOfInterest_config.h","w")
         f.write("#define GFCN Tanh_G\n")
         f.write('#define PREDFCN rhs\n')
```

```
        f.write('#define IMPLICIT_RK4\n')
        f.write(str('#define RK4_H ' + str(dt_step) +'\n'))
        f.write(str('#define RK4_RECURRENCE_LOOPS ' + str(nRecurIters) +'\n'))
        f.close()
```

```
In [30]: import subprocess
         doTrain=0

         output = subprocess.getoutput(['make -j'])
         output = subprocess.getoutput(['sh BUILD.python.module.sh'])

         if doTrain != 0:
             output = subprocess.getoutput(['rm -f param.dat'])
             print("-----------------------------------------------------------------------
             output = subprocess.getoutput(['./nloptTrain.x -p param.dat -d train.dat -t 60 --
             print(output)
             print("-----------------------------------------------------------------------
             output = subprocess.getoutput(['./nloptTrain.x -p param.dat -d train.dat -t 120 -
             print(output)
         else:
             print('training is sensitive to initial conditions and requires playing around. L
             output = subprocess.getoutput(['cp param.dat.implicit param.dat'])
```

```
training is sensitive to initial conditions and requires playing around. Let's use a pretrained
```

```
In [31]: # %load viewPredict.py
         import numpy as np
         from pylab import *
         from farbopt import PyPredFcn

         from scipy.integrate import odeint

         def predicted_rhs(x,t):
             return RHS.predict(x)

         ts = np.linspace(0.0, 50.0, nSamples)
         # Setup the parametric plot
         xlabel('x(t)') # set x-axis label
         ylabel('y(t)') # set y-axis label
         title('IMPLICIT Integration ODEINT: Trained network for van der Pol ODE at u = ' + st
         RHS=PyPredFcn(b'param.dat')

         xs = odeint(predicted_rhs, [0.2, 0.2], ts)
         plt.plot(xs[:,0], xs[:,1])
         plt.show()
```


IMPLICIT Integration ODEINT: Trained network for van der Pol ODE at u = 1.0
15
10
5
0
−5
−10
y(t)
−4
−2
0
2
4
x(t)

```
In [ ]:
```

```
In [ ]:
```