Stochastic Processes Comparison of Accuracy: Data Assimilation, Echo State Machines and Nonlinear Vector Autoregressive Learning Methods.

Comparing discrete and variational data assimilation methods to reservoir computing - machine learning for synthetic chaotic nonlinear dynamical systems

1. **The Attractors Used**
   - Lorenz 1963 Model for Atmospheric Convection (Rayleigh Benard Convection)
2. **Reservoir Computing Methods**
   - Echo State Machine - Hierarchial Structure
3. **Other Neural Network Models**
   - Nonlinear Vector AutoRegression (NVAR)
4. **Data Assimilation Methods**
   - 4D Variational Data Assimilation * Particle Filter

In [ ]:
```
% % capture
!pip3 install reservoirpy
!pip3 install seaborn
!pip3 install sklearn
!pip3 install scikit
!pip3 install matplotlib
!pip3 install numpy
```

```
UsageError: Line magic function `%` not found.
```

Installation of packages used in this experiment

In [ ]:
```
import reservoirpy
import seaborn as seaborn
import numpy as numpy
import matplotlib.pyplot
import random
from sklearn.metrics import mean_squared_error
import string
import time
# %matplotlib qt
matplotlib.rcParams.update(matplotlib.rcParamsDefault)
%matplotlib inline
```

Runge-Kutta Algorithm for three dimensional chaotic dynamical systems

In [ ]:
```
start = time.time()
# inital states for the loops, so each lorenz attarctor will be passed throu
initialXvalue, initialYvalue, initialZvalue = 1, 1, 1
states = None
dt = 0.001
# Initiate states for the approximation
alpha, beta, gamma = 10, 28, 8/3
```

In [ ]:
```
def xt(x, y, z, t):
    return (alpha * (y - x))


def yt(x, y, z, t):
    return (beta*x - x*z - y)


def zt(x, y, z, t):
    return (x*y - gamma * z)
```

In [ ]:
```
def approximate(xt, yt, zt, n, x, y, z, t, dt):
    for k in range(n):
        t[k + 1] = t[k] + dt

        roundOne_k = xt(x[k], y[k], z[k], t[k])
        roundOne_l = yt(x[k], y[k], z[k], t[k])
        roundOne_m = zt(x[k], y[k], z[k], t[k])

        roundTwo_k = xt(
            (x[k] + 0.5*roundOne_k*dt),
            (y[k] + 0.5*roundOne_l*dt),
            (z[k] + 0.5*roundOne_m*dt),
            (t[k] + dt/2))
        roundTwo_l = yt(
            (x[k] + 0.5*roundOne_k*dt),
            (y[k] + 0.5*roundOne_l*dt),
            (z[k] + 0.5*roundOne_m*dt),
            (t[k] + dt/2))
        roundTwo_m = zt(
            (x[k] + 0.5*roundOne_k*dt),
            (y[k] + 0.5*roundOne_l*dt),
            (z[k] + 0.5*roundOne_m*dt),
            (t[k] + dt/2))

        roundThree_k = xt(
            (x[k] + 0.5*roundTwo_k*dt),
            (y[k] + 0.5*roundTwo_l*dt),
            (z[k] + 0.5*roundTwo_m*dt),
            (t[k] + dt/2))
        roundThree_l = yt(
            (x[k] + 0.5*roundTwo_k*dt),
            (y[k] + 0.5*roundTwo_l*dt),
            (z[k] + 0.5*roundTwo_m*dt),
            (t[k] + dt/2))
        roundThree_m = zt(
            (x[k] + 0.5*roundTwo_k*dt),
            (y[k] + 0.5*roundTwo_l*dt),
            (z[k] + 0.5*roundTwo_m*dt),
            (t[k] + dt/2))

        roundFour_k = xt(
            (x[k] + roundThree_k*dt),
            (y[k] + roundThree_l*dt),
            (z[k] + roundThree_m*dt),
            (t[k] + dt))
        roundFour_l = yt(
            (x[k] + roundThree_k*dt),
            (y[k] + roundThree_l*dt),
            (z[k] + roundThree_m*dt),
            (t[k] + dt))
        roundFour_m = zt(
            (x[k] + roundThree_k*dt),
```

```
            (y[k] + roundThree_l*dt),
            (z[k] + roundThree_m*dt),
            (t[k] + dt))

        x[k+1] = x[k] + (dt*(roundOne_k + 2*roundTwo_k +
                         2*roundThree_k + roundFour_k) / 6)
        y[k+1] = y[k] + (dt*(roundOne_l + 2*roundTwo_l +
                         2*roundThree_l + roundFour_l) / 6)
        z[k+1] = z[k] + (dt*(roundOne_m + 2*roundTwo_m +
                         2*roundThree_m + roundFour_m) / 6)
```

In [ ]:
```
def Runge_Kutta_Approximation(xt, yt, zt, n=100000, T=10000):
    x = numpy.zeros(n + 1)  # x[k] is the solution at time t[k]
    y = numpy.zeros(n + 1)  # y[k] is the solution at time t[k]
    z = numpy.zeros(n + 1)  # z[k] is the solution at time t[k]
    t = numpy.zeros(n + 1)
    x[0] = initialXvalue
    y[0] = initialYvalue
    z[0] = initialZvalue
    t[0] = 0
    dt = 0.001
    # Compute the approximate solution at equally spaced times.
    approximate(xt, yt, zt, n, x, y, z, t, dt)

    return x, y, z, t
```

In [ ]:
```
def initialDerivative(xt, yt, zt, x, y, z, t):
    f0_dx = xt(x[0], y[0], z[0], t[0])
    f0_dy = yt(x[0], y[0], z[0], t[0])
    f0_dz = zt(x[0], y[0], z[0], t[0])

    f1_dx = xt(x[1], y[1], z[1], t[1])
    f1_dy = yt(x[1], y[1], z[1], t[1])
    f1_dz = zt(x[1], y[1], z[1], t[1])

    f2_dx = xt(x[2], y[2], z[2], t[2])
    f2_dy = yt(x[2], y[2], z[2], t[2])
    f2_dz = zt(x[2], y[2], z[2], t[2])

    f3_dx = xt(x[3], y[3], z[3], t[3])
    f3_dy = yt(x[3], y[3], z[3], t[3])
    f3_dz = zt(x[3], y[3], z[3], t[3])
    return f0_dx, f0_dy, f0_dz, f1_dx, f1_dy, f1_dz, f2_dx, f2_dy, f2_dz, f3
```

In [ ]:
```
def prediction_round(RungeKutta, xt, yt, zt, initalDerivative, n=100000):
    # adams-bashford method AB4 used for standard paramters for the lorenz s
    x, y, z, t = RungeKutta
    f0_dx, f0_dy, f0_dz, f1_dx, f1_dy, f1_dz, f2_dx, f2_dy, f2_dz, f3_dx, f3
    for k in range(n-1, 0, -1):
        x[k + 1] = x[k] + (dt / 24) * (55*f3_dx - 59 *
                                       f2_dx + 37*f1_dx - 9*f0_dx)
        y[k + 1] = y[k] + (dt / 24) * (55 * f3_dy - 59 *
                                       f2_dy + 37 * f1_dy - 9 * f0_dy)
        z[k + 1] = z[k] + (dt / 24) * (55 * f3_dz - 59 *
                                       f2_dz + 37 * f1_dz - 9 * f0_dz)

        x[k + 1] = x[k] + (dt / 24) * (9 * xt(x[k+1], y[k+1],
                                              z[k+1], t[k+1]) + 19 * f3_dx -
        y[k + 1] = y[k] + (dt / 24) * (9 * yt(x[k+1], y[k+1],
                                              z[k+1], t[k+1]) + 19 * f3_dy -
        z[k + 1] = z[k] + (dt / 24) * (9 * zt(x[k+1], y[k+1],
```

```
                                                z[k+1], t[k+1]) + 19 * f3_dz -
        return x, y, z, t
```

In [ ]:
```
def PredictorCorrector(xt, yt, zt, n=100000, T=35):
    x = numpy.zeros(n + 2)  # x[k] is the solution at time t[k]
    y = numpy.zeros(n + 2)  # y[k] is the solution at time t[k]
    z = numpy.zeros(n + 2)  # z[k] is the solution at time t[k]
    t = numpy.zeros(n + 2)
    x[0] = initialXvalue
    y[0] = initialYvalue
    z[0] = initialZvalue
    t[0] = 0
    x, y, z, t = prediction_round(Runge_Kutta_Approximation(xt, yt, zt, n),
                                  xt, yt, zt,
                                  initialDerivative(xt, yt, zt, x, y, z, t),
                                  n)

    return x, y, z, t
```

In [ ]:
```
dataPoints = PredictorCorrector(xt, yt, zt, n = 100000)
```

Reservoir Computing Echo State Network: Ridge Regression

In [ ]:
```
x, y, z, = dataPoints[0], dataPoints[1], dataPoints[2]
```

In [ ]:
```
def get_string():
    # get random string of letters and digits
    source = string.ascii_letters + string.digits
    result_str = ''.join((random.choice(source) for i in range(8)))

    return result_str
    # invoking the function
```

Plotting for the outputs

In [ ]:
```
def plotTimeSeries(PREDICTED, FORECAST=[], dataPointsTS=[0]):
    matplotlib.pyplot.figure(figsize=(15, 5))
    matplotlib.pyplot.plot(FORECAST, label='Forecast', color='green')
    matplotlib.pyplot.plot(PREDICTED, label='predicted',color='blue', alpha=
    matplotlib.pyplot.plot(dataPointsTS, label='Ground Truth', color='orange
    # matplotlib.pyplot.ylim([-0.25, 0.25])
    # matplotlib.pyplot.xlim([10, 100100])
    # matplotlib.pyplot.autoscale()
    matplotlib.pyplot.legend()
    matplotlib.pyplot.show()



def plot3dTimeSeries(PREDICTEDx, PREDICTEDy, PREDICTEDz):
    fig1 = matplotlib.pyplot.figure(figsize=(10, 10))
    #    fig2 = plt.figure(figsize = (30, 30))
    ax = matplotlib.pyplot.axes(projection='3d')
    ax = matplotlib.pyplot.axes(projection='3d')
    # ax.set_xlim3d([-25, 30])
    # ax.set_ylim3d([-4, 3])
    matplotlib.pyplot.autoscale()
    # Data for a three-dimensional line
    zline = numpy.concatenate(PREDICTEDx, axis=0)
    xline = numpy.concatenate(PREDICTEDy, axis=0)
```

```
    yline = numpy.concatenate(PREDICTEDz, axis=0)
    ax.plot3D(xline, yline, zline)
```

Hierarchial ESN Structure

In [ ]:
```
def HierarchialESN(input, PercentageTrain=0.9, forecast_steps=20000):
     x = numpy.array(input).T
     result = get_string()
     reservoir = reservoirpy.nodes.Reservoir(1000, name='res1-1{}'.format(re
     Readout = reservoirpy.nodes.Ridge(len(input), ridge=2e-7, name='readout
     data = reservoirpy.nodes.Input()
     X_TRAIN = x[:int(len(x) * PercentageTrain)]
     Y_TRAIN = x[1:int(len(x) * PercentageTrain) + 1]
     connection_one = data >> reservoir >> Readout
     connection_two = data >> Readout
     ESN_MODEL = connection_one & connection_two
     ESN_MODEL = ESN_MODEL.fit(X_TRAIN, Y_TRAIN)
     warmup = ESN_MODEL.run(x[: len(x)])
     PREDICTED = numpy.empty((len(dataPoints[0]) + forecast_steps, 3)) # pre
     forecast = warmup[-100]
     for i in range(len(dataPoints[0]) + forecast_steps):
          forecast = ESN_MODEL(forecast)
          PREDICTED[i] = forecast
     return PREDICTED, warmup
```

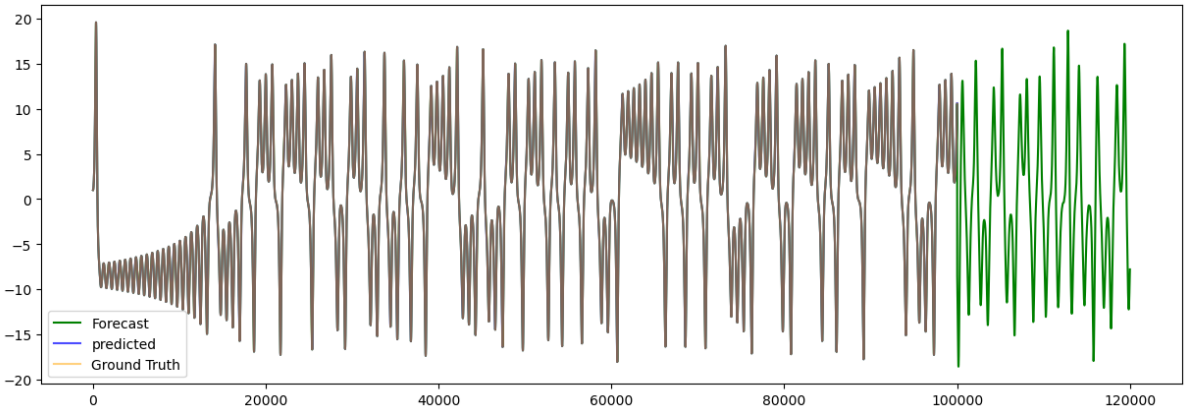
Note here that the greyish-purple colour is an overlapping of the blue and orange

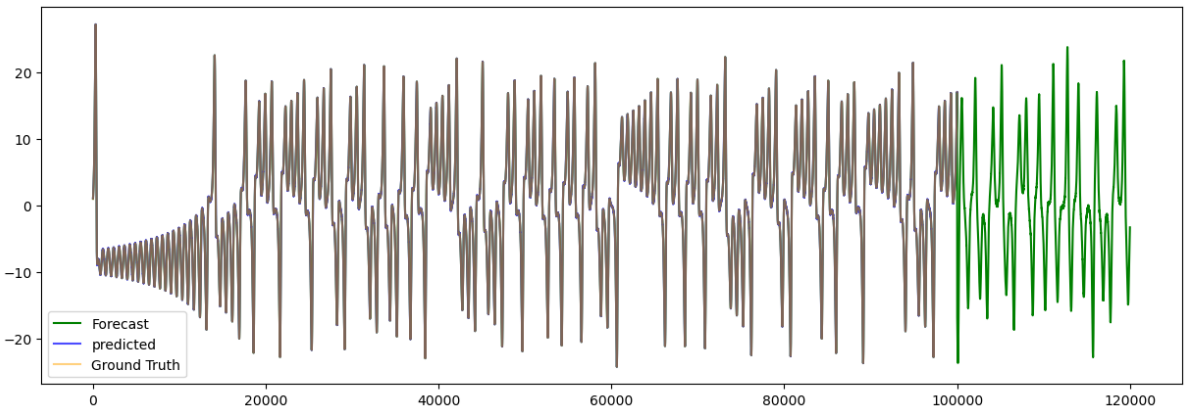
In [ ]:
```
# create simple vector containing each axis time series
x = [dataPoints[0],
     dataPoints[1],
     dataPoints[2]]
```

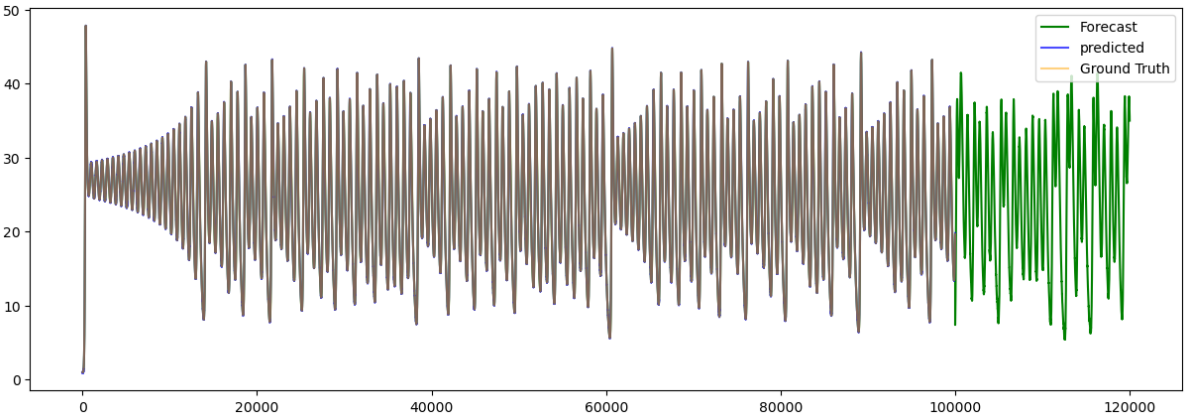
In [ ]:
```
# only show as a forecast, not the approximate of ground state.
PREDICTED, warmup = HierarchialESN(x, PercentageTrain=0.9, forecast_steps=20
```

```
Running Model-87: 90000it [00:25, 3480.30it/s]?, ?it/s]
Running Model-87: 100%|██████████| 1/1 [00:28<00:00, 28.28s/it]
Fitting node readout1-12K4NymF6...
Running Model-87: 100001it [00:24, 4010.47it/s]
```

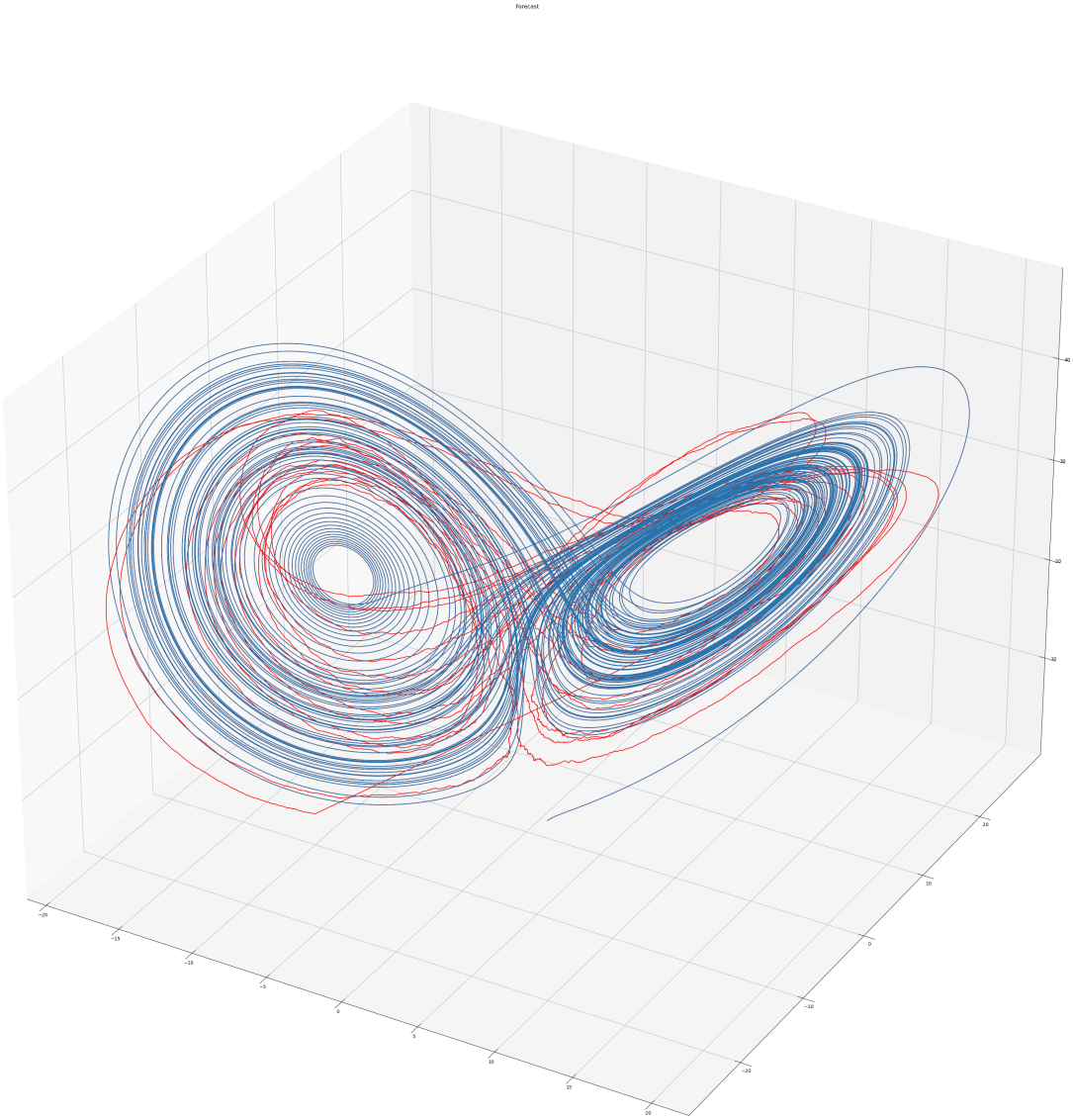
In [ ]:
```
# concatenate the predicted and ground state
x_forecast = numpy.concatenate((dataPoints[0], PREDICTED[len(dataPoints[0]):
y_forecast = numpy.concatenate((dataPoints[1], PREDICTED[len(dataPoints[0]):
z_forecast = numpy.concatenate((dataPoints[2], PREDICTED[len(dataPoints[0]):

plotTimeSeries(warmup[:, 0], x_forecast , dataPoints[0])
plotTimeSeries(warmup[:, 1], y_forecast , dataPoints[1])
plotTimeSeries(warmup[:, 2], z_forecast , dataPoints[2])

fig = matplotlib.pyplot.figure(figsize=(100, 100))
ax = fig.add_subplot(1, 2, 1, projection = '3d')
ax.plot3D(x_forecast,
          y_forecast,
          z_forecast, color = 'r')
ax.set_title('Forecast')
ax.plot3D(dataPoints[0],
          dataPoints[1],
          dataPoints[2])
```







Out[ ]: [<mpl_toolkits.mplot3d.art3d.Line3D at 0x7fb91f35d8b0>]



Kind of...it seems to capture the 'global oscillations' within each time series, however, for the x-axis it is clear that it achieves 'inner' oscillations on the top but not the bottom. The same is true for the y-axis. the z-axis seems damped towards the top of each oscillation and not capturing the 'pulses'.

The innacuracies are clearly demonstrated in the 3D figure. However, globally the forecasted time series follow similar dynamics to that seen in the ground state.

The algorithm fits the series fine, with minimal error

NOTE - this requires further investigation

Nonlinear Vector AutoRegressive (NVAR) Model - using ReservoirPy

In [ ]:
```
# put the system discretised by the runge kutta approximation into a vector
testTime = 2000  # change when needed
forecastTime = 20000 # change when needed
```

In [ ]:
```
X = [dataPoints[0], dataPoints[1], dataPoints[2]]
X = numpy.array(X).T
```

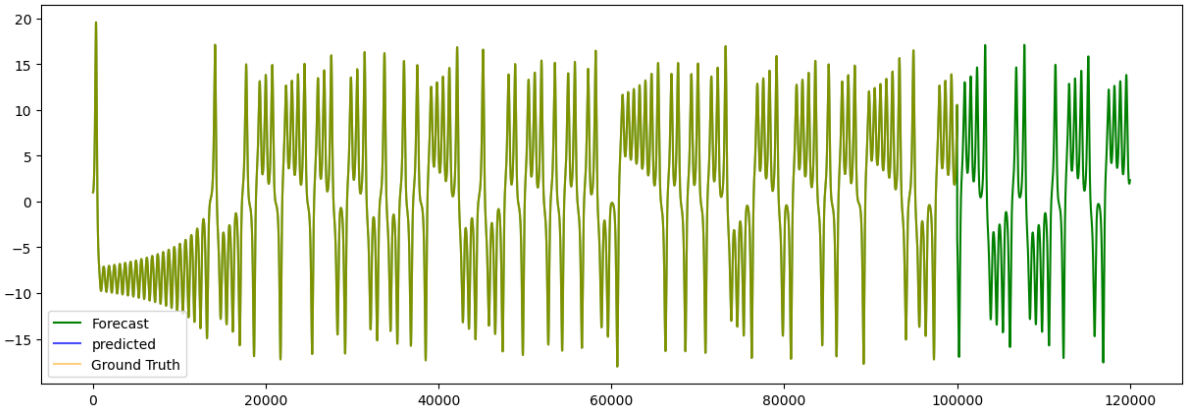
```
NVAR = reservoirpy.nodes.NVAR(delay=2, order=2, strides=1)
Readout = reservoirpy.nodes.Ridge(3, ridge=2.5e-7)
MODEL = NVAR >> Readout
Xi = X[:testTime]
dXi = X[1:testTime + 1] - X[:testTime]  # difference u[t+1] - u[t]
Y_test = X[testTime:]  # testing data
model = MODEL.fit(Xi, dXi)
model = model.fit(Xi, dXi, warmup = 200)
u = X[testTime]
res = numpy.zeros((len(dataPoints[0]) + forecastTime, Readout.output_dim))
for i in range(len(dataPoints[0]) + forecastTime):
    u = u + model(u)
    res[i, :] = u
X_PREDICTED_nvar = []
Y_PREDICTED_nvar = []
Z_PREDICTED_nvar = []

for i in range(len(res)):
    X_PREDICTED_nvar.append(res[i][0])
    Y_PREDICTED_nvar.append(res[i][1])
    Z_PREDICTED_nvar.append(res[i][2])
```

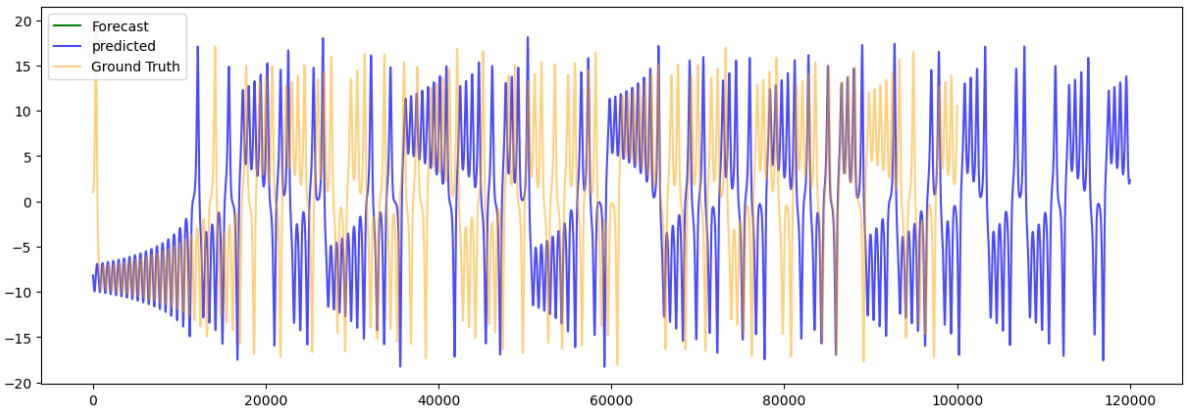
```
Running Model-88: 2000it [00:00, 14806.27it/s]?, ?it/s]
Running Model-88: 100%|██████████| 1/1 [00:00<00:00,  7.18it/s]
/Users/adam/opt/anaconda3/lib/python3.9/site-packages/reservoirpy/nodes/read
outs/ridge.py:17: LinAlgWarning: Ill-conditioned matrix (rcond=1.3671e-17):
result may not be accurate.
  return linalg.solve(XXT + ridge, YXT.T, assume_a="sym")
Fitting node Ridge-16...
```

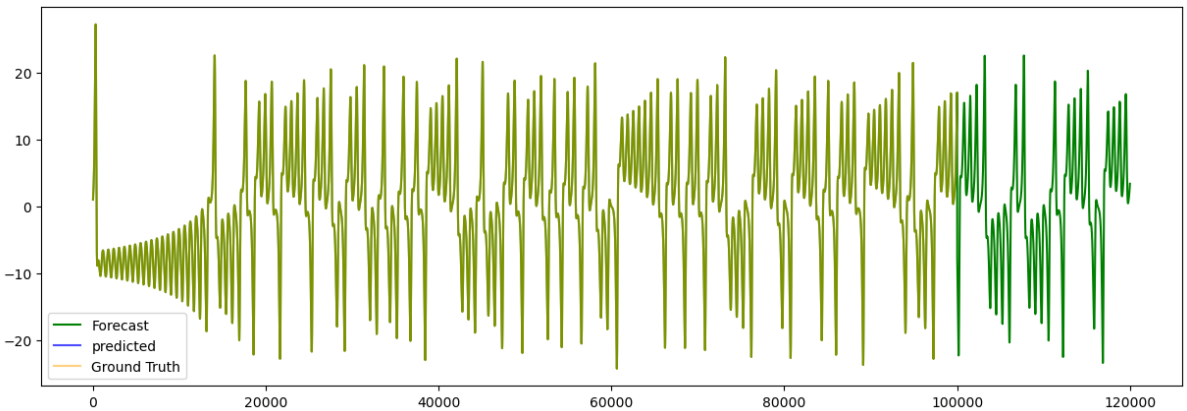
```
Running Model-88: 2000it [00:00, 14603.87it/s]?, ?it/s]
Running Model-88: 100%|██████████| 1/1 [00:00<00:00,  7.02it/s]
/Users/adam/opt/anaconda3/lib/python3.9/site-packages/reservoirpy/nodes/read
outs/ridge.py:17: LinAlgWarning: Ill-conditioned matrix (rcond=7.21223e-18):
result may not be accurate.
  return linalg.solve(XXT + ridge, YXT.T, assume_a="sym")
Fitting node Ridge-16...
```

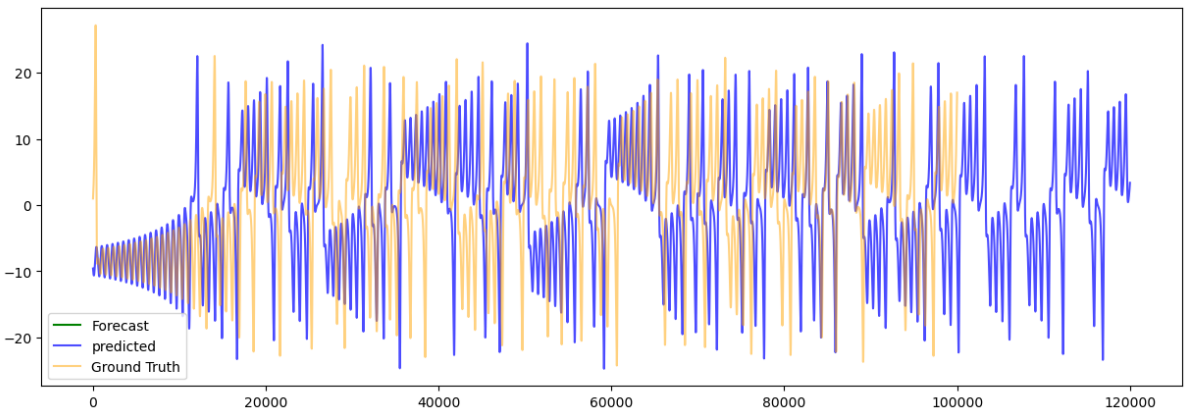
In [ ]:

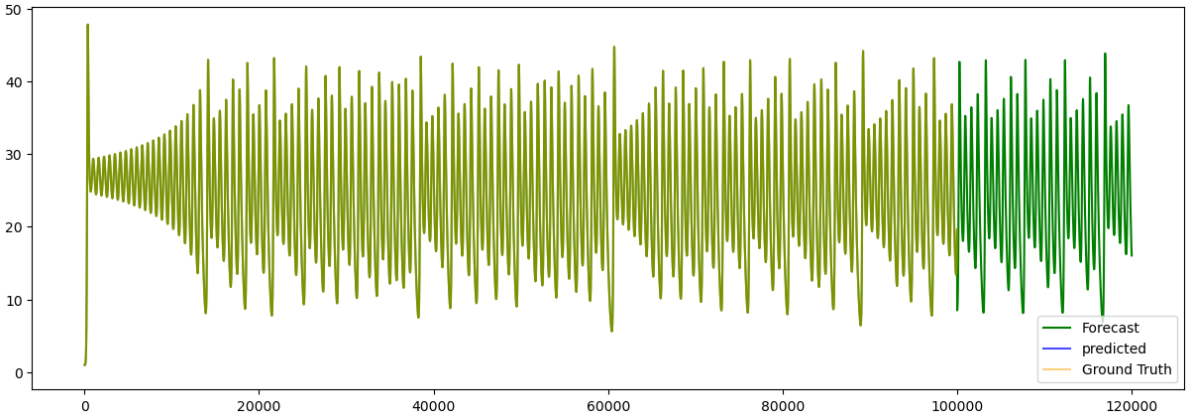
```
# concatenate the predicted and ground state
x_forecast = numpy.concatenate((dataPoints[0], X_PREDICTED_nvar[len(dataPoin
y_forecast = numpy.concatenate((dataPoints[1], Y_PREDICTED_nvar[len(dataPoin
z_forecast = numpy.concatenate((dataPoints[2], Z_PREDICTED_nvar[len(dataPoin

plotTimeSeries(0, x_forecast , dataPoints[0]) # conactenated ground state an
plotTimeSeries(X_PREDICTED_nvar, 0, dataPoints[0])
plotTimeSeries(0, y_forecast , dataPoints[1]) # conactenated ground state an
plotTimeSeries(Y_PREDICTED_nvar, 0, dataPoints[1])
plotTimeSeries(0, z_forecast , dataPoints[2]) # conactenated ground state an
plotTimeSeries(Z_PREDICTED_nvar, 0, dataPoints[2])

fig = matplotlib.pyplot.figure(figsize=(100, 100))
ax = fig.add_subplot(1, 2, 1, projection = '3d')
ax.plot3D(x_forecast,
          y_forecast,
          z_forecast, color = 'r')
ax.set_title('Forecast')
ax.plot3D(dataPoints[0],
          dataPoints[1],
          dataPoints[2])
```

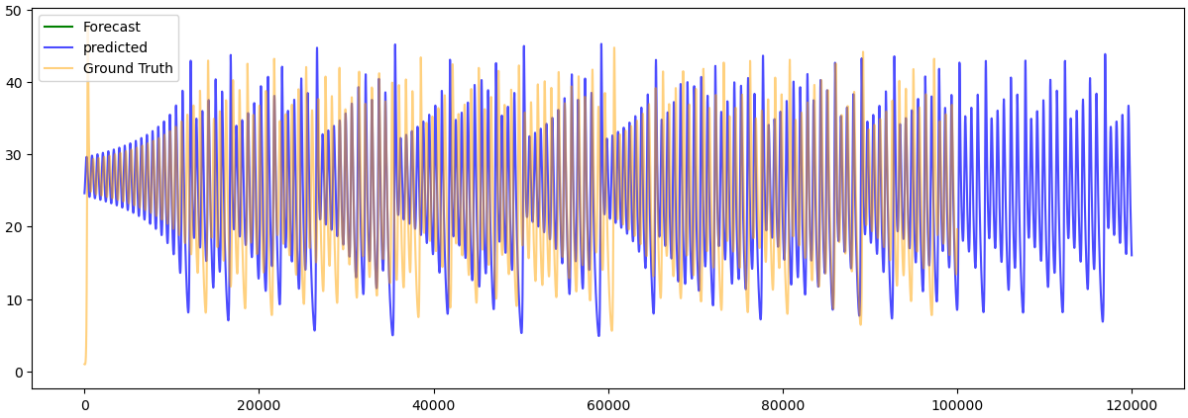

Forecast
predicted
Ground Truth


Forecast
predicted
Ground Truth

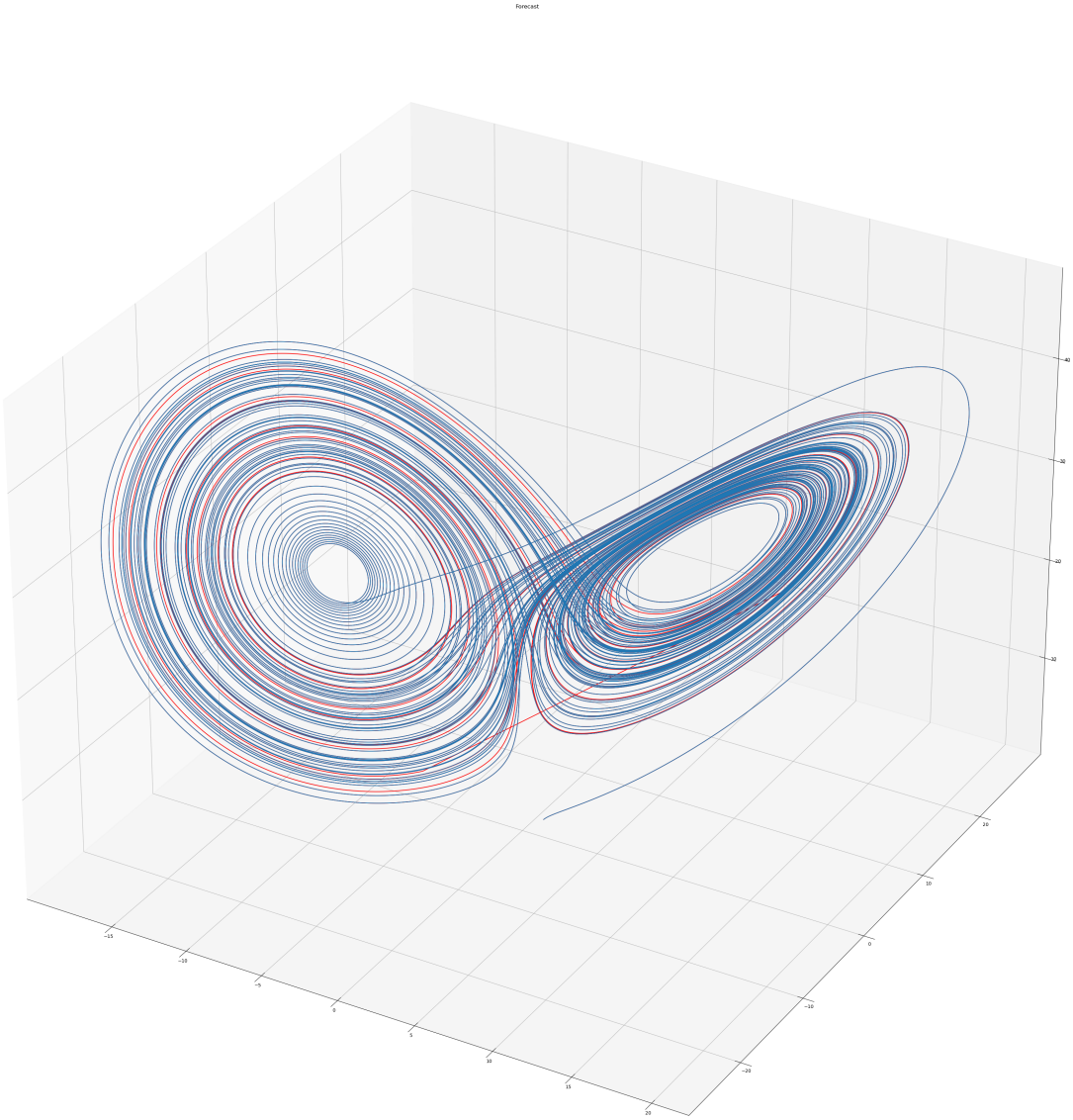

Forecast
predicted
Ground Truth


Forecast
predicted
Ground Truth





Out[ ]: `[<mpl_toolkits.mplot3d.art3d.Line3D at 0x7fbb2b2d0040>]`



NVAR - Isolated Algorithm

In [ ]:
```
# time step
dt = 0.01
# units of time to warm up NVAR. need to have warmup_pts >= 1
WARMUP_TIME = 500
# units of time to train for
TRAIN_TIME = 100000
# units of time to test for
TEST_TIME = 120000
# total time to run for
MAX_TIME = WARMUP_TIME+TRAIN_TIME+TEST_TIME
# Lyapunov time of the Lorenz system
LYAPUNOV_TIME = 1.104

# discrete-time versions of the times defined above - DIVIDE BY dt to increa
# NOTE - you MUST increase the resolution in the runge kutta approximation a
FULL_WARMUP_TIME = round(WARMUP_TIME)
FULL_TRAIN_TIME = round(TRAIN_TIME)
FULL_WARMUP_TRAIN_TIME = FULL_WARMUP_TIME+FULL_TRAIN_TIME
FULL_TEST_TIME = round(TEST_TIME)
FULL_MAX_TIME = round(MAX_TIME)
FULL_LYAPUNOV_TIME = round(LYAPUNOV_TIME)
```

```
# input dimension
INPUT_DIMENSIONS = 3
# number of time delay taps
TIME_DELAY_STEPS = 2
# size of linear part of feature vector
DIM_LINEAR = TIME_DELAY_STEPS*INPUT_DIMENSIONS
# size of nonlinear part of feature vector
DIM_NONLINEAR = int(DIM_LINEAR*(DIM_LINEAR+1)/2)
# total size of feature vector: constant + linear + nonlinear
DIM_TOTAL = 1 + DIM_LINEAR + DIM_NONLINEAR

# ridge parameter for regression
RIDGE_PARAMETER = 2.5e-6

# t values for whole evaluation time
# (need maxtime_pts + 1 to ensure a step of dt)
t_eval = numpy.linspace(0, MAX_TIME, FULL_MAX_TIME + 1)

# I integrated out to t=50 to find points on the attractor, then use these a
dataPoints = PredictorCorrector(xt, yt, zt, n=MAX_TIME)
print("data length", len(dataPoints[0]))
lorenz_soln = numpy.array([dataPoints[0], dataPoints[1], dataPoints[2]])

# total variance of Lorenz solution
INPUT_VARIANCE = numpy.var(lorenz_soln[0:INPUT_DIMENSIONS, :])

# print all variables for reference
print("dt", dt)
print("WARMUP_TIME", WARMUP_TIME)
print("TRAIN_TIME", TRAIN_TIME)
print("TEST_TIME", TEST_TIME)
print("MAX_TIME", MAX_TIME)
print("LYAPUNOV_TIME", LYAPUNOV_TIME)
print("FULL_WARMUP_TIME", FULL_WARMUP_TIME)
print("FULL_TRAIN_TIME", FULL_TRAIN_TIME)
print("FULL_WARMUP_TRAIN_TIME", FULL_WARMUP_TRAIN_TIME)
print("FULL_TEST_TIME", FULL_TEST_TIME)
print("FULL_MAX_TIME", FULL_MAX_TIME)
print("FULL_LYAPUNOV_TIME", FULL_LYAPUNOV_TIME)
print("INPUT_DIMENSIONS", INPUT_DIMENSIONS)
print("TIME_DELAY_STEPS", TIME_DELAY_STEPS)
print("DIM_LINEAR", DIM_LINEAR)
print("DIM_NONLINEAR", DIM_NONLINEAR)
print("DIM_TOTAL", DIM_TOTAL)
print("RIDGE_PARAMETER", RIDGE_PARAMETER)
print("t_eval", t_eval)
print("INPUT_VARIANCE", INPUT_VARIANCE)
```

```
data length 220501
dt 0.01
WARMUP_TIME 500
TRAIN_TIME 100000
TEST_TIME 120000
MAX_TIME 220500
LYAPUNOV_TIME 1.104
FULL_WARMUP_TIME 500
FULL_TRAIN_TIME 100000
FULL_WARMUP_TRAIN_TIME 100500
FULL_TEST_TIME 120000
FULL_MAX_TIME 220500
FULL_LYAPUNOV_TIME 1
INPUT_DIMENSIONS 3
TIME_DELAY_STEPS 2
DIM_LINEAR 6
DIM_NONLINEAR 21
DIM_TOTAL 28
RIDGE_PARAMETER 2.5e-06
t_eval [0.00000e+00 1.00000e+00 2.00000e+00 ... 2.20498e+05 2.20499e+05
 2.20500e+05]
INPUT_VARIANCE 200.74250237942252
```

In [ ]:
```
def Fill_linear_part(linearFeatureVector):
    for delay in range(TIME_DELAY_STEPS):
        for j in range(delay, FULL_MAX_TIME):
            linearFeatureVector[INPUT_DIMENSIONS * delay : INPUT_DIMENSIONS
    return linearFeatureVector
```

In [ ]:
```
def Fill_nonlinear_part(linearFeatureVector, out_train):
    count = 0
    for row in range(DIM_LINEAR):
        for column in range(row, DIM_LINEAR):
        # shift by one for constant
            out_train[DIM_LINEAR+1+count] = linearFeatureVector[row, FULL_WA
            count += 1
    return out_train
```

In [ ]:
```
def RMSE(linearFeatureVector, x_predict):
    rootMeanSquaredError = numpy.sqrt(
    numpy.mean((linearFeatureVector[0 : INPUT_DIMENSIONS, FULL_WARMUP_TIME :
    print('training nrmse: '+str(rootMeanSquaredError))
    return rootMeanSquaredError
```

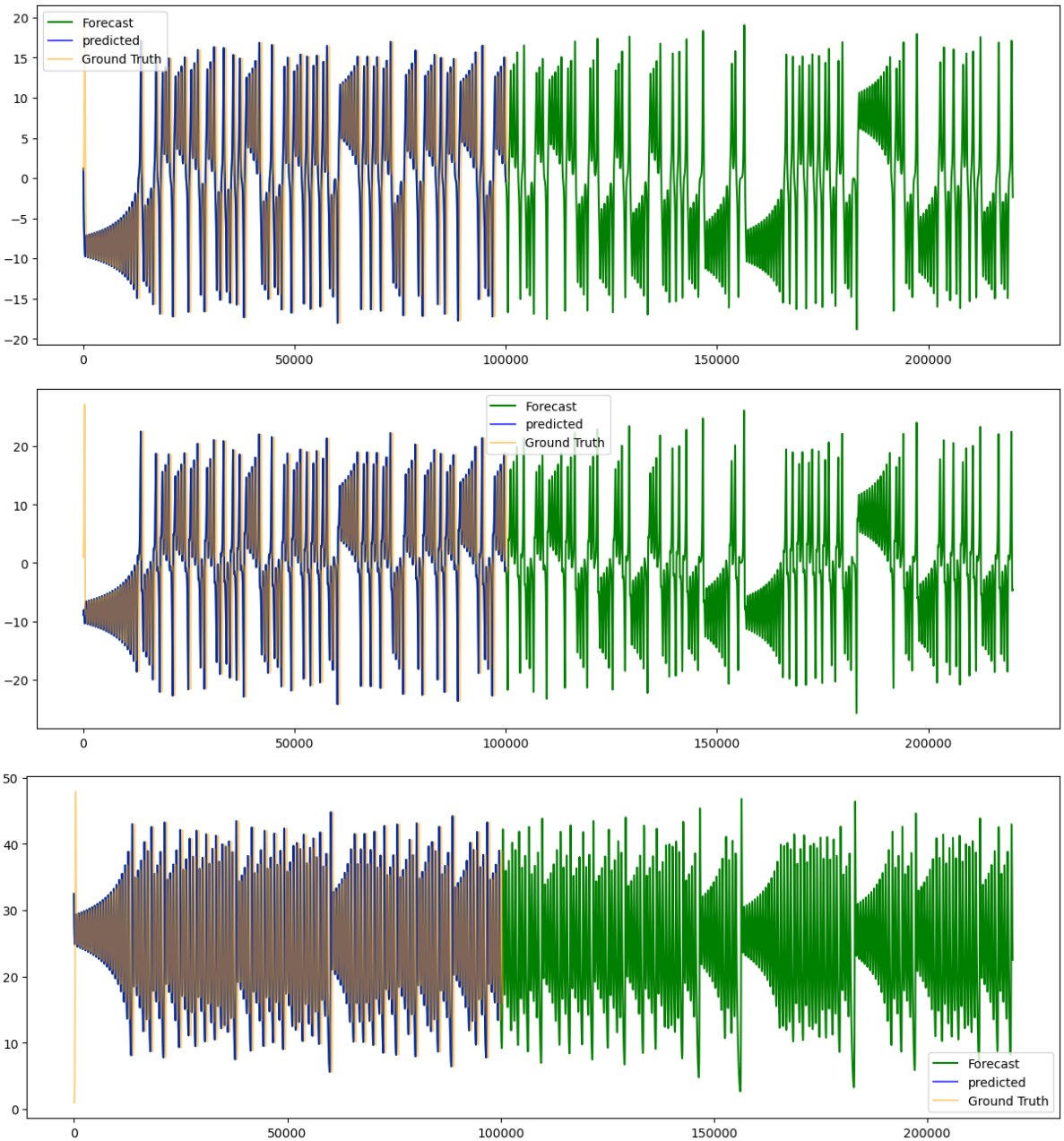
In [ ]:
```
# create an array to hold the linear part of the feature vector
linearFeatureVector = numpy.zeros((DIM_LINEAR, FULL_MAX_TIME))
# create an array to hold the full feature vector for training time
# (use ones so the constant term is already 1)
out_train = numpy.ones((DIM_TOTAL, FULL_TRAIN_TIME))
# create a place to store feature vectors for prediction
out_test = numpy.zeros(DIM_TOTAL)                    # full feature vector
x_test = numpy.zeros((DIM_LINEAR, FULL_TEST_TIME))  # linear part
# copy over the linear part (shift over by one to account for constant)
out_train[1:DIM_LINEAR + 1, :] = linearFeatureVector[:, FULL_WARMUP_TIME - 1

# fill in the linear part of the feature vector for all times
linearFeatureVector = Fill_linear_part(linearFeatureVector)
# fill in the non-linear part
out_train = Fill_nonlinear_part(linearFeatureVector, out_train)


# ridge regression: train W_out to map out_train to Lorenz[t] - Lorenz[t - 1
W_out = (linearFeatureVector[0:INPUT_DIMENSIONS, FULL_WARMUP_TIME:FULL_WARMU
```

```
        1:FULL_WARMUP_TRAIN_TIME-1]) @ out_train[:, :].T @ numpy.linalg.pin
# apply W_out to the training feature vector to get the training output
x_predict = linearFeatureVector[0:INPUT_DIMENSIONS, FULL_WARMUP_TIME-1:FULL_
    W_out @  out_train[:, 0:FULL_TRAIN_TIME]

# calculate NRMSE between true Lorenz and training output
rootMeanSquaredError = RMSE(linearFeatureVector, x_predict)

# copy over initial linear feature vector
x_test[:, 0] = linearFeatureVector[:, FULL_WARMUP_TRAIN_TIME-1]
# calculate NRMSE between true Lorenz and prediction for one Lyapunov time
test_nrmse = numpy.sqrt(numpy.mean(
    (linearFeatureVector[0:INPUT_DIMENSIONS, FULL_WARMUP_TRAIN_TIME-1:FULL_W
print('test nrmse: '+str(test_nrmse))
```

```
training nrmse: 0.0007285849023907341
test nrmse: 0.0
```

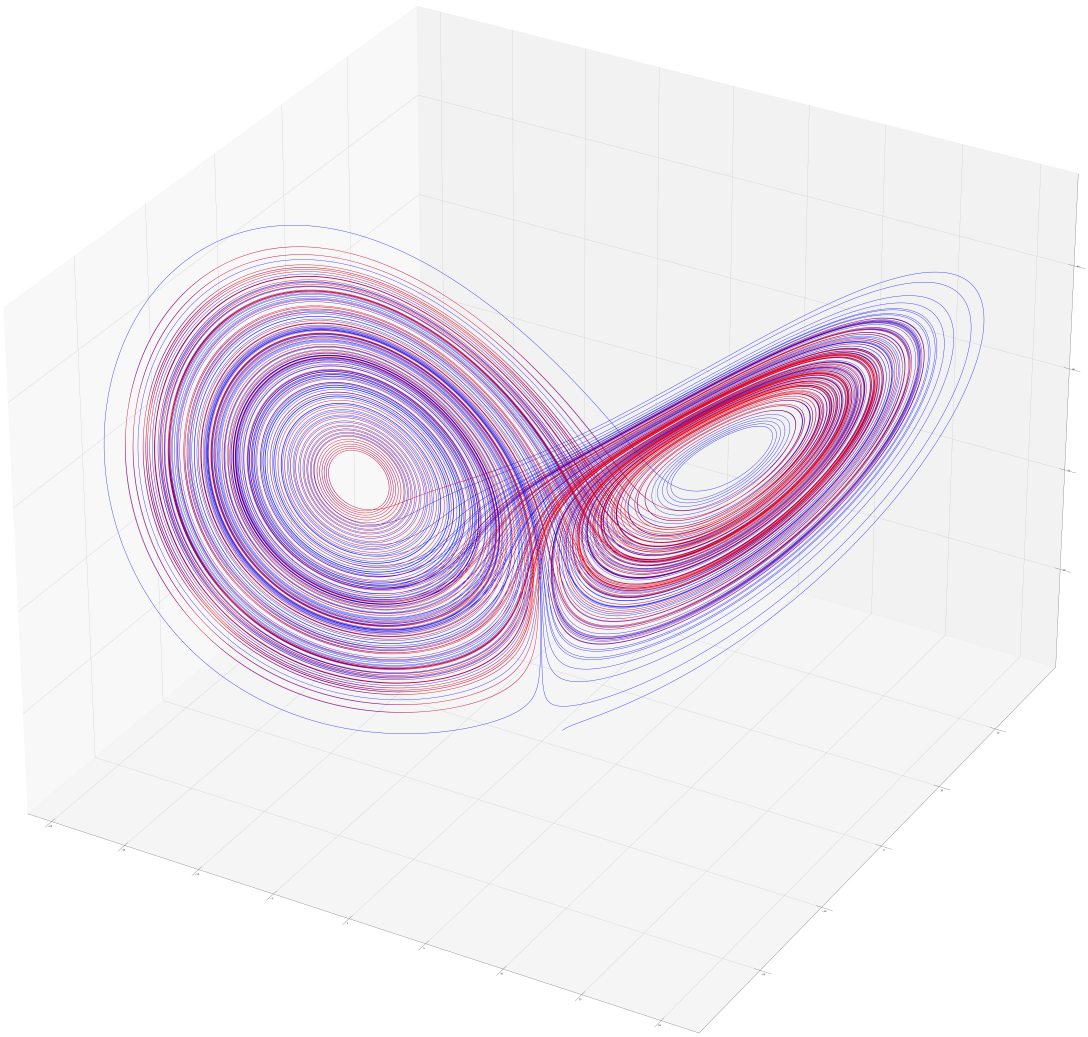
In [ ]:

```
plotTimeSeries(x_predict[0 ,:], linearFeatureVector[0 ,FULL_WARMUP_TIME:] ,
plotTimeSeries(x_predict[1 ,:], linearFeatureVector[1, FULL_WARMUP_TIME:] ,
plotTimeSeries(x_predict[2 ,:], linearFeatureVector[2, FULL_WARMUP_TIME:] ,
```



Red - Reconstruction: Blue - Ground State

In [ ]:
```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import numpy as np

def plot_3d_figure(x1, y1, z1, x2, y2, z2):
    # Initialize the plot
    fig = plt.figure(figsize=(100, 100))
    ax = fig.add_subplot(111, projection='3d')

    # Plot the first figure
    ax.plot3D(x1, y1, z1, c='blue')

    # Plot the second figure
    ax.plot3D(x2, y2, z2, c='red')

    # Show the plot
    plt.show()

# Call the function
plot_3d_figure(dataPoints[0], dataPoints[1], dataPoints[2],
               x_predict[0, :], x_predict[1, :], x_predict[2, :])
```



Data Assimilation Methods

- Adjacent Four Dimensional Variational Data Assimilation Method - PyDA code - not very efficient
- update plotting

In [ ]:
```
def RK4(rhs,state,dt,*args):

    k1 = rhs(state,*args)
    k2 = rhs(state+k1*dt/2,*args)
    k3 = rhs(state+k2*dt/2,*args)
    k4 = rhs(state+k3*dt,*args)

    new_state = state + (dt/6)*(k1+2*k2+2*k3+k4)
    return new_state


def JRK4(rhs,Jrhs,state,dt,*args):

    n = len(state)
    k1 = rhs(state,*args)
    k2 = rhs(state+k1*dt/2,*args)
    k3 = rhs(state+k2*dt/2,*args)
    #k4 = rhs(state+k3*dt,*args)

    dk1 = Jrhs(state,*args)
    dk2 = Jrhs(state+k1*dt/2,*args) @ (numpy.eye(n)+dk1*dt/2)
    dk3 = Jrhs(state+k2*dt/2,*args) @ (numpy.eye(n)+dk2*dt/2)
    dk4 = Jrhs(state+k3*dt,*args) @ (numpy.eye(n)+dk3*dt)

    DM = numpy.eye(n) + (dt/6) * (dk1+2*dk2+2*dk3+dk4)
    return DM
```

In [ ]:
```
# -*- coding: utf-8 -*-
"""
examples file for a tutorial paper on data assimilation
"PyDA: A hands-on introduction to dynamical data assimilation with Python"
@authors: Shady E. Ahmed, Suraj Pawar, Omer San
"""

import numpy as np

def Lorenz63(state,*args): #Lorenz 96 model
    #rho = 28.0     #sigma = 10.0     #beta = 8.0 / 3.0

    sigma = args[0]
    beta = args[1]
    rho = args[2]
    x, y, z = state #Unpack the state vector
    f = numpy.zeros(3) #Derivatives
    f[0] = sigma * (y - x)
    f[1] = x * (rho - z) - y
    f[2] = x * y - beta * z
    return f

def JLorenz63(state,*args): #Jacobian of Lorenz 96 model
    sigma = args[0]
    beta = args[1]
    rho = args[2]
    x, y, z = state #Unpack the state vector
    df = np.zeros([3,3]) #Derivatives
    df[0,0] = sigma * (-1)
    df[0,1] = sigma * (1)
```

```
    df[0,2] = sigma * (0)
    df[1,0] = 1 * (rho - z)
    df[1,1] = -1
    df[1,2] = x * (-1)
    df[2,0] = 1 * y
    df[2,1] = x * 1
    df[2,2] = - beta
    return df
```

In [ ]:
```
def Adj4dvar(rhs, Jrhs, ObsOp, JObsOp, t, ind_m, u0b, w, R, opt, *args):

    n = len(u0b)
    # determine the assimilation window
    t = t[:ind_m[-1]+1]  # cut the time till the last observation point
    nt = len(t)-1
    dt = t[1] - t[0]
    ub = numpy.zeros([n, nt+1])  # base trajectory
    lam = numpy.zeros([n, nt+1])  # lambda sequence
    fk = numpy.zeros([n, len(ind_m)])

    Ri = numpy.linalg.inv(R)

    ub[:, 0] = u0b
    if opt == 1:  # RK4
        # forward model
        for k in range(nt):
            ub[:, k+1] = RK4(rhs, ub[:, k], dt, *args)

        # backward adjoint
        k = ind_m[-1]
        fk[:, -1] = (JObsOp(ub[:, k])).T @ Ri @ (w[:, -1] - ObsOp(ub[:, k]))
        lam[:, k] = fk[:, -1]  # lambda_N = f_N

        km = len(ind_m)-2
        for k in range(ind_m[-1], 0, -1):
            DM = JRK4(rhs, Jrhs, ub[:, k-1], dt, *args)
            lam[:, k-1] = (DM).T @ lam[:, k]
            if k-1 == ind_m[km]:
                fk[:, km] = (JObsOp(ub[:, k-1])
                             ).T @ Ri @ (w[:, km] - ObsOp(ub[:, k-1]))
                lam[:, k-1] = lam[:, k-1] + fk[:, km]
                km = km - 1

    dJ0 = -lam[:, 0]
    return dJ0
```

In [ ]:
```
# Loss function (w-h(u))^T * R^{-1} * (w-h(u))
def loss(rhs, ObsOp, t, ind_m, u0, w, R, opt, *args):

    n = len(u0)
    # determine the assimilation window
    t = t[:ind_m[-1]+1]  # cut the time till the last observation point
    nt = len(t)-1
    dt = t[1] - t[0]
    u = numpy.zeros([n, nt+1])  # trajectory

    u[:, 0] = u0

    Ri = numpy.linalg.inv(R)
    floss = 0
    km = 0
    nt_m = len(ind_m)
    if opt == 1:  # RK4
```

```
        # forward model
        for k in range(nt):
            u[:, k+1] = RK4(rhs, u[:, k], dt, *args)
            if (km < nt_m) and (k+1 == ind_m[km]):
                tmp = w[:, km] - ObsOp(u[:, k+1])
                tmp = tmp.reshape(-1, 1)
                floss = floss + numpy.linalg.multi_dot((tmp.T, Ri, tmp))
                km = km + 1

    floss = floss[0, 0]/2
    return floss
```

In [ ]:
```
def GoldenAlpha(p, rhs, ObsOp, t, ind_m, u0, w, R, opt, *args):

    # p is the optimization direction
    a0 = 0
    b0 = 1
    r = (3-numpy.sqrt(5))/2

    uncert = 1e-5  # Specified uncertainty

    a1 = a0 + r*(b0-a0)
    b1 = b0 - r*(b0-a0)
    while (b0-a0) > uncert:

        if loss(rhs, ObsOp, t, ind_m, u0+a1*p, w, R, opt, *args) < loss(rhs,
            b0 = b1
            b1 = a1
            a1 = a0 + r*(b0-a0)
        else:
            a0 = a1
            a1 = b1
            b1 = b0 - r*(b0-a0)
    alpha = (b0+a0)/2

    return alpha
```

In [ ]:
```
sigma = 10.0
beta = 8.0/3.0
rho = 28.0
dt = 0.01
tm = 10
nt = int(tm/dt)
t = numpy.linspace(0, tm, nt+1)
```

In [ ]:
```
############################ Twin experiment ################################
# Observation operator

def h(u):
    w = u
    return w


def Dh(u):
    n = len(u)
    D = numpy.eye(n)
    return D
```

In [ ]:
```
u0True = numpy.array([1, 1, 1])  # True initial conditions
numpy.random.seed(seed=1)
sig_m = 0.15  # standard deviation for measurement noise
R = sig_m**2*numpy.eye(3)  # covariance matrix for measurement noise
```

```
dt_m = 0.2  # time period between observations
tm_m = 2  # maximum time for observations
nt_m = int(tm_m/dt_m)  # number of observation instants

ind_m = (numpy.linspace(int(dt_m/dt), int(tm_m/dt), nt_m)).astype(int)
t_m = t[ind_m]

# time integration
uTrue = numpy.zeros([3, nt+1])
uTrue[:, 0] = u0True
km = 0
w = numpy.zeros([3, nt_m])
for k in range(nt):
    uTrue[:, k+1] = RK4(Lorenz63, uTrue[:, k], dt, sigma, beta, rho)
    if (km < nt_m) and (k+1 == ind_m[km]):
        w[:, km] = h(uTrue[:, k+1]) + numpy.random.normal(0, sig_m, [3, ])
        km = km+1

########################## Data Assimilation ##############################
u0b = numpy.array([2.0, 3.0, 4.0])
u0a = u0b
J0 = loss(Lorenz63, h, t, ind_m, u0a, w, R, 1, sigma, beta, rho)
for iter in range(200):

    # computing the gradient of cost functional with base trajectory
    dJ = Adj4dvar(Lorenz63, JLorenz63, h, Dh, t, ind_m,
                  u0a, w, R, 1, sigma, beta, rho)
    # minimization direction
    p = -dJ/numpy.linalg.norm(dJ)
    # Golden method for linesearch
    alpha = GoldenAlpha(p, Lorenz63, h, t, ind_m, u0a,
                        w, R, 1, sigma, beta, rho)
    # update initial condition with gradient descent
    u0a = u0a + alpha*p

    J = loss(Lorenz63, h, t, ind_m, u0a, w, R, 1, sigma, beta, rho)

    if numpy.abs(J0-J) < 1e-2:
        print('Convergence: loss function')
        print(iter)
        break
    else:
        J0 = J
    if numpy.linalg.norm(dJ) < 1e-4:
        print('Convergence: gradient of loss function')
        print(iter)
        break
```

```
Convergence: loss function
125
```

In [ ]:
```
ub = numpy.zeros([3,nt+1])
ub[:,0] = u0b
ua = numpy.zeros([3,nt+1])
ua[:,0] = u0a
km = 0
for k in range(nt):
    ub[:,k+1] = RK4(Lorenz63,ub[:,k],dt,sigma,beta,rho)
    ua[:,k+1] = RK4(Lorenz63,ua[:,k],dt,sigma,beta,rho)
```

In [ ]:
```
font = {'family' : 'normal',
        'weight' : 'bold',
        'size'   : 20}
```

```
fig, ax = matplotlib.pyplot.subplots(nrows=3,ncols=1, figsize=(10,8))
ax = ax.flat

for k in range(3):
    ax[k].plot(t,uTrue[k,:], label='True', linewidth = 3)
    ax[k].plot(t,ub[k,:], ':', label = 'Background', linewidth = 3)
    ax[k].plot(t[ind_m],w[k,:], 'o', fillstyle = 'none', \
               label= 'Observation', markersize = 8, markeredgewidth = 2)
    ax[k].plot(t,ua[k,:], '--', label = 'Analysis', linewidth = 3)
    ax[k].set_xlabel('t',fontsize=22)
    ax[k].axvspan(0, tm_m, color='y', alpha=0.4, lw=0)

ax[0].legend(loc="center", bbox_to_anchor=(0.5,1.25),ncol =4,fontsize=15)

ax[0].set_ylabel('x(t))', labelpad=5)
ax[1].set_ylabel('y(t)', labelpad=-12)
ax[2].set_ylabel('z(t)')
fig.subplots_adjust(hspace=0.5)
```