NumSA

Release 0.0.1

Umberto Zerbinati

Dec 10, 2021

CONTENTS

1 NumSA **1**
- 1.1 Getting Started . . . 1
- 1.2 Bibliography . . . 2

2 TensorFlow **3**
- 2.1 Hessian Example . . . 3
- 2.2 MNIST Neural Network Example . . . 10
- 2.3 Random SVD . . . 15
- 2.4 Newton Method . . . 18
- 2.5 Federated Newton Learn . . . 28
- 2.6 Physically Informed Neural Network . . . 40

3 Partial Differential Equations (PDE) **57**
- 3.1 Eigenvalue Problem (NGS) . . . 57
- 3.2 Pattern Formation PDE (FD) . . . 60

4 Indices **79**

CHAPTER

ONE

NUMSA

NumSA is a highly modular numerical analysis toolbox I developed during my stay at, King Abdullah University Of Science and Technology 2020. NumSA rely on the following dependencies to work: - PyBind11, this it the C++ Python binder, more info here, - Eigen, this is the linear algebra library we decied to use, more info here,

1.1 Getting Started

1.1.1 PyBind11

To compile pybind11 move in the pybind11 folder (dep/pybind11) and use the following commands,

```
mkdir build
cd build
cmake ..
make check -j 4
```

1.1.2 Eigen

To compile QHull move in the pybind11 folder (dep/QHull) and use the following commands,

```
mkdir build
cd build
cmake ..
make install
```

last operation might require root privileges. An other possibility in Debian system is to install the package using the following command,

```
apt-get install libeigen3-dev
```

last operation might require root privileges.

1.1.3 TFHessian

The TFHessian package inside NumSA require tensorflow, numpy and scipy to work. When installing NumSA python package with the option Hessian pip will take care of installing all requirements. To install NumSA with the Hessian option run the following command inside the py folder,

```
pip install -e .[Hessian]
```

1.1.4 FEM

Suggested PETSc configuration,

```
./configure --with-cc=gcc --with-cxx=g++ --with-fc=gfortran --download-fblaslapack --
↪download-openmpi --download-mumps --download-scalapack --download-parmetis --download-
↪metis --with-petsc4py=1
make all
make check
```

while SLEPc suggested configuration,

```
./configure --with-slepc4py=1
```

1.2 Bibliography

1. Halko, N., Martinsson, P.G. and Tropp, J.A., 2011. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. SIAM review, 53(2), pp.217-288.
2. Safaryan, M., Islamov, R., Qian, X. and Richtárik, P., 2021. FedNL: Making Newton-Type Methods Applicable to Federated Learning. arXiv preprint arXiv:2106.02969.
3. He, J., Li, L., Xu, J. and Zheng, C., 2018. Relu deep neural networks and linear finite elements. arXiv preprint arXiv:1807.03973.
4. Yu, B., 2017. The deep Ritz method: a deep learning-based numerical algorithm for solving variational problems. arXiv preprint arXiv:1710.00211.

CHAPTER

TWO

TENSORFLOW

2.1 Hessian Example

2.1.1 Cubic Lost Function and Taylor Expansion

Here it is shown how to compute the Hessian of a lost function delcared using TensorFlow. Let us consider the following lost function,

$$\mathcal{L}(x,y) = 2x^3y^3$$

$$\nabla\mathcal{L}(x,y) = \begin{bmatrix} 6x^2y^3 \\ 6x^3y^2 \end{bmatrix}$$

$$\mathcal{H}(x,y) = \begin{bmatrix} 12xy^3 & 18x^2y^2 \\ 18x^2y^2 & 12x^3y \end{bmatrix}$$

now thanks to the Taylor expansion the Hessian is used to approximate the lost function.

$$\mathcal{L}(x^*,y^*) = \mathcal{L}(x_0,y_0) + \nabla\mathcal{L}(x_0,y_0)\cdot\begin{bmatrix}\Delta x\\ \Delta y\end{bmatrix} + \begin{bmatrix}\Delta x\\ \Delta y\end{bmatrix}\cdot\mathcal{H}(x_0,y_0)\begin{bmatrix}\Delta x\\ \Delta y\end{bmatrix} + \mathcal{O}(\max\{\Delta x,\Delta y\}^3)$$

[2]:
```
#We import all the library we are gona need
import tensorflow as tf
import numpy as np
%matplotlib inline
import matplotlib.pyplot as plt
from numsa.TFHessian import *
```

[33]:
```
#Defining the Loss Function
def Loss(X):
    return 2*X[0]**3*X[1]**3;
#Defining the Hessian class for the above loss function in x
x0 = tf.Variable([1.0,1.0])
H =  Hessian(Loss,x0)
```

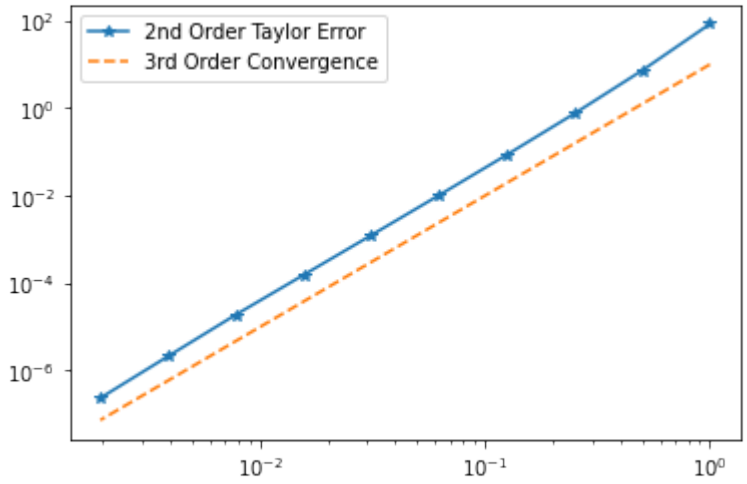
[3]:
```
Error = [];
ErrorH = [];
N = 10
for n in range(N):
    h = 1/(2**n);
```

(continues on next page)

(continued from previous page)

```
    ErrorH = ErrorH + [h];
    x = tf.Variable([1.0+h,1.0+h]);
    v = tf.Variable([h,h]);
    Grad, Hv = H.action(v,True)
    err = abs(Loss(x)-Loss(x0)-tf.tensordot(Grad,v,1)
    -0.5*tf.tensordot(v,Hv,1));
    Error = Error + [err];
plt.loglog(ErrorH,Error,"*-")
plt.loglog(ErrorH,[10*h**3 for h in ErrorH],"--")
plt.legend(["2nd Order Taylor Error","3rd Order Convergence"])
order = (tf.math.log(Error[7])-tf.math.log(Error[9]))/(np.log(ErrorH[7])-np.
↪log(ErrorH[9]))
print("Convergence order {}".format(order))
```

```
Convergence order 3.160964012145996
```



$$\mathcal{H}(1,1) = \begin{bmatrix} 12 & 18 \\ 18 & 12 \end{bmatrix}, \qquad \lambda_1 = 30 \qquad \lambda_2 = -6$$

[4]:
```
H.eig("pi-max") #Power iteration to find the maximum eigenvalue.
```

[4]:
```
<tf.Tensor: shape=(), dtype=float32, numpy=30.000002>
```

2.1.2 Very Small Neural Network Built With KERAS

[5]:
```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from numsa.TFHessian import *

from tqdm.notebook import tqdm
```

[6]:
```
Ord = 1;
optimizer = tf.keras.optimizers.SGD(learning_rate=1e-3)
init = tf.keras.initializers.RandomNormal(mean=0.0, stddev=0.05, seed=None)
# Inputs
training_data = np.array([[0,0],[0,1],[1,0],[1,1]], "float32")
# Outputs

training_label = np.array([[0],[1],[1],[0]], "float32")
training_dataset = tf.data.Dataset.from_tensor_slices((training_data, training_label))
model = tf.keras.Sequential([
    tf.keras.layers.Dense(2,input_dim=2, activation='softmax',bias_initializer=init),#2␣
↪nodes hidden layer
    tf.keras.layers.Dense(1)
])
def loss_fn(y,x):
    return tf.math.reduce_mean(tf.math.squared_difference(y, x));
def Loss(weights):
    predictions = model(training_data, training=True) #Logits for this minibatch
    # Compute the loss value for this minibatch.
    loss_value = loss_fn(training_label, predictions);
    return loss_value;
for epoch in tqdm(range(100)):
    for step, (x,y) in enumerate(training_dataset):
        if Ord == 2:
            # Compute Hessian and Gradients
            H = Hessian(Loss,model.trainable_weights,"KERAS")
            fullH, grad = H.mat(model.trainable_weights,grad=True);
            #Reshaping the Hessians
            grads = [tf.Variable(grad[0:4].reshape(2,2),dtype=np.float32),
                     tf.Variable(grad[4:6].reshape(2,),dtype=np.float32),
                     tf.Variable(grad[6:8].reshape(2,1),dtype=np.float32),
                     tf.Variable(grad[8].reshape(1,),dtype=np.float32),]
        if Ord == 1:
            with tf.GradientTape() as tape:
                # Run the forward pass of the layer.
                # The operations that the layer applies
                # to its inputs are going to be recorded
                # on the GradientTape.
                labels = model(training_data, training=True)  # Logits for this minibatch
                # Compute the loss value for this minibatch.
                loss_value = loss_fn(training_label, labels)
                # Use the gradient tape to automatically retrieve
                # the gradients of the trainable variables with respect to the loss.
                grads = tape.gradient(loss_value, model.trainable_weights)
        optimizer.apply_gradients(zip(grads, model.trainable_weights))
for step, (x,y) in enumerate(training_dataset):
    print("Test {} with data {} produce {} .".format(step,x,tf.math.round(model(np.
↪array([x])))))
```

```
  0%|          | 0/100 [00:00<?, ?it/s]
```

```
Test 0 with data [0. 0.] produce [[0.]] .
Test 1 with data [0. 1.] produce [[0.]] .
Test 2 with data [1. 0.] produce [[0.]] .
```

(continues on next page)

(continued from previous page)

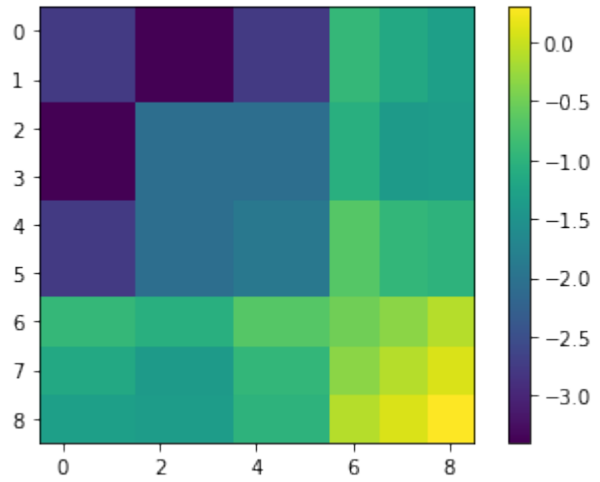
```
Test 3 with data [1. 1.] produce [[0.]] .
```

[7]:
```
print("Number of trainable layers {}".format(len(model.trainable_weights)))
print("Number of weights trainable per layer 0, {}".format(model.trainable_weights[0].
↪shape))
print("Number of weights trainable per layer 1, {}".format(model.trainable_weights[1].
↪shape))
print("Number of weights trainable per layer 2, {}".format(model.trainable_weights[2].
↪shape))
print("Number of weights trainable per layer 3, {}".format(model.trainable_weights[3].
↪shape))
```

```
Number of trainable layers 4
Number of weights trainable per layer 0, (2, 2)
Number of weights trainable per layer 1, (2,)
Number of weights trainable per layer 2, (2, 1)
Number of weights trainable per layer 3, (1,)
```

[8]:
```
#Function written for parallel
def Loss(weights,comm):
    batches = np.array_split(training_data,comm.Get_size());
    predictions = model(batches[comm.Get_rank()], training=True) #Logits for this␣
↪minibatch
    # Compute the loss value for this minibatch.
    loss_value = loss_fn(training_label, predictions);
    return loss_value;

#Function written for serial
def Loss(weights):
    predictions = model(training_data, training=True) #Logits for this minibatch
    # Compute the loss value for this minibatch.
    loss_value = loss_fn(training_label, predictions);
    return loss_value;

H = Hessian(Loss,model.trainable_weights,"KERAS")
H.SwitchVerbose(True)
fullH= H.mat();
plt.imshow((np.log10(abs(fullH)+1e-16)))
plt.colorbar()
print(np.min(abs(fullH)))
```

```
MPI the world is 1 process big !
0.00039930374
```



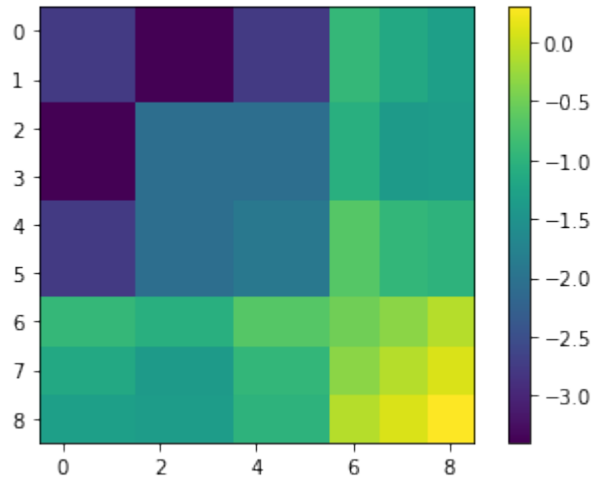
[9]:
```
fullH= H.matrix("KERAS");
plt.imshow((np.log10(abs(fullH)+1e-16)))
plt.colorbar()
print(np.min(abs(fullH)))
```

```
  0%|          | 0/4 [00:00<?, ?it/s]
```

```
MPI the world is 1 process big !
```

```
100%|| 4/4 [00:00<00:00, 38.37it/s]
100%|| 2/2 [00:00<00:00, 43.11it/s]
100%|| 2/2 [00:00<00:00, 42.38it/s]
100%|| 1/1 [00:00<00:00, 42.90it/s]
```

```
0.00039993037389591336
```

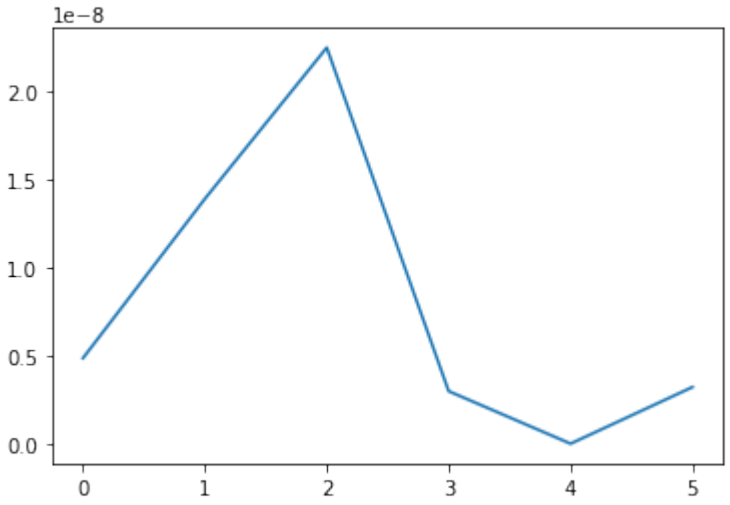


[10]:
```
w = np.random.rand(9,1)
print((abs(fullH@w-(H.vecprod(w).reshape(9,1)))<1e-6).all())
```

```
True
```

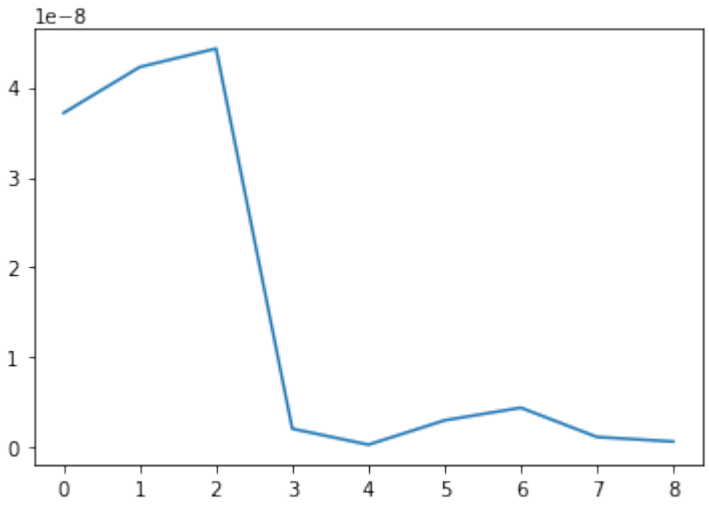
[11]:
```
_,sigmas,_ = np.linalg.svd(fullH)
_,Rsigmas,_ = H.RandMatSVD(4,2);
plt.plot([abs(Rsigmas[r]-sigmas[r]) for r in range(len(Rsigmas))])
```

[11]:
```
[<matplotlib.lines.Line2D at 0x7f336d4e3100>]
```



[12]:
```
_,KRsigmas,_ = H.RandMatSVD(4,2,Krylov=3);
plt.plot([abs(KRsigmas[r]-sigmas[r]) for r in range(len(KRsigmas))])
```

[12]:
```
[<matplotlib.lines.Line2D at 0x7f336d445f10>]
```



[13]:
```
b = np.ones((9,1));
print(la.eig(fullH)[0])
H.pCG(b,4,2).shape
```

```
[ 3.05964799e+00  4.48098994e-01 -3.85493644e-01  1.37021044e-02
  5.93803975e-04 -1.53009930e-08 -3.46201486e-09 -1.07528117e-09
  1.05662170e-09]
Iteration 0 residual 0.8491937201068991, alpha -5.656056460522596 < 0
```

[13]:
```
(9, 1)
```

2.1.3 Hessian Based Training

[14]:
```
Ord = 1;
optimizer = tf.keras.optimizers.SGD(learning_rate=1e-3)
init = tf.keras.initializers.RandomNormal(mean=0.0, stddev=0.05, seed=None)
# Inputs
training_data = np.array([[0,0],[0,1],[1,0],[1,1]], "float32")
# Outputs

training_label = np.array([[0],[1],[1],[0]], "float32")
training_dataset = tf.data.Dataset.from_tensor_slices((training_data, training_label))
model = tf.keras.Sequential([
    tf.keras.layers.Dense(2,input_dim=2, activation='softmax',bias_initializer=init),#2␣
↪nodes hidden layer
    tf.keras.layers.Dense(1)
])
def loss_fn(y,x):
    return tf.math.reduce_mean(tf.math.squared_difference(y, x));
def Loss(weights):
    predictions = model(training_data, training=True) #Logits for this minibatch
    # Compute the loss value for this minibatch.
    loss_value = loss_fn(training_label, predictions);
    return loss_value;
for epoch in tqdm(range(200)):
    for step, (x,y) in enumerate(training_dataset):
        with tf.GradientTape() as tape:
            # Run the forward pass of the layer.
            # The operations that the layer applies
            # to its inputs are going to be recorded
            # on the GradientTape.
            labels = model(training_data, training=True)  # Logits for this minibatch
            # Compute the loss value for this minibatch.
            loss_value = loss_fn(training_label, labels)
            # Use the gradient tape to automatically retrieve
            # the gradients of the trainable variables with respect to the loss.
            grads = tape.gradient(loss_value, model.trainable_weights)
            H = Hessian(Loss,model.trainable_weights,"KERAS")
            Grad = H.vec(grads).reshape((9,1));
            q = H.pCG(Grad,4,2);
            search = [tf.Variable(q[0:4].reshape(2,2),dtype=np.float32),
                      tf.Variable(q[4:6].reshape(2,),dtype=np.float32),
                      tf.Variable(q[6:8].reshape(2,1),dtype=np.float32),
                      tf.Variable(q[8].reshape(1,),dtype=np.float32),]
    optimizer.apply_gradients(zip(search, model.trainable_weights))
    if sum([abs(y-tf.math.round(model(np.array([x])))) for (x,y) in training_dataset])␣
↪== 0.0:
        break;
for step, (x,y) in enumerate(training_dataset):
    print("Test {} with data {} produce {} .".format(step,x,tf.math.round(model(np.
↪array([x])))))
```

(continues on next page)

(continued from previous page)

```
    print("Test {} with data {} produce {} .".format(step,x,model(np.array([x]))))
```

```
  0%|          | 0/200 [00:00<?, ?it/s]
```

```
Test 0 with data [0. 0.] produce [[0.]] .
Test 0 with data [0. 0.] produce [[0.48190117]] .
Test 1 with data [0. 1.] produce [[0.]] .
Test 1 with data [0. 1.] produce [[0.46368325]] .
Test 2 with data [1. 0.] produce [[1.]] .
Test 2 with data [1. 0.] produce [[0.500142]] .
Test 3 with data [1. 1.] produce [[0.]] .
Test 3 with data [1. 1.] produce [[0.4818064]] .
```

[ ]:

2.2 MNIST Neural Network Example

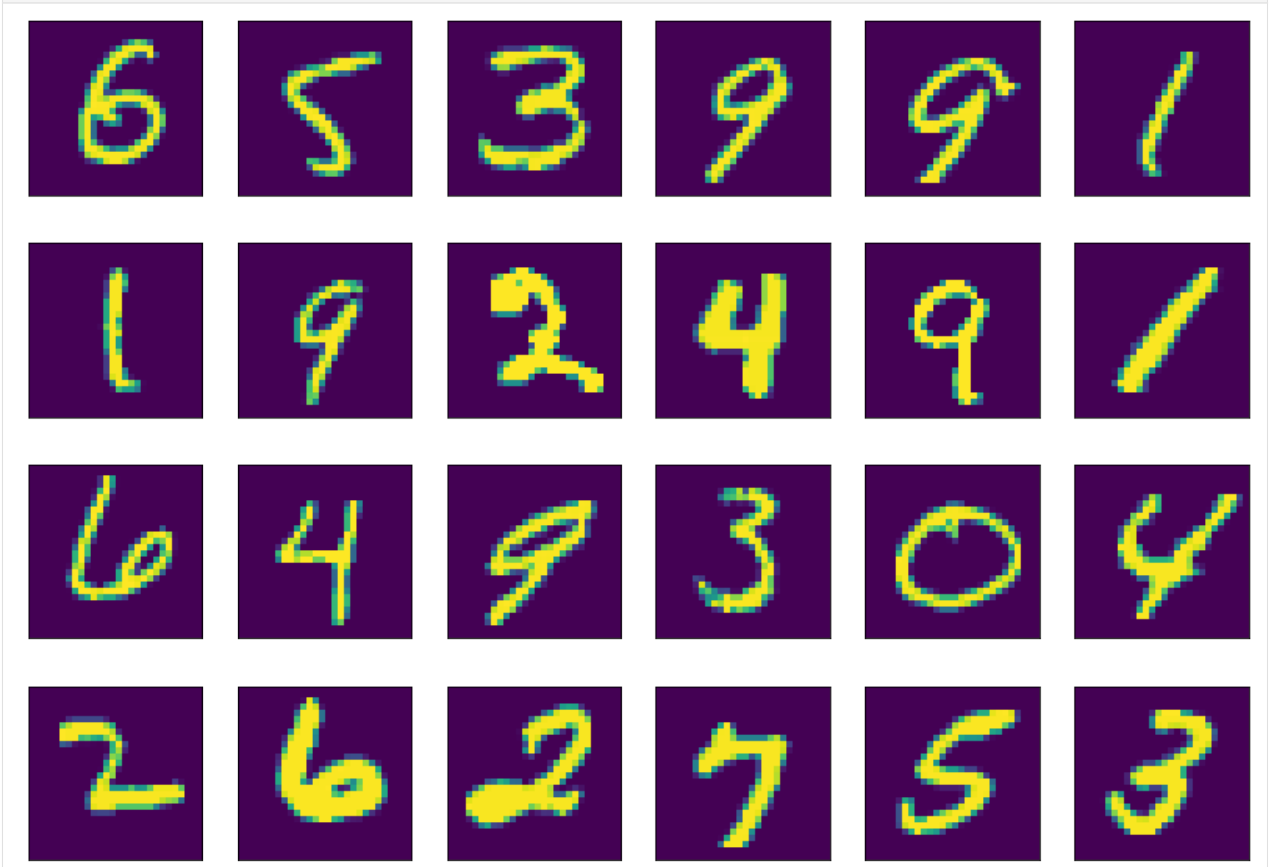
In this notebook we will show how to make NumSA interact with TensorFlow. We will begin building a Neural Network that recognise digit from the MNIST dataset.

[1]:
```
#We import all the library we are gona need
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from numsa.TFHessian import *
from random import *
from tqdm.notebook import tqdm

mnist = tf.keras.datasets.mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
print("In the training set there are {} images, of size {}x{}."
      .format(train_images.shape[0],train_images.shape[1],train_images.shape[2]))
print("The data are labeled in the following categories, {}"
      .format(train_labels))
#We normalize the dataset
train_images = train_images/255.0
test_images = test_images/255.0
#We assemble the data set
# Prepare the training dataset.
batch_size = 64
train_dataset = tf.data.Dataset.from_tensor_slices((train_images, train_labels))
train_dataset = train_dataset.shuffle(buffer_size=1024).batch(batch_size)

# Prepare the validation dataset.
test_dataset = tf.data.Dataset.from_tensor_slices((test_images, test_labels))
test_dataset = test_dataset.batch(batch_size)
```

```
In the training set there are 60000 images, of size 28x28.
The data are labeled in the following categories, [5 0 4 ... 5 6 8]
```

We can plot a random image from the MNIST data set using MatPlotLib.

[2]:
```
plt.figure(figsize=(14,10));
for i in range(24):
    plt.subplot(4,6,i+1)
    rn = randrange(0,train_images.shape[0]-1);
    plt.imshow(train_images[rn])
    plt.grid(False)
    plt.xticks([])
    plt.yticks([])
plt.show()
```



Now we create the proper neural newtorks that we are intrested in studying, in particular to begin with we will sequantialey add two layer of size 128 and 10. But at the beging of the neural network we will put a layer that will "vectorify" the matrix storing the image. We chose as activation function the RELU function to begin with.

[3]:
```
#Keras Sequential allow us to place one layer after the other
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),#Vectorifying layer
    tf.keras.layers.Dense(20, activation='sigmoid'),#128 weights layer
    tf.keras.layers.Dense(10)#10 leyers weights.
])
print("Number of trainable layers {}".format(len(model.trainable_weights)))
print("Number of weights trainable per layer 0, {}".format(model.trainable_weights[0].
↪shape))
print("Number of weights trainable per layer 1, {}".format(model.trainable_weights[1].
↪shape))
```

(continues on next page)

(continued from previous page)

```
print("Number of weights trainable per layer 2, {}".format(model.trainable_weights[2].
↪shape))
print("Number of weights trainable per layer 3, {}".format(model.trainable_weights[3].
↪shape))
```

```
Number of trainable layers 4
Number of weights trainable per layer 0, (784, 20)
Number of weights trainable per layer 1, (20,)
Number of weights trainable per layer 2, (20, 10)
Number of weights trainable per layer 3, (10,)
```

We will now explicitly write the traing loop for the NN, in order to acceses it later when using NumSA.

[4]:
```
# Instantiate an optimizer.
optimizer = tf.keras.optimizers.SGD(learning_rate=1e-3)
#Importing a lost function
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
# Prepare the metrics.
train_acc_metric = tf.keras.metrics.SparseCategoricalAccuracy()
test_acc_metric = tf.keras.metrics.SparseCategoricalAccuracy()

#Defining number of iterations
epochs = 21
for epoch in tqdm(range(epochs)):
    # Iterate over the batches of the dataset.
    for step, (batch_train_images, batch_train_labels) in enumerate(train_dataset):
        # Open a GradientTape to record the operations run
        # during the forward pass, which enables auto-differentiation.
        with tf.GradientTape() as tape:
            # Run the forward pass of the layer.
            # The operations that the layer applies
            # to its inputs are going to be recorded
            # on the GradientTape.
            logits = model(batch_train_images, training=True)  # Logits for this␣
↪minibatch
            # Compute the loss value for this minibatch.
            loss_value = loss_fn(batch_train_labels, logits)
            # Use the gradient tape to automatically retrieve
            # the gradients of the trainable variables with respect to the loss.
            grads = tape.gradient(loss_value, model.trainable_weights)

        # Run one step of gradient descent by updating
        # the value of the variables to minimize the loss.
        optimizer.apply_gradients(zip(grads, model.trainable_weights))
        # Update training metric.
        train_acc_metric.update_state(batch_train_labels, logits)

    # Display metrics at the end of each epoch.
    train_acc = train_acc_metric.result()
    # Reset training metrics at the end of each epoch
    train_acc_metric.reset_states()

    # Run a validation loop at the end of each epoch.
```

(continues on next page)

(continued from previous page)

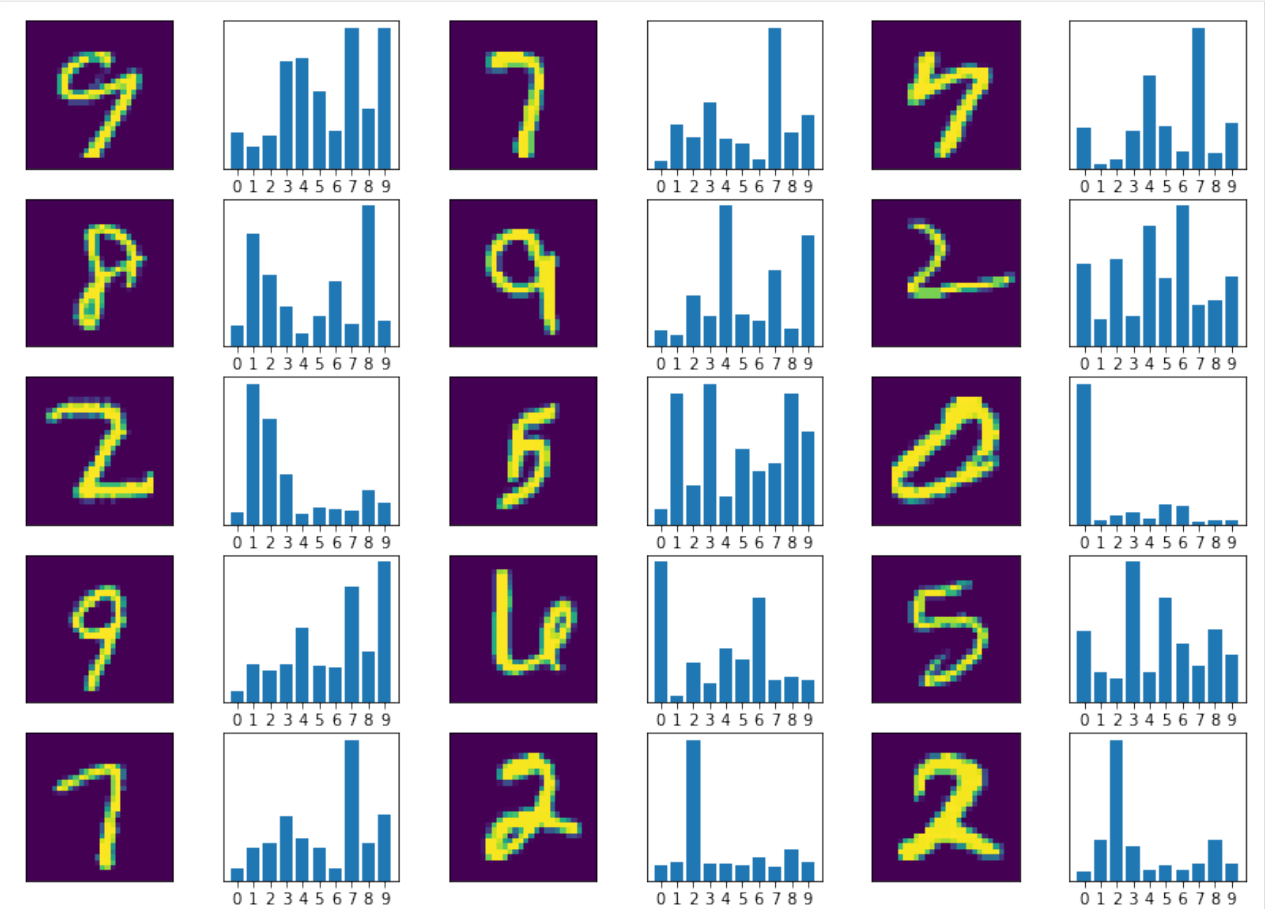
```
    for batch_test_images, batch_test_labels in test_dataset:
        test_logits = model(batch_test_images, training=False)
        # Update val metrics
        test_acc_metric.update_state(batch_test_labels, test_logits)
    test_acc = test_acc_metric.result()
    test_acc_metric.reset_states()
print("Validation acc: %.4f" % (float(test_acc),))
```

```
  0%|          | 0/21 [00:00<?, ?it/s]
```

```
Validation acc: 0.7548
```

Now we test the neural network on a sample digit, randomly selected from the test dataset.

[5]:
```
#Adding a softmax layer to get result in term of probability
pmodel = tf.keras.Sequential([model,tf.keras.layers.Softmax()])
#Evaluating the test data set
predictions = pmodel(test_images);
plt.figure(figsize=(14,10));
for i in range(1,16):
    plt.subplot(5,6,(i*2)-1)
    rn = randrange(0,test_images.shape[0]-1);
    plt.imshow(test_images[rn])
    plt.grid(False)
    plt.xticks([])
    plt.yticks([])
    plt.subplot(5,6,(i*2))
    plt.bar([0,1,2,3,4,5,6,7,8,9],predictions[rn])
    plt.xticks([0,1,2,3,4,5,6,7,8,9])
    plt.yticks([])
plt.show()
```



We now show how to use our package to compute the Hessian of the NN wrt to the weights with a fixed image, first we compute one partial Hessian for each layer of the Neural Network then we compute the the full Hessian of the neural network.

[12]:
```
def Loss(weights):
    logits = model(batch_train_images, training=True) #Logits for this minibatch
    # Compute the loss value for this minibatch.
    loss_value = loss_fn(batch_train_labels, logits);
    return loss_value;
H = Hessian(Loss,model.trainable_weights,"KERAS")
H.SwitchVerbose(True)
fullH= H.mat();
```

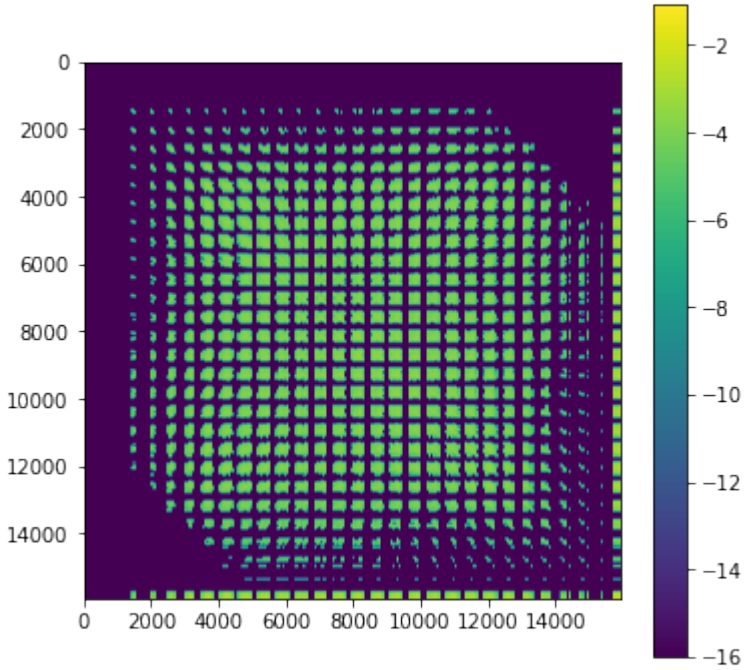
```
  0%|          | 3/15680 [00:00<11:20, 23.04it/s]
```

```
MPI the world is 1 process big !
```

```
100%|| 15680/15680 [09:31<00:00, 27.43it/s]
100%|| 20/20 [00:00<00:00, 24.91it/s]
100%|| 200/200 [00:07<00:00, 25.24it/s]
100%|| 10/10 [00:00<00:00, 25.57it/s]
```

[13]:
```
plt.figure(figsize=(6,6))
plt.imshow(np.log10(abs(fullH)+1e-16))
plt.colorbar()
```

[13]:
```
<matplotlib.colorbar.Colorbar at 0x7f25b14ee2b0>
```



2.3 Random SVD

In this notebook we focus on singular value approximation using randomised range finder approximation as presented in [1]. First we construct a NN to do digit recognition and we then use TFHessian to compute the full Hessian matrix. In particular first we define the layout of the NN.

[1]:
```
#We import all the library we are gona need
import tensorflow as tf
import numpy as np
import scipy.linalg as spla
%matplotlib notebook
import matplotlib.pyplot as plt
from numsa.TFHessian import *
from random import *
from tqdm.notebook import tqdm

mnist = tf.keras.datasets.mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
print("In the training set there are {} images, of size {}x{}."
      .format(train_images.shape[0],train_images.shape[1],train_images.shape[2]))
print("The data are labeled in the following categories, {}"
      .format(train_labels))
#We normalize the dataset
train_images = train_images/255.0
test_images = test_images/255.0
#We assemble the data set
```

(continues on next page)

(continued from previous page)

```
# Prepare the training dataset.
batch_size = 64
train_dataset = tf.data.Dataset.from_tensor_slices((train_images, train_labels))
train_dataset = train_dataset.shuffle(buffer_size=1024).batch(batch_size)

# Prepare the validation dataset.
test_dataset = tf.data.Dataset.from_tensor_slices((test_images, test_labels))
test_dataset = test_dataset.batch(batch_size)

#Keras Sequential allow us to place one layer after the other
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),#Vectorifying layer
    tf.keras.layers.Dense(15, activation='sigmoid'),#128 weights layer
    tf.keras.layers.Dense(10)#10 leyers weights.
])

print("Number of trainable layers {}".format(len(model.trainable_weights)))
print("Number of weights trainable per layer 0, {}".format(model.trainable_weights[0].
↪shape))
print("Number of weights trainable per layer 1, {}".format(model.trainable_weights[1].
↪shape))
print("Number of weights trainable per layer 2, {}".format(model.trainable_weights[2].
↪shape))
print("Number of weights trainable per layer 3, {}".format(model.trainable_weights[3].
↪shape))
```

```
In the training set there are 60000 images, of size 28x28.
The data are labeled in the following categories, [5 0 4 ... 5 6 8]
Number of trainable layers 4
Number of weights trainable per layer 0, (784, 15)
Number of weights trainable per layer 1, (15,)
Number of weights trainable per layer 2, (15, 10)
Number of weights trainable per layer 3, (10,)
```

We now proceed to train the neural network using a sthocastic gradient descent method.

[2]:
```
# Instantiate an optimizer.
optimizer = tf.keras.optimizers.SGD(learning_rate=1e-3)
#Importing a lost function
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
# Prepare the metrics.
train_acc_metric = tf.keras.metrics.SparseCategoricalAccuracy()
test_acc_metric = tf.keras.metrics.SparseCategoricalAccuracy()

#Defining number of iterations
epochs = 21
for epoch in tqdm(range(epochs)):
    # Iterate over the batches of the dataset.
    for step, (batch_train_images, batch_train_labels) in enumerate(train_dataset):
        # Open a GradientTape to record the operations run
        # during the forward pass, which enables auto-differentiation.
        with tf.GradientTape() as tape:
            # Run the forward pass of the layer.
```

(continues on next page)

(continued from previous page)

```
            # The operations that the layer applies
            # to its inputs are going to be recorded
            # on the GradientTape.
            logits = model(batch_train_images, training=True)  # Logits for this␣
↪minibatch
            # Compute the loss value for this minibatch.
            loss_value = loss_fn(batch_train_labels, logits)
            # Use the gradient tape to automatically retrieve
            # the gradients of the trainable variables with respect to the loss.
            grads = tape.gradient(loss_value, model.trainable_weights)

        # Run one step of gradient descent by updating
        # the value of the variables to minimize the loss.
        optimizer.apply_gradients(zip(grads, model.trainable_weights))
        # Update training metric.
        train_acc_metric.update_state(batch_train_labels, logits)

    # Display metrics at the end of each epoch.
    train_acc = train_acc_metric.result()
    # Reset training metrics at the end of each epoch
    train_acc_metric.reset_states()

    # Run a validation loop at the end of each epoch.
    for batch_test_images, batch_test_labels in test_dataset:
        test_logits = model(batch_test_images, training=False)
        # Update val metrics
        test_acc_metric.update_state(batch_test_labels, test_logits)
    test_acc = test_acc_metric.result()
    test_acc_metric.reset_states()
print("Validation acc: %.4f" % (float(test_acc),))
```

```
  0%|          | 0/21 [00:00<?, ?it/s]
```

```
Validation acc: 0.7308
```

[3]:
```
def Loss(weights):
    logits = model(batch_train_images, training=True) #Logits for this minibatch
    # Compute the loss value for this minibatch.
    loss_value = loss_fn(batch_train_labels, logits);
    return loss_value;
#We now use TFHessian to compute the Hessian of the NN.
H = Hessian(Loss,model.trainable_weights,"KERAS")
H.SwitchVerbose(True)
```

[4]:
```
matH= H.mat();
```

```
MPI the world is 1 process big !
```

[5]:
```
sigmas = spla.svdvals(matH)
```

[19]:
```
plt.figure()
plt.title("Hessian's Singular Value For MNIST Neural Network")
```

(continues on next page)

(continued from previous page)

```
plt.semilogy(sigmas,"--")
plt.legend([r"$\sigma_j$"])
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[19]:
```
<matplotlib.legend.Legend at 0x7fb9ec199df0>
```

[6]:
```
_,randsigmas,_ = H.RandMatSVD(50,10);
```

[10]:
```
plt.figure()
plt.title("Hessian's Singular Value rank=50,overfit=10")
plt.plot(randsigmas)
plt.legend([r"$\sigma$",r"$\overset{\sim}{\sigma}$"])
plt.show()
plt.figure()
plt.title("Hessian's Singular Value rank=50,overfit=10")
plt.plot([abs((randsigmas[i]-sigmas[i])/sigmas[i]) for i in range(len(randsigmas))],"--")
plt.legend([r"$|\sigma_j-\overset{\sim}{\sigma}_j|$"])
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[10]:
```
<matplotlib.legend.Legend at 0x7fb9ec64d580>
```

2.4 Newton Method

2.4.1 Quadratic Energy Minimization

We are minimizing the following energy functional, using a Netwon method based on the TF Hessian library.

$$J(x, y) = x^2y^2 + xy$$

which is the unique stationary point of ∇J given the fact that $J(x, y)$ is convex.

[2]:
```
#We import all the library we are gona need
import tensorflow as tf
import numpy as np
import pandas as pd
%matplotlib inline
import matplotlib.pyplot as plt
from numsa.TFHessian import *
```

[4]:
```
#################| SETTINGS |##############
itmax = 10; # Number of epoch.
tol = 1e-6
step_size = 1; #Learning rate
```

(continues on next page)

(continued from previous page)

```
tau = 0.5;
########################################
def Loss(x):
    return (x[0]**2)*(x[1]**2)+x[0]*x[1];
#Defining the Hessian class for the above loss function in x0
x = tf.Variable(0.1*np.ones((2,1),dtype=np.float32))
H =  Hessian(Loss,x)
grad = H.grad().numpy();
print("Lost funciton at this iteration {}, gradient norm {} and is achived at point {}"
      .format(Loss(x),np.linalg.norm(grad),x));
print("Computed the  first gradient ...")
q = H.pCG(grad,1,1,tol=tol,itmax=100);
print("Computed search search diratcion ...")
print("Entering the Netwton optimization loop")
for it in tqdm(range(itmax)):
    x = x - tf.constant(step_size,dtype=np.float32)*tf.Variable(q,dtype=np.float32);
    x =  tf.Variable(x)
    if it%50 == 0:
        print("Lost funciton at this iteration {}  and gradient norm {}".format(Loss(x),
↪np.linalg.norm(grad)));
    if np.linalg.norm(grad)<tol:
        print("Lost funciton at this iteration {}, gradient norm {} and is achived at␣
↪point {}"
      .format(Loss(x),np.linalg.norm(grad),x));
        break
    H =  Hessian(Loss,x)
    grad = H.grad().numpy();
    q = H.pCG(grad,1,1,tol=tol,itmax=100);
    while True:
        step = step_size;
        if Loss(x) > Loss(x - tf.constant(step,dtype=np.float32)*tf.Variable(q,dtype=np.
↪float32)):
            print("[Back Tracking] Choosen step size {}".format(step))
            break;
        if 1 == 1+step:
            break;
        step = tau*step;
```

```
  0%|          | 0/10 [00:00<?, ?it/s]
```

```
Lost funciton at this iteration [0.0101], gradient norm 0.1442497819662094 and is␣
↪achived at point <tf.Variable 'Variable:0' shape=(2, 1) dtype=float32, numpy=
array([[0.1],
       [0.1]], dtype=float32)>
Computed the  first gradient ...
Computed search search diratcion ...
Entering the Netwton optimization loop
Lost funciton at this iteration [1.4240455e-05]  and gradient norm 0.1442497819662094
[Back Tracking] Choosen step size 1
```

```
 20%|        | 2/10 [00:00<00:00, 13.54it/s]
```

```
[Back Tracking] Choosen step size 1
Lost funciton at this iteration [1.7290094e-15], gradient norm 3.074902963362547e-07 and␣
↪is achived at point <tf.Variable 'Variable:0' shape=(2, 1) dtype=float32, numpy=
array([[-7.6709966e-08],
       [-2.2539567e-08]], dtype=float32)>
```

2.4.2 Regression

Our objective is to minimize the function:

$$f(\vec{x}) = \frac{1}{m}\sum_{i=1}^{m} \log\left(1 + \exp\left(-b_j \vec{a_j}^T \vec{x}\right)\right) \qquad for\ x \in \mathbb{R}^d \tag{2.1}$$

where d is the feature number and $\vec{a}_j$ are the data while b_j are the labels. Now we would like to this applying the newton method to find a point that minimize such a function. This is possible because since f is convex, all stationary points are minimizers and we search for the "roots" of the equation $\nabla f = 0$. The newton method we implement is of the form,

$$\vec{x}_{n+1} = \vec{x}_n - \gamma H f(\vec{x}_n)^{-1} \nabla f(\vec{x}_n) \tag{2.2}$$

where γ is the step size. We solve the system $Hf(\vec{x}_n)q = \nabla f(\vec{x}_n)$ using the CG method where as a preconditioned we have taken a the inverse of $Hf(\vec{x}_n)$ computed using the random SVD presented in [1].

[3]:
```
#We import all the library we are gona need
import tensorflow as tf
import numpy as np
import pandas as pd
#%matplotlib inline
import matplotlib.pyplot as plt
from numsa.TFHessian import *
import dsdl
```

[4]:
```
ds = dsdl.load("a1a")

X, Y = ds.get_train()
print(X.shape, Y.shape)
```

```
(1605, 119) (1605,)
```

[5]:
```
#Setting the parameter of this run, we will use optimization nomeclature not ML one.
itmax = 100; # Number of epoch.
tol = 1e-4
step_size = 0.2; #Learning rate
Err = [];
"""
#Old Lost Function, intesead using Stefano's.
#Defining the Loss Function
def Loss(x):
    S = tf.Variable(0.0);
    for j in range(X.shape[0]):
        a = tf.constant((X[j,:].todense().reshape(119,1)),dtype=np.float32);
```

(continues on next page)

(continued from previous page)

```
        b = tf.constant(Y[j],dtype=np.float32)
        a = tf.reshape(a,(119,1));
        x = tf.reshape(x,(119,1));
        dot = tf.matmul(tf.transpose(a),x);
        S = S+tf.math.log(1+tf.math.exp(-b*dot))
    S = (1/X.shape[0])*S;
    return S;
"""
tfX = tf.sparse.from_dense(np.array(X.todense(), dtype=np.float32))
tfY = tf.convert_to_tensor(np.array(Y, dtype=np.float32).reshape(X.shape[0], 1))


#Defining the Loss Function
def Loss(x):
    x = tf.reshape(x, (119, 1))
    Z = tf.sparse.sparse_dense_matmul(tfX, x, adjoint_a=False)
    Z = tf.math.multiply(tfY, Z)
    S = tf.reduce_sum(tf.math.log(1 + tf.math.exp(-Z)) / tfX.shape[0])
    return S

#Defining the Hessian class for the above loss function in x0
x = tf.Variable(0.1*np.ones((119,1),dtype=np.float32))
H =  Hessian(Loss,x)
grad = H.grad().numpy();
print("Computed the  first gradient ...")
q = grad #H.pCG(grad,10,2,tol=1e-3,itmax=10);
print("Computed search search diratcion ...")
for it in tqdm(range(itmax)):
    x = x - tf.constant(step_size,dtype=np.float32)*tf.Variable(q,dtype=np.float32);
    x =  tf.Variable(x)
    if it%50 == 0:
        print("Lost funciton at this iteration {}  and gradient norm {}".format(Loss(x),
↪np.linalg.norm(grad)));
    if np.linalg.norm(grad)<tol:
        break
    H =  Hessian(Loss,x)
    grad = H.grad().numpy();
    q = grad #H.pCG(grad,10,2,tol=1e-3,itmax=10);
itmax = 100; # Number of epoch.
for it in tqdm(range(itmax)):
    x = x - tf.constant(step_size,dtype=np.float32)*tf.Variable(q,dtype=np.float32);
    x =  tf.Variable(x)
    Err = Err + [np.linalg.norm(grad)];
    if it%10 == 0:
        print("Lost funciton at this iteration {}  and gradient norm {}".format(Loss(x),
↪np.linalg.norm(grad)));
    if np.linalg.norm(grad)<tol:
        break
    H =  Hessian(Loss,x)
    grad = H.grad().numpy();
    q = H.pCG(grad,65,10,tol=1e-4,itmax=100);
```

```
 20%|        | 20/100 [00:00<00:00, 193.69it/s]
Computed the  first gradient ...
Computed search search diratcion ...
Lost funciton at this iteration 0.9243690371513367  and gradient norm 1.3872039318084717
 81%|  | 81/100 [00:00<00:00, 199.51it/s]
Lost funciton at this iteration 0.3988267183303833  and gradient norm 0.07041343301534653
100%|| 100/100 [00:00<00:00, 197.61it/s]
  0%|          | 0/100 [00:00<?, ?it/s]
Lost funciton at this iteration 0.37019962072372437  and gradient norm 0.
↪04193235188722610 5
 10%|         | 10/100 [00:35<05:14,  3.50s/it]
Lost funciton at this iteration 0.3072308301925659  and gradient norm 0.
↪006752993445843458
 20%|        | 20/100 [01:10<04:35,  3.44s/it]
Lost funciton at this iteration 0.30269062519073486  and gradient norm 0.
↪0011640462325885892
 30%|       | 30/100 [01:45<04:02,  3.47s/it]
Lost funciton at this iteration 0.30072298645973206  and gradient norm 0.
↪0007762470631860197
 40%|      | 40/100 [02:20<03:26,  3.45s/it]
Lost funciton at this iteration 0.2988802492618561  and gradient norm 0.
↪00027924508322030306
 50%|     | 50/100 [02:54<02:55,  3.51s/it]
Lost funciton at this iteration 0.2984878718852997  and gradient norm 0.
↪0001676468673394993
 57%|    | 57/100 [03:19<02:30,  3.50s/it]
```

[6]:
```
print("Lost funciton at this iteration {}  and gradient norm {}".format(Loss(x),np.
↪linalg.norm(grad)));
```

```
Lost funciton at this iteration 0.29834863543510437  and gradient norm 9.
↪871969814412296e-05
```

2.4.3 Quasi-Newton Method

[8]:
```
#We import all the library we are gona need
import tensorflow as tf
import numpy as np
import pandas as pd
%matplotlib notebook
import matplotlib.pyplot as plt
from numsa.TFHessian import *
import dsdl
```

[11]:
```
ds = dsdl.load("a1a")
X, Y = ds.get_train()
print(X.shape, Y.shape)
```

```
(1605, 119) (1605,)
```

[9]:
```
#Setting the parameter of this run, we will use optimization nomeclature not ML one.
itmax = 20; # Number of epoch.
tol = 1e-4
step_size = 0.2; #Learning rate
Err = [];
Hs = [];
Rsigmas50 = [];
Rsigmas80 = [];
"""
#Old Lost Function, intesead using Stefano's.
#Defining the Loss Function
def Loss(x):
    S = tf.Variable(0.0);
    for j in range(X.shape[0]):
        a = tf.constant((X[j,:].todense().reshape(119,1)),dtype=np.float32);
        b = tf.constant(Y[j],dtype=np.float32)
        a = tf.reshape(a,(119,1));
        x = tf.reshape(x,(119,1));
        dot = tf.matmul(tf.transpose(a),x);
        S = S+tf.math.log(1+tf.math.exp(-b*dot))
    S = (1/X.shape[0])*S;
    return S;
"""
tfX = tf.sparse.from_dense(np.array(X.todense(), dtype=np.float32))
tfY = tf.convert_to_tensor(np.array(Y, dtype=np.float32).reshape(X.shape[0], 1))


#Defining the Loss Function
def Loss(x):
    lam = 1e-3
    x = tf.reshape(x, (119, 1))
    Z = tf.sparse.sparse_dense_matmul(tfX, x, adjoint_a=False)
    Z = tf.math.multiply(tfY, Z)
    S = tf.reduce_sum(tf.math.log(1 + tf.math.exp(-Z)) / tfX.shape[0])#+lam*tf.norm(x)**2
    return S
#Defining the Hessian class for the above loss function in x0
x = tf.Variable(0.1*np.ones((119,1),dtype=np.float32))
H =  Hessian(Loss,x)
grad = H.grad().numpy();
q = grad;
itmax = 100; # Number of epoch.
for it in tqdm(range(itmax)):
    x = x - tf.constant(step_size,dtype=np.float32)*tf.Variable(q,dtype=np.float32);
    x =  tf.Variable(x)
    Err = Err + [np.linalg.norm(grad)];
    if it%5 == 0:
        print("Lost funciton at this iteration {}  and gradient norm {}".format(Loss(x),
↪np.linalg.norm(grad)));
```

(continues on next page)

(continued from previous page)

```
        Hs = Hs + [H.mat()];
    if np.linalg.norm(grad)<tol:
        break
    H =  Hessian(Loss,x)
    grad = H.grad().numpy();
    U, s, Vt = H.RandMatSVD(50,10)
    if it%5 == 0:
        Rsigmas50 = Rsigmas50 + [s];
    U, s, Vt = H.RandMatSVD(80,10)
    if it%5 == 0:
        Rsigmas80 = Rsigmas80 + [s];
    q = (Vt.transpose()@np.linalg.inv(np.diag(s))@U.transpose())@grad;
```

```
  0%|          | 0/100 [00:00<?, ?it/s]
Lost funciton at this iteration 0.9243690371513367  and gradient norm 1.3872039318084717
  5%|         | 5/100 [00:36<11:00,  6.95s/it]
Lost funciton at this iteration 0.43581220507621765  and gradient norm 0.4351786673069
 10%|         | 10/100 [01:12<10:24,  6.94s/it]
Lost funciton at this iteration 0.33864614367485046  and gradient norm 0.
↪15352341532707214
 15%|        | 15/100 [01:48<09:47,  6.91s/it]
Lost funciton at this iteration 0.3093506991863251  and gradient norm 0.05440719053149223
 20%|        | 20/100 [02:24<09:09,  6.87s/it]
Lost funciton at this iteration 0.3009910583496094  and gradient norm 0.
↪019055956974625587
 25%|       | 25/100 [03:00<08:37,  6.90s/it]
Lost funciton at this iteration 0.2988140881061554  and gradient norm 0.
↪006567216478288174
 30%|       | 30/100 [03:36<08:07,  6.96s/it]
Lost funciton at this iteration 0.29820406436920166  and gradient norm 0.
↪0022333976812660694
 35%|      | 35/100 [04:12<07:30,  6.93s/it]
Lost funciton at this iteration 0.2980078160762787  and gradient norm 0.
↪0007552480674348772
 40%|      | 40/100 [04:48<06:56,  6.95s/it]
Lost funciton at this iteration 0.29793307185173035  and gradient norm 0.
↪0002552660880610347
 45%|     | 45/100 [05:25<06:23,  6.97s/it]
Lost funciton at this iteration 0.29789888858795166  and gradient norm 8.
↪670691022416577e-05
 45%|     | 45/100 [05:27<06:40,  7.28s/it]
```

[10]:
```
np.save("Hs",Hs)
print("Lost funciton at this iteration {}  and gradient norm {}".format(Loss(x),np.
↪linalg.norm(grad)));
```

```
Lost funciton at this iteration 0.29789888858795166  and gradient norm 8.
↪670691022416577e-05
```

We would like to make a case for using the random SVD alghorithm in second order optimization problem. Let us consider the Hessian produced during in the in pevious example and show that given their quick decay the randomised SVD is a good approximation.

[11]:
```
Hs = np.load("Hs.npy")
plt.figure()
for H in Hs:
    _, s,_ = np.linalg.svd(H)
    plt.semilogy(s)
plt.legend([r"$H_{"+str(5*i)+"}$" for i in range(len(Hs))],loc=3)
plt.title("Singular Value Hessian Regressian For Adult Problem")
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[11]:
```
Text(0.5, 1.0, 'Singular Value Hessian Regressian For Adult Problem')
```

[12]:
```
plt.figure()
_, s,_ = np.linalg.svd(Hs[4])
plt.semilogy(s)
plt.semilogy(Rsigmas50[4],"--")
plt.semilogy(Rsigmas80[4],"--")
plt.legend([r"$H_{4}$",r"$\overset{\sim}{H}_4^{50}$",r"$\overset{\sim}{H}_4^{80}$"])
plt.title("Singular Value Hessian Regressian For Adult Problem")
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[12]:
```
Text(0.5, 1.0, 'Singular Value Hessian Regressian For Adult Problem')
```

2.4.4 MPI

We show below how to use the Quasi Newton method on multiple cores.

[1]:
```
from ipyparallel import Client
c = Client()
c.ids
```

[1]:
```
[0, 1]
```

[2]:
```
%%px
import tensorflow as tf
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from numsa.TFHessian import *
```

(continues on next page)

(continued from previous page)

```
from mpi4py import MPI
import dsdl
```

```
%px:   0%|          | 0/2 [00:00<?, ?tasks/s]
```

[3]:
```
%%px
itmax = 100; # Number of epoch.
tol = 1e-4
step_size = 0.4; #Learning rate
lam = 1e-3;
```

[4]:
```
%%px
comm = MPI.COMM_WORLD

ds = dsdl.load("a1a")

X, Y = ds.get_train()
indx = np.array_split(range(X.shape[0]),int(comm.Get_size()));
tfX = []
tfY = []
tfXs = tf.sparse.from_dense(np.array(X.todense(), dtype=np.float32))
tfYs = tf.convert_to_tensor(np.array(Y, dtype=np.float32).reshape(X.shape[0], 1))
for k in range(len(indx)):
    tfX = tfX + [tf.sparse.from_dense(np.array(X[indx[comm.Get_rank()]].todense(),␣
↪dtype=np.float32))]
    tfY = tfY + [tf.convert_to_tensor(np.array(Y[indx[comm.Get_rank()]], dtype=np.
↪float32).reshape(X[indx[comm.Get_rank()]].shape[0], 1))]
#Defining the Loss Function
def LossSerial(x):
    x = tf.reshape(x, (119, 1))
    Z = tf.sparse.sparse_dense_matmul(tfXs, x, adjoint_a=False)
    Z = tf.math.multiply(tfYs, Z)
    S = tf.reduce_sum(tf.math.log(1 + tf.math.exp(-Z)) / tfXs.shape[0])+lam*tf.norm(x)**2
    return S
def Loss(x,comm):
    x = tf.reshape(x, (119, 1))
    Z = tf.sparse.sparse_dense_matmul(tfX[comm.Get_rank()], x, adjoint_a=False)
    Z = tf.math.multiply(tfY[comm.Get_rank()], Z)
    S = tf.reduce_sum(tf.math.log(1 + tf.math.exp(-Z)) / tfX[comm.Get_rank()].
↪shape[0])+lam*tf.norm(x)**2
    return S
```

[ ]:
```
%%px
x = tf.Variable(0.1*np.ones((119,1),dtype=np.float32))
H =  Hessian(Loss,x)
grad = H.grad().numpy();
q = grad
#GRADIENT DESCENT
for it in tqdm(range(itmax)):
    x = x - tf.constant(step_size,dtype=np.float32)*tf.Variable(q,dtype=np.float32);
    x =  tf.Variable(x)
    if it%50 == 0:
```

(continues on next page)

(continued from previous page)

```
        print("[Iteration. {}] Lost funciton at this iteration {}  and gradient norm {}".
↪format(it,LossSerial(x),np.linalg.norm(grad)));
    if np.linalg.norm(grad)<tol:
        break
    H =  Hessian(Loss,x)
    grad = H.grad().numpy();
    q = grad
#NEWTON METHOD
x = H.comm.bcast(x,root=0);
q = H.comm.bcast(q,root=0);
grad = q
itmax = 200
for it in tqdm(range(itmax)):
    x = x - tf.constant(step_size,dtype=np.float32)*tf.Variable(q,dtype=np.float32);
    x =  tf.Variable(x)
    if it%5 == 0:
        print("[Iteration. {}] Lost funciton at this iteration {}  and gradient norm {}".
↪format(it,LossSerial(x),np.linalg.norm(grad)));
    if np.linalg.norm(grad)<tol:
        break
    H =  Hessian(Loss,x)
    grad = H.grad().numpy();
    U, s, Vt = H.RandMatSVD(70,10)
    q = (Vt.transpose()@np.linalg.inv(np.diag(s))@U.transpose())@grad;
```

```
[stderr:0] 100%|| 100/100 [00:00<00:00, 210.59it/s]
 23%|       | 46/200 [03:04<10:18,  4.02s/it]
```

```
[stderr:1] 100%|| 100/100 [00:00<00:00, 208.87it/s]
 23%|       | 46/200 [03:04<10:13,  3.99s/it]
```

```
[stdout:0] [Iteration. 0] Lost funciton at this iteration 0.6866949200630188  and␣
↪gradient norm 1.3883576393127441
[Iteration. 50] Lost funciton at this iteration 0.3794311285018921  and gradient norm 0.
↪03935262933373451
[Iteration. 0] Lost funciton at this iteration 0.3619699776172638  and gradient norm 0.
↪02337537333369255
[Iteration. 5] Lost funciton at this iteration 0.34870174527168274  and gradient norm 0.
↪010573619976639748
[Iteration. 10] Lost funciton at this iteration 0.34821632504463196  and gradient norm 0.
↪004498184192925692
[Iteration. 15] Lost funciton at this iteration 0.34894078969955444  and gradient norm 0.
↪002134274458512664
[Iteration. 20] Lost funciton at this iteration 0.34947240352630615  and gradient norm 0.
↪001075760112144053
[Iteration. 25] Lost funciton at this iteration 0.3497142195701599  and gradient norm 0.
↪0005948865436948836
[Iteration. 30] Lost funciton at this iteration 0.34987202286720276  and gradient norm 0.
↪0003507413202896714
[Iteration. 35] Lost funciton at this iteration 0.3499649465084076  and gradient norm 0.
↪00023213459644466639
[Iteration. 40] Lost funciton at this iteration 0.3499954640865326  and gradient norm 0.
↪0001495754549978301
```

(continues on next page)

(continued from previous page)

```
[Iteration. 45] Lost funciton at this iteration 0.35001274943351746  and gradient norm 0.
↪00010022369679063559
Lost funciton at this iteration 0.34680306911468506  and gradient norm 9.
↪214854799211025e-05
```

```
[stdout:1] [Iteration. 0] Lost funciton at this iteration 0.6857503652572632  and␣
↪gradient norm 1.3894309997558594
[Iteration. 50] Lost funciton at this iteration 0.37399160861968994  and gradient norm 0.
↪04358687251806259
[Iteration. 0] Lost funciton at this iteration 0.3619699776172638  and gradient norm 0.
↪02337537333369255
[Iteration. 5] Lost funciton at this iteration 0.3431945741176605  and gradient norm 0.
↪022315166890621185
[Iteration. 10] Lost funciton at this iteration 0.34593257308006287  and gradient norm 0.
↪009717000648379326
[Iteration. 15] Lost funciton at this iteration 0.3493150472640991  and gradient norm 0.
↪004882718436419964
[Iteration. 20] Lost funciton at this iteration 0.3513091504573822  and gradient norm 0.
↪0026368398685008287
[Iteration. 25] Lost funciton at this iteration 0.35240837931632996  and gradient norm 0.
↪001460326719097793
[Iteration. 30] Lost funciton at this iteration 0.352982759475708  and gradient norm 0.
↪0008101381245069206
[Iteration. 35] Lost funciton at this iteration 0.3532729744911194  and gradient norm 0.
↪0004539780493360013
[Iteration. 40] Lost funciton at this iteration 0.35341376066207886  and gradient norm 0.
↪0002526630705688149
[Iteration. 45] Lost funciton at this iteration 0.3534865379333496  and gradient norm 0.
↪00014314994041342288
```

```
%px:   0%|          | 0/2 [00:00<?, ?tasks/s]
```

2.5 Federated Newton Learn

In this section we present the algorithm named Federated Newton Learning (FEDNL) introduced in [2]. The following table contains information regarding the convergence of different FedNL flavours in particular the error column contains $\mathcal{L}(x_{50}) - \mathcal{L}(x^*)$.

Flavour	tol	itera-tion	Error
Vanilla Newton	1e-4	4	0.0
Rank 1 Compression	1e-4	10	1.4901161193847656e-07
Rank 1 Compression Diagonal Regularisation Identity Initial Hessian	1e-4	>50	0.1078076362609863
Rank 1 Compression Diagonal Regularisation Null Initial Hessian	1e-4	>50	8.705258369445801e-05

[1]:
```
from ipyparallel import Client
c = Client()
c.ids
```

[1]:
```
[0, 1]
```

2.5.1 Vanilla Newton

First we reimplement the vanilla Newton method in the framework of FEDNL.

[8]:
```
%%px
import tensorflow as tf
import numpy as np
import scipy.linalg as la
import pandas as pd
import matplotlib.pyplot as plt
from numsa.TFHessian import *
import dsdl

comm = MPI.COMM_WORLD

ds = dsdl.load("a1a")

X, Y = ds.get_train()
indx = np.array_split(range(X.shape[0]),int(comm.Get_size()));
tfX = []
tfY = []
for k in range(len(indx)):
    tfX = tfX + [tf.sparse.from_dense(np.array(X[indx[comm.Get_rank()]].todense(),␣
↪dtype=np.float32))]
    tfY = tfY + [tf.convert_to_tensor(np.array(Y[indx[comm.Get_rank()]], dtype=np.
↪float32).reshape(X[indx[comm.Get_rank()]].shape[0], 1))]

tfXs = tf.sparse.from_dense(np.array(X.todense(), dtype=np.float32))
tfYs = tf.convert_to_tensor(np.array(Y, dtype=np.float32).reshape(X.shape[0], 1))
#Defining the Loss Function
def LossSerial(x):
    lam = 1e-3; #Regularisation
    x = tf.reshape(x, (119, 1))
    Z = tf.sparse.sparse_dense_matmul(tfXs, x, adjoint_a=False)
    Z = tf.math.multiply(tfYs, Z)
    S = tf.reduce_sum(tf.math.log(1 + tf.math.exp(-Z)) / tfXs.shape[0]) + lam*tf.
↪norm(x)**2

    return S
#Defining the Loss Function
def Loss(x,comm):
    lam = 1e-3; #Regularisation
    x = tf.reshape(x, (119, 1))
    Z = tf.sparse.sparse_dense_matmul(tfX[comm.Get_rank()], x, adjoint_a=False)
    Z = tf.math.multiply(tfY[comm.Get_rank()], Z)
    S = tf.reduce_sum(tf.math.log(1 + tf.math.exp(-Z)) / tfX[comm.Get_rank()].shape[0])␣
↪+ lam*tf.norm(x)**2
```

(continues on next page)

(continued from previous page)

```
    return S
################! Setting Of The Solver!##################
itmax = 50
tol = 1e-4;
step_size=1;
############################################################
x = tf.Variable(0.1*np.ones((119,1),dtype=np.float32))

Res = [];

H = Hessian(Loss,x);
H.shift(x)#,start=0*np.identity(x.numpy().shape[0])) #We initialize the shifter
#We now collect and average the loc Hessians in the master node (rk 0)
Hs = H.comm.gather(H.memH, root=0);
if H.comm.Get_rank()==0:
    Hm = (1/len(Hs))*np.sum(Hs,0);
else:
    Hm = None
print("The master Hessian has been initialised")
for it in tqdm(range(itmax)):
    # Obtaining the compression of the difference between local mat
    # and next local mat.
    U,sigma,Vt,ell = H.shift(x,{"comp":MatSVDCompDiag,"rk":119,"type":"mat"});
    shift = Vt.transpose()@np.diag(sigma)@U.transpose();
    #print("Updating local Hessian")
    H.memH = H.memH+step_size*shift;
    grad = H.grad().numpy();
    #Now we update the master Hessian and perform the Newton method step
    Shifts = H.comm.gather(shift, root=0);
    Grads = H.comm.gather(grad, root=0);
    Ells = H.comm.gather(ell, root=0);
    if H.comm.Get_rank() == 0:
        #print("Computing the avarage of the local shifts and grad ...")
        Shift = (1/len(Shifts))*np.sum(Shifts,0);
        Grad = (1/len(Grads))*np.sum(Grads,0);
        Ell = (1/len(Ells))*np.sum(Ells,0);
        res = np.linalg.norm(Grad);
        Res = Res + [res]
        #print("Computing the master Hessian ...")
        Hm = Hm + step_size*Shift;
        #print("Searching new search direction ...")
        A = Hm; #A = Hm + Ell*np.identity(Hm.shape[0]);
        q = np.linalg.solve(A,Grad);
        #print("Found search dir, ",q);
        if it%25 == 0:
            print("(FedNL) [Iteration. {}] Lost funciton at this iteration {}  and␣
↪gradient norm {}".format(it,LossSerial(x),np.linalg.norm(Grad)));
        x = x - tf.Variable(q,dtype=np.float32);
        x =  tf.Variable(x)
    else:
        res = None
    #Distributing the search direction
```

(continues on next page)

(continued from previous page)

```
    x = H.comm.bcast(x,root=0)
    res = H.comm.bcast(res,root=0)
    if res<tol:
            break
LossStar =  0.33691510558128357;
print("Lost funciton at this iteration {}, gradient norm {} and error {}.".
↪format(LossSerial(x),np.linalg.norm(grad),abs(LossSerial(x)-LossStar)))
```

```
%px:   0%|          | 0/2 [00:00<?, ?tasks/s]
```

```
[stderr:1]   8%|         | 4/50 [00:18<03:30,  4.57s/it]
```

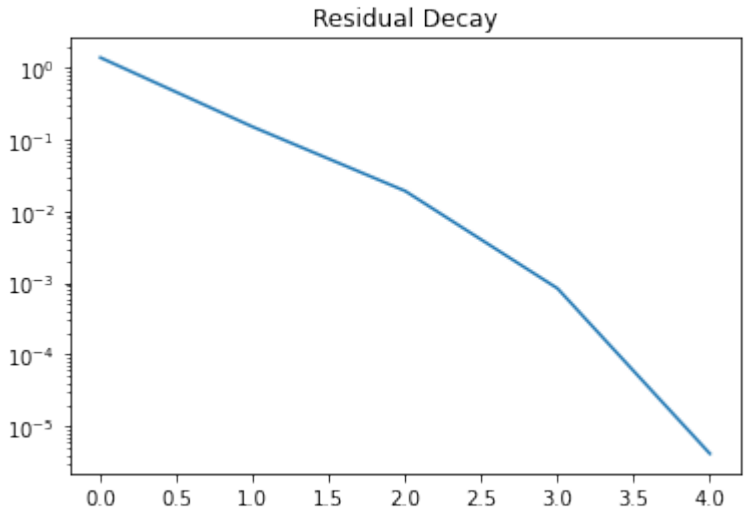
```
[stderr:0]   8%|         | 4/50 [00:18<03:30,  4.57s/it]
```

```
[stdout:1] The master Hessian has been initialised
Lost funciton at this iteration 0.33691510558128357, gradient norm 0.020152254030108452␣
↪and error 0.0.
```

```
[stdout:0] The master Hessian has been initialised
(FedNL) [Iteration. 0] Lost funciton at this iteration 1.2675940990447998  and gradient␣
↪norm 1.388305902481079
Lost funciton at this iteration 0.33691510558128357, gradient norm 0.020149935036897666␣
↪and error 0.0.
```

[10]:
```
import matplotlib.pyplot as plt
Rs = c[:]["Res"][0]
plt.semilogy(range(len(Rs)),Rs)
plt.title("Residual Decay")
```

[10]:
```
Text(0.5, 1.0, 'Residual Decay')
```



2.5.2 FEDNL Rank 1 Compression

[2]:
```
%%px
import tensorflow as tf
import numpy as np
import scipy.linalg as la
import pandas as pd
import matplotlib.pyplot as plt
from numsa.TFHessian import *
import dsdl

comm = MPI.COMM_WORLD

ds = dsdl.load("a1a")

X, Y = ds.get_train()
indx = np.array_split(range(X.shape[0]),int(comm.Get_size()));
tfX = []
tfY = []
for k in range(len(indx)):
    tfX = tfX + [tf.sparse.from_dense(np.array(X[indx[comm.Get_rank()]].todense(),␣
↪dtype=np.float32))]
    tfY = tfY + [tf.convert_to_tensor(np.array(Y[indx[comm.Get_rank()]], dtype=np.
↪float32).reshape(X[indx[comm.Get_rank()]].shape[0], 1))]

tfXs = tf.sparse.from_dense(np.array(X.todense(), dtype=np.float32))
tfYs = tf.convert_to_tensor(np.array(Y, dtype=np.float32).reshape(X.shape[0], 1))
#Defining the Loss Function
def LossSerial(x):
    lam = 1e-3; #Regularisation
    x = tf.reshape(x, (119, 1))
    Z = tf.sparse.sparse_dense_matmul(tfXs, x, adjoint_a=False)
    Z = tf.math.multiply(tfYs, Z)
    S = tf.reduce_sum(tf.math.log(1 + tf.math.exp(-Z)) / tfXs.shape[0]) + lam*tf.
↪norm(x)**2

    return S
#Defining the Loss Function
def Loss(x,comm):
    lam = 1e-3; #Regularisation
    x = tf.reshape(x, (119, 1))
    Z = tf.sparse.sparse_dense_matmul(tfX[comm.Get_rank()], x, adjoint_a=False)
    Z = tf.math.multiply(tfY[comm.Get_rank()], Z)
    S = tf.reduce_sum(tf.math.log(1 + tf.math.exp(-Z)) / tfX[comm.Get_rank()].shape[0])␣
↪+ lam*tf.norm(x)**2
    return S
################! Setting Of The Solver!##################
itmax = 50
tol = 1e-4;
step_size=1;
############################################################
x = tf.Variable(0.1*np.ones((119,1),dtype=np.float32))
```

(continues on next page)

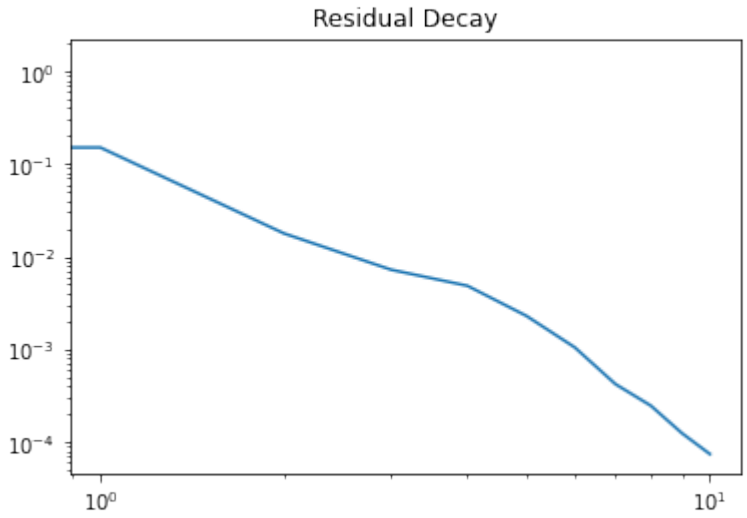
(continued from previous page)

```
Res = [];

H = Hessian(Loss,x);
H.shift(x)#,start=0*np.identity(x.numpy().shape[0])) #We initialize the shifter
#We now collect and average the loc Hessians in the master node (rk 0)
Hs = H.comm.gather(H.memH, root=0);
if H.comm.Get_rank()==0:
    Hm = (1/len(Hs))*np.sum(Hs,0);
else:
    Hm = None
print("The master Hessian has been initialised")
for it in tqdm(range(itmax)):
    # Obtaining the compression of the difference between local mat
    # and next local mat.
    U,sigma,Vt,ell = H.shift(x,{"comp":MatSVDCompDiag,"rk":1,"type":"mat"});
    shift = Vt.transpose()@np.diag(sigma)@U.transpose();
    #print("Updating local Hessian")
    H.memH = H.memH+step_size*shift;
    grad = H.grad().numpy();
    #Now we update the master Hessian and perform the Newton method step
    Shifts = H.comm.gather(shift, root=0);
    Grads = H.comm.gather(grad, root=0);
    Ells = H.comm.gather(ell, root=0);
    if H.comm.Get_rank() == 0:
        #print("Computing the avarage of the local shifts and grad ...")
        Shift = (1/len(Shifts))*np.sum(Shifts,0);
        Grad = (1/len(Grads))*np.sum(Grads,0);
        Ell = (1/len(Ells))*np.sum(Ells,0);
        res = np.linalg.norm(Grad);
        Res = Res + [res];
        #print("Computing the master Hessian ...")
        Hm = Hm + step_size*Shift;
        #print("Searching new search direction ...")
        A = Hm; #A = Hm + Ell*np.identity(Hm.shape[0]);
        q = np.linalg.solve(A,Grad);
        #print("Found search dir, ",q);
        if it%25 == 0:
            print("(FedNL) [Iteration. {}] Lost funciton at this iteration {}  and␣
↪gradient norm {}".format(it,LossSerial(x),np.linalg.norm(Grad)));
        x = x - tf.Variable(q,dtype=np.float32);
        x =  tf.Variable(x)
    else:
        res = None
    #Distributing the search direction
    x = H.comm.bcast(x,root=0)
    res = H.comm.bcast(res,root=0)
    if res<tol:
            break
LossStar =  0.33691510558128357;
print("Lost funciton at this iteration {}, gradient norm {} and error {}.".
↪format(LossSerial(x),np.linalg.norm(grad),abs(LossSerial(x)-LossStar)))
```

```
%px:   0%|          | 0/2 [00:00<?, ?tasks/s]
[stderr:0]  20%|        | 10/50 [00:37<02:28,  3.72s/it]
[stderr:1]  20%|        | 10/50 [00:37<02:28,  3.72s/it]
[stdout:1] The master Hessian has been initialised
Lost funciton at this iteration 0.3369152545928955, gradient norm 0.020147902891039855
↪and error 1.4901161193847656e-07.
[stdout:0] The master Hessian has been initialised
(FedNL) [Iteration. 0] Lost funciton at this iteration 1.2675940990447998  and gradient
↪norm 1.388305902481079
Lost funciton at this iteration 0.3369152545928955, gradient norm 0.020166713744401932
↪and error 1.4901161193847656e-07.
```

[6]:
```
import matplotlib.pyplot as plt
Rs = c[:]["Res"][0]
plt.loglog(range(len(Rs)),Rs)
plt.title("Residual Decay")
```

[6]:
```
Text(0.5, 1.0, 'Residual Decay')
```



2.5.3 FEDNL With Diagonal Regularisation And Different Initial Hessian

[5]:
```
%%px
import tensorflow as tf
import numpy as np
import scipy.linalg as la
import pandas as pd
import matplotlib.pyplot as plt
from numsa.TFHessian import *
```

(continues on next page)

(continued from previous page)

```
import dsdl

comm = MPI.COMM_WORLD

ds = dsdl.load("a1a")

X, Y = ds.get_train()
indx = np.array_split(range(X.shape[0]),int(comm.Get_size()));
tfX = []
tfY = []
for k in range(len(indx)):
    tfX = tfX + [tf.sparse.from_dense(np.array(X[indx[comm.Get_rank()]].todense(),␣
↪dtype=np.float32))]
    tfY = tfY + [tf.convert_to_tensor(np.array(Y[indx[comm.Get_rank()]], dtype=np.
↪float32).reshape(X[indx[comm.Get_rank()]].shape[0], 1))]

tfXs = tf.sparse.from_dense(np.array(X.todense(), dtype=np.float32))
tfYs = tf.convert_to_tensor(np.array(Y, dtype=np.float32).reshape(X.shape[0], 1))
#Defining the Loss Function
def LossSerial(x):
    lam = 1e-3; #Regularisation
    x = tf.reshape(x, (119, 1))
    Z = tf.sparse.sparse_dense_matmul(tfXs, x, adjoint_a=False)
    Z = tf.math.multiply(tfYs, Z)
    S = tf.reduce_sum(tf.math.log(1 + tf.math.exp(-Z)) / tfXs.shape[0]) + lam*tf.
↪norm(x)**2

    return S
#Defining the Loss Function
def Loss(x,comm):
    lam = 1e-3; #Regularisation
    x = tf.reshape(x, (119, 1))
    Z = tf.sparse.sparse_dense_matmul(tfX[comm.Get_rank()], x, adjoint_a=False)
    Z = tf.math.multiply(tfY[comm.Get_rank()], Z)
    S = tf.reduce_sum(tf.math.log(1 + tf.math.exp(-Z)) / tfX[comm.Get_rank()].shape[0])␣
↪+ lam*tf.norm(x)**2
    return S
################! Setting Of The Solver!##################
itmax = 50
tol = 1e-4;
step_size=1;
#############################################################
x = tf.Variable(0.1*np.ones((119,1),dtype=np.float32))

H = Hessian(Loss,x);
H.shift(x,start=np.eye(x.numpy().shape[0])) #We initialize the shifter
#We now collect and average the loc Hessians in the master node (rk 0)
Hs = H.comm.gather(H.memH, root=0);
if H.comm.Get_rank()==0:
    Hm = (1/len(Hs))*np.sum(Hs,0);
else:
    Hm = None
```

(continues on next page)

(continued from previous page)

```
print("The master Hessian has been initialised")
for it in tqdm(range(itmax)):
    # Obtaining the compression of the difference between local mat
    # and next local mat.
    U,sigma,Vt,ell = H.shift(x,{"comp":MatSVDCompDiag,"rk":1,"type":"mat"});
    shift = Vt.transpose()@np.diag(sigma)@U.transpose();
    #print("Updating local Hessian")
    H.memH = H.memH+step_size*shift;
    grad = H.grad().numpy();
    #Now we update the master Hessian and perform the Newton method step
    Shifts = H.comm.gather(shift, root=0);
    Grads = H.comm.gather(grad, root=0);
    Ells = H.comm.gather(ell, root=0);
    if H.comm.Get_rank() == 0:
        #print("Computing the avarage of the local shifts and grad ...")
        Shift = (1/len(Shifts))*np.sum(Shifts,0);
        Grad = (1/len(Grads))*np.sum(Grads,0);
        Ell = (1/len(Ells))*np.sum(Ells,0);
        res = np.linalg.norm(Grad);
        #print("Computing the master Hessian ...")
        Hm = Hm + step_size*Shift;
        #print("Searching new search direction ...")
        A = Hm + Ell*np.identity(Hm.shape[0]);
        q = np.linalg.solve(A,Grad);
        #print("Found search dir, ",q);
        if it%25 == 0:
            print("(FedNL) [Iteration. {}] Lost funciton at this iteration {}  and␣
↪gradient norm {}".format(it,LossSerial(x),np.linalg.norm(Grad)));
        x = x - tf.Variable(q,dtype=np.float32);
        x =  tf.Variable(x)
    else:
        res = None
    #Distributing the search direction
    x = H.comm.bcast(x,root=0)
    res = H.comm.bcast(res,root=0)
    if res<tol:
            break
LossStar =  0.33691510558128357;
print("Lost funciton at this iteration {}, gradient norm {} and error {}.".
↪format(LossSerial(x),np.linalg.norm(grad),abs(LossSerial(x)-LossStar)))
```

```
[stderr:1] 100%|| 50/50 [02:58<00:00,  3.58s/it]
```

```
[stderr:0] 100%|| 50/50 [02:58<00:00,  3.58s/it]
```

```
[stdout:1] The master Hessian has been initialised
Lost funciton at this iteration 0.4447227418422699, gradient norm 0.1290498971939087 and␣
↪error 0.10780763626098633.
```

```
[stdout:0] The master Hessian has been initialised
(FedNL) [Iteration. 0] Lost funciton at this iteration 1.2675940990447998  and gradient␣
↪norm 1.3883059024810079
```

(continues on next page)

(continued from previous page)

```
(FedNL) [Iteration. 25] Lost funciton at this iteration 0.4930209219455719  and gradient␣
↪norm 0.16765998303890228
Lost funciton at this iteration 0.4447227418422699, gradient norm 0.10724713653326035␣
↪and error 0.10780763626098633.


%px:   0%|          | 0/2 [00:00<?, ?tasks/s]
```

2.5.4 FEDNL LowRank + Diagonal

[3]:
```
from ipyparallel import Client
c = Client()
c.ids
```

[3]:
```
[0, 1]
```

[7]:
```
%%px
import tensorflow as tf
import numpy as np
import scipy.linalg as la
import pandas as pd
import matplotlib.pyplot as plt
from numsa.TFHessian import *
import dsdl

comm = MPI.COMM_WORLD

ds = dsdl.load("a1a")

X, Y = ds.get_train()
indx = np.array_split(range(X.shape[0]),int(comm.Get_size()));
tfX = []
tfY = []
for k in range(len(indx)):
    tfX = tfX + [tf.sparse.from_dense(np.array(X[indx[comm.Get_rank()]].todense(),␣
↪dtype=np.float32))]
    tfY = tfY + [tf.convert_to_tensor(np.array(Y[indx[comm.Get_rank()]], dtype=np.
↪float32).reshape(X[indx[comm.Get_rank()]].shape[0], 1))]

tfXs = tf.sparse.from_dense(np.array(X.todense(), dtype=np.float32))
tfYs = tf.convert_to_tensor(np.array(Y, dtype=np.float32).reshape(X.shape[0], 1))
#Defining the Loss Function
def LossSerial(x):
    lam = 1e-3; #Regularisation
    x = tf.reshape(x, (119, 1))
    Z = tf.sparse.sparse_dense_matmul(tfXs, x, adjoint_a=False)
    Z = tf.math.multiply(tfYs, Z)
    S = tf.reduce_sum(tf.math.log(1 + tf.math.exp(-Z)) / tfXs.shape[0]) + lam*tf.
↪norm(x)**2

    return S
```

(continues on next page)

(continued from previous page)

```
#Defining the Loss Function
def Loss(x,comm):
    lam = 1e-3; #Regularisation
    x = tf.reshape(x, (119, 1))
    Z = tf.sparse.sparse_dense_matmul(tfX[comm.Get_rank()], x, adjoint_a=False)
    Z = tf.math.multiply(tfY[comm.Get_rank()], Z)
    S = tf.reduce_sum(tf.math.log(1 + tf.math.exp(-Z)) / tfX[comm.Get_rank()].shape[0])␣
↪+ lam*tf.norm(x)**2
    return S
################! Setting Of The Solver!##################
itmax = 50
tol = 1e-4;
step_size=1;
#############################################################
x = tf.Variable(0.1*np.ones((119,1),dtype=np.float32))

Res = [];
TBCHistory = [];

H = Hessian(Loss,x);
H.loc = True;
#We now collect and average the loc Hessians in the master node (rk 0)

print("Compressing Initial Hessian ")
cU, cS, cVt = MatSVDComp(H.mat()-np.diag(np.diag(H.mat())),30);
print("Assembling the the comp ...")
C =  cU@np.diag(cS)@cVt;
print("Built the compression ...")
A = np.diag(np.diag(H.mat()))+C;
H.shift(x,start=A) #We initialize the shifter
Hs = H.comm.gather(H.memH, root=0);
if H.comm.Get_rank()==0:
    Hm = (1/len(Hs))*np.sum(Hs,0);
else:
    Hm = None
print("The master Hessian has been initialised")
for it in tqdm(range(itmax)):
    # Obtaining the compression of the difference between local mat
    # and next local mat.
    U,sigma,Vt,ell = H.shift(x,{"comp":MatSVDCompDiag,"rk":1,"type":"mat"});
    shift = Vt.transpose()@np.diag(sigma)@U.transpose();
    TBCHistory = TBCHistory + [sigma[0]];
    #print("Updating local Hessian")
    H.memH = H.memH+step_size*shift;
    grad = H.grad().numpy();
    #Now we update the master Hessian and perform the Newton method step
    Shifts = H.comm.gather(shift, root=0);
    Grads = H.comm.gather(grad, root=0);
    Ells = H.comm.gather(ell, root=0);
    if H.comm.Get_rank() == 0:
        #print("Computing the avarage of the local shifts and grad ...")
        Shift = (1/len(Shifts))*np.sum(Shifts,0);
```

(continues on next page)

(continued from previous page)

```
        Grad = (1/len(Grads))*np.sum(Grads,0);
        Ell = (1/len(Ells))*np.sum(Ells,0);
        res = np.linalg.norm(Grad);
        Res = Res + [res];
        #print("Computing the master Hessian ...")
        Hm = Hm + step_size*Shift;
        #print("Searching new search direction ...")
        A = Hm; #A = Hm + Ell*np.identity(Hm.shape[0]);
        q = np.linalg.solve(A,Grad);
        #print("Found search dir, ",q);
        if it%25 == 0:
            print("(FedNL) [Iteration. {}] Lost funciton at this iteration {}  and␣
↪gradient norm {}".format(it,LossSerial(x),np.linalg.norm(Grad)));
        x = x - tf.Variable(q,dtype=np.float32);
        x =  tf.Variable(x)
    else:
        res = None
    #Distributing the search direction
    x = H.comm.bcast(x,root=0)
    res = H.comm.bcast(res,root=0)
    if res<tol:
            break
LossStar =  0.33691510558128357;
print("Lost funciton at this iteration {}, gradient norm {} and error {}.".
↪format(LossSerial(x),np.linalg.norm(grad),abs(LossSerial(x)-LossStar)))
```

```
[stdout:0] Compressing Initial Hessian
Assembling the the comp ...
Built the compression ...
The master Hessian has been initialised
(FedNL) [Iteration. 0] Lost funciton at this iteration 1.2675940990447998  and gradient␣
↪norm 1.388305902481079
Lost funciton at this iteration 0.3369157016277313, gradient norm 0.02016145922243595␣
↪and error 5.960464477539062e-07.
```

```
[stdout:1] Compressing Initial Hessian
Assembling the the comp ...
Built the compression ...
The master Hessian has been initialised
Lost funciton at this iteration 0.3369157016277313, gradient norm 0.020154612138867378␣
↪and error 5.960464477539062e-07.
```

```
%px:   0%|          | 0/2 [00:00<?, ?tasks/s]
```

```
[stderr:0]  46%|     | 23/50 [01:20<01:33,  3.48s/it]
```

```
[stderr:1]  46%|     | 23/50 [01:20<01:33,  3.48s/it]
```

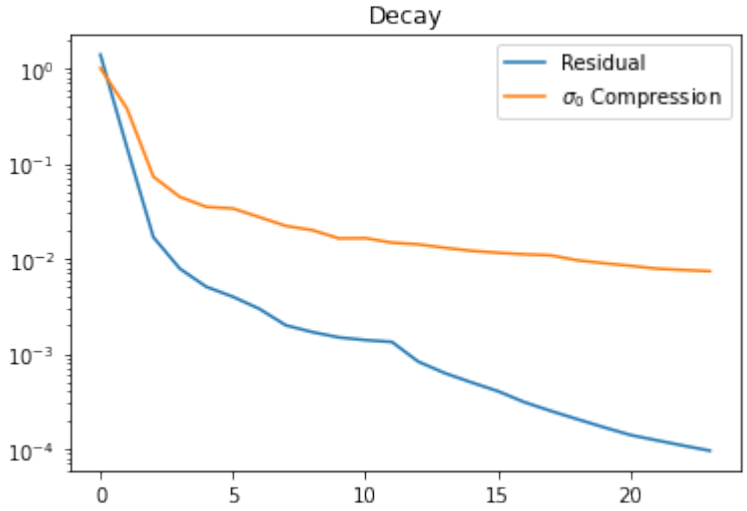
[9]:
```
import matplotlib.pyplot as plt
Rs = c[:]["Res"][0]
```

(continues on next page)

(continued from previous page)

```
plt.semilogy(range(len(Rs)),Rs)
plt.title("Residual Decay")
TBCs = c[:]["TBCHistory"][0]
TBCs[0]=1.
plt.semilogy(range(len(TBCs)),TBCs)
plt.title("Decay")
plt.legend(["Residual",r"$\sigma_0$ Compression"])
```

[9]:
```
<matplotlib.legend.Legend at 0x7fa2b6eb1610>
```



2.6 Physically Informed Neural Network

Here is show how to implement some of the ideas that are presented in [3].

2.6.1 Example - 1D Poisson (DNN3 $\sigma = \tanh$)

[90]:
```
import tensorflow as tf
from tensorflow.keras.constraints import min_max_norm
import logging
tf.get_logger().setLevel(logging.ERROR)
import numpy as np
import matplotlib.pyplot as plt
from numsa.TFHessian import *
from random import *
from tqdm.notebook import tqdm


tf.random.set_seed(7)
############| SETTINGS |############
a = -1; b = 1;
N = 50; #Nuerons in the Hidden Layer
N_EvEl = 101; #Spacing of the evaluation function
```

(continues on next page)

(continued from previous page)

```
itmax = 10000;
step = 1e-3;
tol=1e-32
gamma = tf.Variable(1e0,dtype=np.float32)
def f(x):
    return (np.pi**2)*np.sin(np.pi*x);
####################################

#Pnts = np.linspace(a,b,N_EvEl)


Pnts = np.array([0.0])
Pnts = np.append(Pnts,np.random.uniform(a,b,N_EvEl))
Pnts = np.append(Pnts,1.0)
#Pnts = np.random.uniform(a,b,N_EvEl)

Pnts = np.sort(Pnts)

u = tf.keras.Sequential([
    tf.keras.layers.Dense(N,input_dim=1, activation='tanh'),#Hidden Layer
    tf.keras.layers.Dense(N,activation="tanh"),
    tf.keras.layers.Dense(N,activation="tanh"),
    tf.keras.layers.Dense(1,use_bias=False)#10 leyers weights.
])

#We create an evaluation mesh
mesh = tf.Variable([[point] for point in Pnts]);
F = tf.cast(tf.Variable([[-f(point)] for point in Pnts]),dtype=np.float32);
#Defining the lost function
def ColEnergy(weights):
    with tf.GradientTape() as gtape2:
        with tf.GradientTape() as gtape:
            uh = u(mesh, training=True)
        du_dx = gtape.gradient(uh, [mesh])
    #We use TF Gradient Tape to compute grad uh
    d2u_dx2 = gtape2.gradient(du_dx,[mesh])
    Delta = (d2u_dx2-F)
    Energy = (1/N_EvEl)*tf.reduce_sum(tf.square(Delta));
    #Adding B.C using penalty method idea;
    Energy = Energy + (gamma/N_EvEl)*(tf.square(u(tf.Variable([[a]])))+tf.square(u(tf.
↪Variable([[b]]))))
    return Energy;
```

[91]:
```
print(u(mesh).shape)
print(ColEnergy(u.trainable_weights))
```

```
(103, 1)
tf.Tensor([[40.050835]], shape=(1, 1), dtype=float32)
```

[92]:
```
optimizer = tf.keras.optimizers.Adamax(learning_rate=step)

# Iterate over the batches of a dataset.
```

(continues on next page)

(continued from previous page)

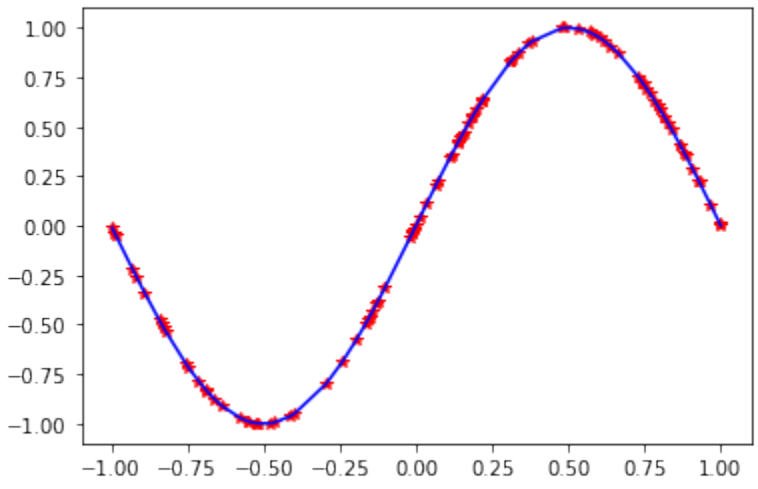
```
for it in tqdm(range(itmax)):
    # Open a GradientTape.
    with tf.GradientTape() as tape:
        Loss = ColEnergy(u.trainable_weights)
    # Get gradients of loss wrt the weights.
    gradients = tape.gradient(Loss, u.trainable_weights)
    res = sum([tf.norm(grad) for grad in gradients]);
    # Update the weights of the model.
    optimizer.apply_gradients(zip(gradients, u.trainable_weights))
    if res <= tol:
        break;
    if it%(itmax/10) == 0:
        print("[It. {}] Loss function value {} and residual {}.".format(it,Loss,res))

print("Loss function value {} and residual {}.".format(Loss,res))
```

```
  0%|          | 0/10000 [00:00<?, ?it/s]
```

```
[It. 0] Loss function value [[40.050835]] and residual 29.165576934814453.
[It. 1000] Loss function value [[0.00762097]] and residual 0.15168429911136627.
[It. 2000] Loss function value [[1.4110279e-05]] and residual 0.11412914842367172.
[It. 3000] Loss function value [[5.9321637e-06]] and residual 0.16223815083503723.
[It. 4000] Loss function value [[3.0792385e-06]] and residual 0.0004416355805005878.
[It. 5000] Loss function value [[2.884495e-06]] and residual 0.0009915133705362678.
[It. 6000] Loss function value [[2.7776434e-06]] and residual 0.008474547415971756.
[It. 7000] Loss function value [[3.0952278e-06]] and residual 0.05875520408153534.
[It. 8000] Loss function value [[1.87274e-05]] and residual 0.35876917839050293.
[It. 9000] Loss function value [[2.5348052e-06]] and residual 0.003511270973831415.
Loss function value [[0.00036402]] and residual 2.1709649562835693.
Loss function value [[0.00036402]] and residual 2.1709649562835693.
```

[93]:
```
plt.plot(mesh.numpy(),u(mesh).numpy(),"r*")
plt.plot(mesh.numpy(),np.sin(np.pi*mesh.numpy()),"b-")
```

[93]:
```
[<matplotlib.lines.Line2D at 0x7fe6286fbb50>]
```



2.6.2 Example - 1D Poisson (DNN3 $\sigma = \tanh$)

[42]:
```
import tensorflow as tf
from tensorflow.keras.constraints import min_max_norm
import logging
tf.get_logger().setLevel(logging.ERROR)
import numpy as np
import matplotlib.pyplot as plt
from numsa.TFHessian import *
from random import *
from tqdm.notebook import tqdm


tf.random.set_seed(7)
############| SETTINGS |############
a = -1; b = 1;
N = 50; #Nuerons in the Hidden Layer
N_EvEl = 501; #Spacing of the evaluation function
itmax = 50000;
step = 1e-3;
tol=1e-32
gamma = tf.Variable(1e0,dtype=np.float32)
def f(x):
    return (np.pi**2)*np.sin(np.pi*x);
####################################

#Pnts = np.linspace(a,b,N_EvEl)


Pnts = np.array([0.0])
Pnts = np.append(Pnts,np.random.uniform(a,b,N_EvEl))
Pnts = np.append(Pnts,1.0)
#Pnts = np.random.uniform(a,b,N_EvEl)

Pnts = np.sort(Pnts)

u = tf.keras.Sequential([
    tf.keras.layers.Dense(N,input_dim=1, activation='tanh'),#Hidden Layer
    tf.keras.layers.Dense(N,activation="tanh"),
    tf.keras.layers.Dense(N,activation="tanh"),
    tf.keras.layers.Dense(1,use_bias=False)
])
#We create an evaluation mesh
mesh = tf.Variable([[point] for point in Pnts]);
F = tf.cast(tf.Variable([[-f(point)] for point in Pnts]),dtype=np.float32);
#Defining the lost function
def ColEnergy(weights):
    with tf.GradientTape() as gtape2:
        with tf.GradientTape() as gtape:
            uh = u(mesh, training=True)
        du_dx = gtape.gradient(uh, [mesh])
    #We use TF Gradient Tape to compute grad uh
    d2u_dx2 = gtape2.gradient(du_dx,[mesh])
```

(continues on next page)

(continued from previous page)

```
    Delta = (d2u_dx2-F)
    Energy = (1/N_EvEl)*tf.reduce_sum(tf.square(Delta));
    #Adding B.C using penalty method idea;
    Energy = Energy + (gamma/N_EvEl)*(tf.square(u(tf.Variable([[a]])))+tf.square(u(tf.
↪Variable([[b]]))))
    return Energy;
optimizer = tf.keras.optimizers.Adamax(learning_rate=step)

def Residual(u,F,mesh):
    with tf.GradientTape() as gtape2:
        with tf.GradientTape() as gtape:
            uh = u(mesh, training=True)
        du_dx = gtape.gradient(uh, [mesh])
    #We use TF Gradient Tape to compute grad uh
    d2u_dx2 = gtape2.gradient(du_dx,[mesh])
    Delta = (d2u_dx2-F)
    return tf.abs(Delta)

# Iterate over the batches of a dataset.
for it in tqdm(range(itmax)):
    # Open a GradientTape.
    with tf.GradientTape() as tape:
        Loss = ColEnergy(u.trainable_weights)
    # Get gradients of loss wrt the weights.
    gradients = tape.gradient(Loss, u.trainable_weights)
    res = sum([tf.norm(grad) for grad in gradients]);
    # Update the weights of the model.
    optimizer.apply_gradients(zip(gradients, u.trainable_weights))
    if res <= tol:
        break;
    if it%(itmax/10) == 0:
        print("[It. {}] Loss function value {} and residual {}.".format(it,Loss,res))

print("Loss function value {} and residual {}.".format(Loss,res))
```

```
  0%|          | 0/50000 [00:00<?, ?it/s]
```

```
[It. 0] Loss function value [[48.291153]] and residual 30.685426712036133.
[It. 5000] Loss function value [[2.1055244e-05]] and residual 0.34498804807662964.
[It. 10000] Loss function value [[2.187425e-05]] and residual 0.4546234905719757.
[It. 15000] Loss function value [[9.026974e-06]] and residual 0.26926928758621216.
[It. 20000] Loss function value [[1.0501252e-06]] and residual 0.027121542021632195.
[It. 25000] Loss function value [[1.2005267e-05]] and residual 0.3423703610897064.
[It. 30000] Loss function value [[1.8846424e-06]] and residual 0.10099022090435028.
[It. 35000] Loss function value [[1.5217504e-05]] and residual 0.37315642833709717.
[It. 40000] Loss function value [[9.6387965e-05]] and residual 0.9605128169059753.
[It. 45000] Loss function value [[0.00021875]] and residual 1.4392529726028442.
Loss function value [[1.8259313e-06]] and residual 0.09424027800559998.
```

[43]:
```
print(np.sort(u.trainable_weights[1]))
```

```
[-0.35823503 -0.33831617 -0.33742264 -0.26369816 -0.25181732 -0.21676466
 -0.15891671 -0.15383002 -0.14794554 -0.1464389  -0.1438641  -0.14049067
```

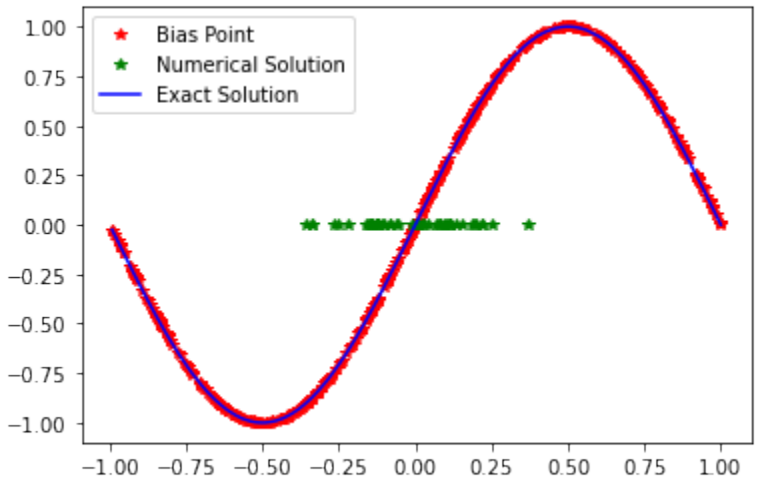
(continues on next page)

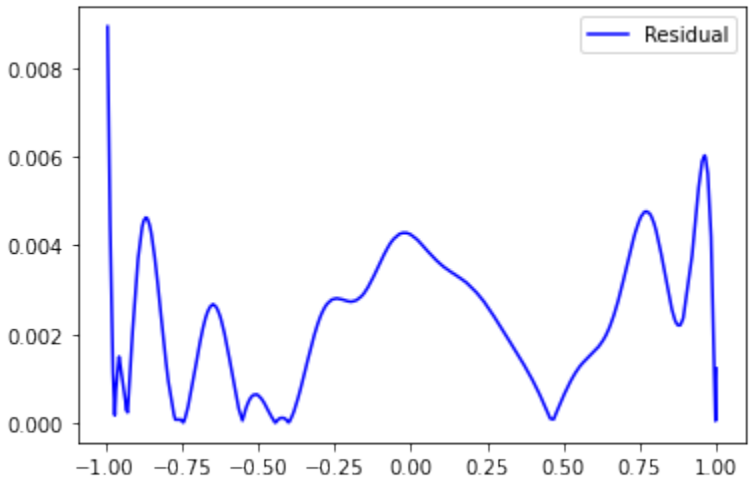
(continued from previous page)

```
-0.1305898  -0.12797573 -0.12470637 -0.1222436  -0.12012127 -0.10821651
-0.10432903 -0.0843064  -0.0820904  -0.06586344 -0.05771144 -0.0130109
 0.00157357  0.00846939  0.00968238  0.01053899  0.02348686  0.02790974
 0.04256134  0.06551021  0.07652889  0.07799305  0.07947817  0.09196392
 0.10043197  0.10841352  0.11284401  0.11348713  0.11466438  0.13370815
 0.15069737  0.18803813  0.1919146   0.20009439  0.21880934  0.22043915
 0.25118658  0.3693546 ]
```

[44]:
```
plt.figure()
plt.plot(mesh.numpy(),u(mesh).numpy(),"r*")
plt.plot(np.sort(u.trainable_weights[1]),np.zeros((len(np.sort(u.trainable_weights[1])),
↪1)),"g*")
plt.plot(mesh.numpy(),np.sin(np.pi*mesh.numpy()),"b-")
plt.legend(["Bias Point","Numerical Solution","Exact Solution"])
plt.figure()
plt.plot(mesh.numpy(),Residual(u,F,mesh)[0].numpy(),"b-")
plt.legend(["Residual"])
```

[44]:
```
<matplotlib.legend.Legend at 0x7f84bc2aaf40>
```





2.6.3 Example - 2D Poisson (DNN1 $\sigma = \tanh$)

[1]:
```
import tensorflow as tf
from tensorflow.keras.constraints import min_max_norm
import logging
tf.get_logger().setLevel(logging.ERROR)
import numpy as np
import matplotlib.pyplot as plt
from numsa.TFHessian import *
from random import *
from tqdm.notebook import tqdm
import itertools


tf.random.set_seed(7)
############| SETTINGS |############
a = 0; b = 0;
c = 1; d= 1;
N = 50; #Nuerons in the Hidden Layer
N_EvEl = 1001; #Spacing of the evaluation function
N_EvEl_BC = 20; #Spacing of the evaluation function
itmax = 50000;
step = 1e-3;
tol=1e-32
data_type="float32"
gamma = tf.Variable(1e0,dtype=np.float32)
def f(x,y):
    return (2*np.pi**2)*np.sin(np.pi*x)*np.sin(np.pi*y);
#####################################

#Pnts = np.linspace(a,b,N_EvEl)


xy_min = [a, b]
xy_max = [c, d]
Pnts = np.random.uniform(low=xy_min, high=xy_max, size=(N_EvEl,2))
BNDPnts = [[P,b] for P in np.random.uniform(a,c,N_EvEl_BC)]
BNDPnts = BNDPnts + [[c,P] for P in np.random.uniform(b,d,N_EvEl_BC)]
BNDPnts = BNDPnts + [[P,d] for P in np.random.uniform(a,c,N_EvEl_BC)]
BNDPnts = BNDPnts + [[a,P] for P in np.random.uniform(b,d,N_EvEl_BC)]
plt.figure();
plt.scatter([P[0] for P in Pnts],[P[1] for P in Pnts]);
plt.scatter([P[0] for P in BNDPnts],[P[1] for P in BNDPnts])
plt.title("Evaluation Points")

u = tf.keras.Sequential([
    tf.keras.layers.Dense(N,input_dim=2, activation='tanh'),#Hidden Layer
    #tf.keras.layers.Dense(N,activation="tanh"),
    #tf.keras.layers.Dense(N,activation="tanh"),
    tf.keras.layers.Dense(1,use_bias=False)
])
#We create an evaluation mesh
mesh = np.array([[point[0],point[1]] for point in Pnts],data_type);
```

(continues on next page)

(continued from previous page)

```
mesh = tf.Variable(mesh)
BNDMesh = np.array([[point[0],point[1]] for point in BNDPnts],data_type);
BNDMesh = tf.Variable(BNDMesh)
F = tf.cast(tf.Variable([[-f(point[0],point[1])] for point in Pnts]),dtype=np.float32);
#Defining the lost function
def ColEnergy(weights):
    v = tf.Variable([[1.0,0.0] for i in range(N_EvEl)])
    with tf.autodiff.ForwardAccumulator(mesh,v) as acc:
        with tf.GradientTape() as tape:
            uh = u(mesh, training=True)
        gradu = tape.gradient(uh,mesh);
    d2u_dx2=acc.jvp(gradu)[:,0]
    v = tf.Variable([[0.0,1.0] for i in range(N_EvEl)])
    with tf.autodiff.ForwardAccumulator(mesh,v) as acc:
        with tf.GradientTape() as tape:
            uh = u(mesh, training=True)
        gradu = tape.gradient(uh,mesh);
    d2u_dy2=acc.jvp(gradu)[:,1]
    Delta = (d2u_dx2+d2u_dy2-tf.reshape(F,(N_EvEl,)))
    Energy = (1/(N_EvEl))*tf.reduce_sum(tf.square(Delta));
    #Adding B.C using penalty method idea;
    Energy = Energy + (gamma/N_EvEl_BC)*tf.reduce_sum((tf.square(u(BNDMesh))))
    return Energy;


LossHistory = [];


optimizer = tf.keras.optimizers.Adamax(learning_rate=step)

# Iterate over the batches of a dataset.
for it in tqdm(range(itmax)):
    # Open a GradientTape.
    with tf.GradientTape() as tape:
        Loss = ColEnergy(u.trainable_weights)
    # Get gradients of loss wrt the weights.
    gradients = tape.gradient(Loss, u.trainable_weights)
    res = sum([tf.norm(grad) for grad in gradients]);
    # Update the weights of the model.
    optimizer.apply_gradients(zip(gradients, u.trainable_weights))
    if res <= tol:
        break;
    if it%(itmax/10) == 0:
        print("[It. {}] Loss function value {} and residual {}.".format(it,Loss,res))
    LossHistory = LossHistory + [Loss];
```

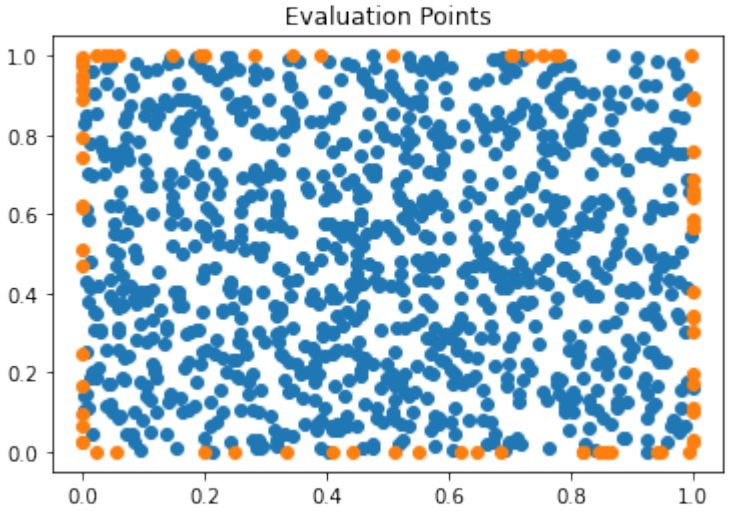
```
  0%|          | 0/50000 [00:00<?, ?it/s]
```

```
[It. 0] Loss function value 100.23746490478516 and residual 13.624043464660645.
[It. 5000] Loss function value 0.003609491745010018 and residual 0.07286364585161209.
[It. 10000] Loss function value 0.0003100232570432126 and residual 0.
↪0010910315904766321.
```

(continues on next page)

(continued from previous page)

```
[It. 15000] Loss function value 0.00019777187844738364 and residual 0.
↪0006636990001425147.
[It. 20000] Loss function value 0.00013671928900294006 and residual 0.
↪0018985089845955372.
[It. 25000] Loss function value 0.00010311126243323088 and residual 0.
↪0003722708497662097.
[It. 30000] Loss function value 8.403413085034117e-05 and residual 0.0022913324646651745.
[It. 35000] Loss function value 7.178398664109409e-05 and residual 0.0007917672046460211.
[It. 40000] Loss function value 6.29963687970303e-05 and residual 0.00032822656794451177.
[It. 45000] Loss function value 5.6171134929172695e-05 and residual 0.
↪00020835785835515708.
```



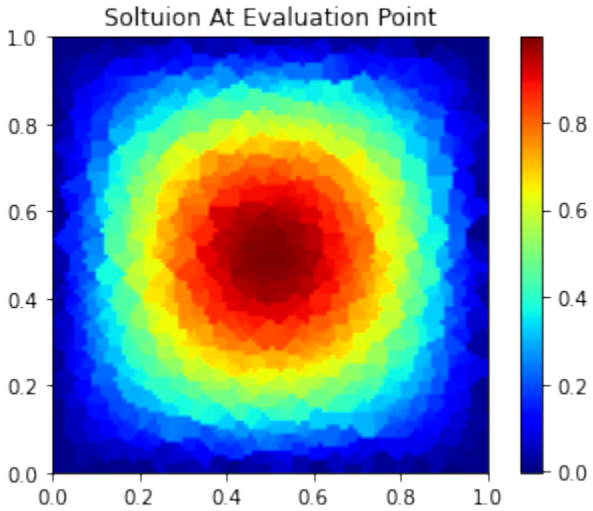
[6]:
```
from scipy.interpolate import griddata
grid_x, grid_y = np.mgrid[0:1:100j, 0:1:100j]
def Residual(u,F,mesh):
    v = tf.Variable([[1.0,0.0] for i in range(N_EvEl)])
    with tf.autodiff.ForwardAccumulator(mesh,v) as acc:
        with tf.GradientTape() as tape:
            uh = u(mesh, training=True)
        gradu = tape.gradient(uh,mesh);
    d2u_dx2=acc.jvp(gradu)[:,0]
    v = tf.Variable([[0.0,1.0] for i in range(N_EvEl)])
    with tf.autodiff.ForwardAccumulator(mesh,v) as acc:
        with tf.GradientTape() as tape:
            uh = u(mesh, training=True)
        gradu = tape.gradient(uh,mesh);
    d2u_dy2=acc.jvp(gradu)[:,1]
    Delta = (d2u_dx2+d2u_dy2-tf.reshape(F,(N_EvEl,)))
    return tf.abs(Delta)


grid_u = griddata(mesh.numpy(), u(mesh).numpy().reshape(N_EvEl,),(grid_x, grid_y),␣
↪method='nearest')
plt.figure()
```

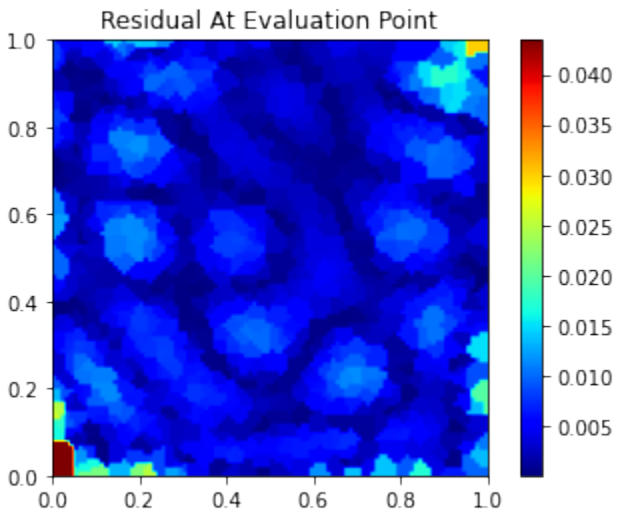
(continues on next page)

(continued from previous page)

```
plt.imshow(grid_u.T, extent=(0,1,0,1), origin='lower',cmap="jet")
plt.colorbar()
plt.title("Solution At Evaluation Point")

grid_res = griddata(mesh.numpy(), Residual(u,F,mesh).numpy().reshape(N_EvEl,),(grid_x,␣
↪grid_y), method='nearest')
plt.figure()
plt.imshow(grid_res.T, extent=(0,1,0,1), origin='lower',cmap="jet")
plt.colorbar()
plt.title("Residual At Evaluation Point")
```

[6]:
```
Text(0.5, 1.0, 'Residual At Evaluation Point')
```



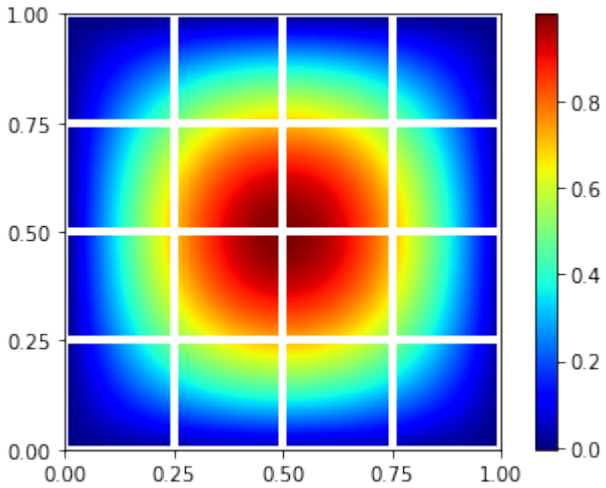


[7]:
```
x = np.arange(0.0, 1.0, 0.01)
y = np.arange(0.0, 1.0, 0.01)
X, Y = np.meshgrid(x, y)
Z = 1*X;
for i in range(len(X)):
```

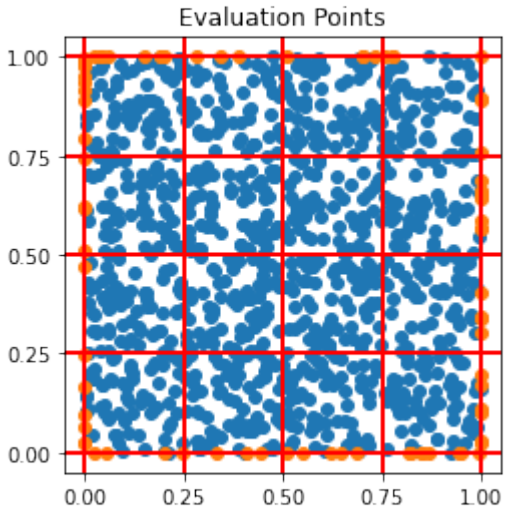
(continues on next page)

(continued from previous page)

```
    for j in range(len(Y)):
        Z[i,j] = u(np.array([[X[i,j],Y[i,j]]]));
import matplotlib.ticker as plticker
fig,ax = plt.subplots()
loc = plticker.MultipleLocator(base=0.25)
ax.xaxis.set_major_locator(loc)
ax.yaxis.set_major_locator(loc)
plt.imshow(Z, cmap='jet',extent=(0,1,0,1))
plt.colorbar()
plt.grid(color='white', linestyle='-', linewidth=4)
```



[8]:
```
import matplotlib.ticker as plticker
fig,ax = plt.subplots()
ax.scatter([P[0] for P in Pnts],[P[1] for P in Pnts]);
ax.scatter([P[0] for P in BNDPnts],[P[1] for P in BNDPnts])
ax.set_title("Evaluation Points")
ax.set_aspect('equal', 'box')
loc = plticker.MultipleLocator(base=0.25)
ax.xaxis.set_major_locator(loc)
ax.yaxis.set_major_locator(loc)
plt.grid(color='red', which='major',linestyle='-', linewidth=2)
```



[9]:
```
print("Number of NN parameters: ",util_shape_product([l.shape  for l in u.trainable_
↪weights]))
print([l.shape for l in u.trainable_weights])
```

```
Number of NN parameters:  200
[TensorShape([2, 50]), TensorShape([50]), TensorShape([50, 1])]
```

2.6.4 Example - 2D Poisson (DNN1 Sigmoid)

[10]:
```
import tensorflow as tf
from tensorflow.keras.constraints import min_max_norm
import logging
tf.get_logger().setLevel(logging.ERROR)
import numpy as np
import matplotlib.pyplot as plt
from numsa.TFHessian import *
from random import *
from tqdm.notebook import tqdm
import itertools


tf.random.set_seed(7)
############| SETTINGS |############
a = 0; b = 0;
c = 1; d= 1;
N = 50; #Nuerons in the Hidden Layer
N_EvEl = 1001; #Spacing of the evaluation function
N_EvEl_BC = 20; #Spacing of the evaluation function
itmax = 50000;
step = 1e-3;
tol=1e-32
data_type="float32"
gamma = tf.Variable(1e0,dtype=np.float32)
def f(x,y):
```

(continues on next page)

(continued from previous page)

```
    return (2*np.pi**2)*np.sin(np.pi*x)*np.sin(np.pi*y);
###################################

#Pnts = np.linspace(a,b,N_EvEl)


xy_min = [a, b]
xy_max = [c, d]
Pnts = np.random.uniform(low=xy_min, high=xy_max, size=(N_EvEl,2))
BNDPnts = [[P,b] for P in np.random.uniform(a,c,N_EvEl_BC)]
BNDPnts = BNDPnts + [[c,P] for P in np.random.uniform(b,d,N_EvEl_BC)]
BNDPnts = BNDPnts + [[P,d] for P in np.random.uniform(a,c,N_EvEl_BC)]
BNDPnts = BNDPnts + [[a,P] for P in np.random.uniform(b,d,N_EvEl_BC)]
plt.figure();
plt.scatter([P[0] for P in Pnts],[P[1] for P in Pnts]);
plt.scatter([P[0] for P in BNDPnts],[P[1] for P in BNDPnts])
plt.title("Evaluation Points")

u = tf.keras.Sequential([
    tf.keras.layers.Dense(N,input_dim=2, activation='sigmoid'),
    tf.keras.layers.Dense(1,use_bias=False)
])
#We create an evaluation mesh
mesh = np.array([[point[0],point[1]] for point in Pnts],data_type);
mesh = tf.Variable(mesh)
BNDMesh = np.array([[point[0],point[1]] for point in BNDPnts],data_type);
BNDMesh = tf.Variable(BNDMesh)
F = tf.cast(tf.Variable([[-f(point[0],point[1])] for point in Pnts]),dtype=np.float32);
#Defining the lost function
def ColEnergy(weights):
    v = tf.Variable([[1.0,0.0] for i in range(N_EvEl)])
    with tf.autodiff.ForwardAccumulator(mesh,v) as acc:
        with tf.GradientTape() as tape:
            uh = u(mesh, training=True)
        gradu = tape.gradient(uh,mesh);
    d2u_dx2=acc.jvp(gradu)[:,0]
    v = tf.Variable([[0.0,1.0] for i in range(N_EvEl)])
    with tf.autodiff.ForwardAccumulator(mesh,v) as acc:
        with tf.GradientTape() as tape:
            uh = u(mesh, training=True)
        gradu = tape.gradient(uh,mesh);
    d2u_dy2=acc.jvp(gradu)[:,1]
    Delta = (d2u_dx2+d2u_dy2-tf.reshape(F,(N_EvEl,)))
    Energy = (1/(N_EvEl))*tf.reduce_sum(tf.square(Delta));
    #Adding B.C using penalty method idea;
    Energy = Energy + (gamma/N_EvEl_BC)*tf.reduce_sum((tf.square(u(BNDMesh))))
    return Energy;


LossHistory = [];
```

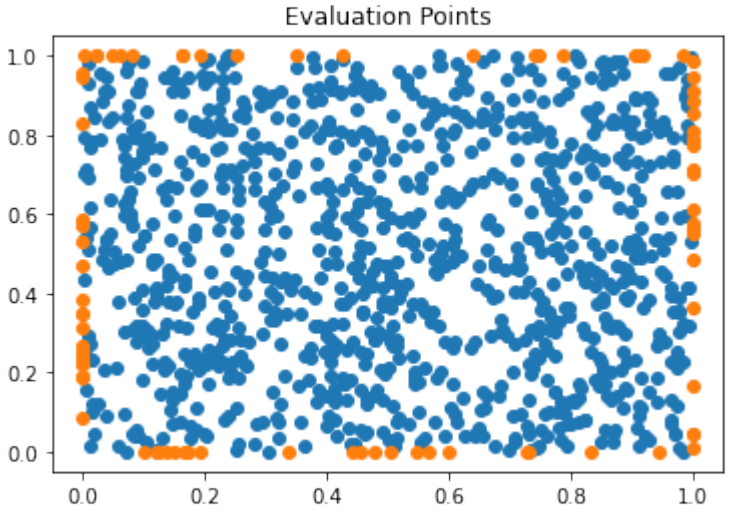
(continues on next page)

(continued from previous page)

```
optimizer = tf.keras.optimizers.Adamax(learning_rate=step)

# Iterate over the batches of a dataset.
for it in tqdm(range(itmax)):
    # Open a GradientTape.
    with tf.GradientTape() as tape:
        Loss = ColEnergy(u.trainable_weights)
    # Get gradients of loss wrt the weights.
    gradients = tape.gradient(Loss, u.trainable_weights)
    res = sum([tf.norm(grad) for grad in gradients]);
    # Update the weights of the model.
    optimizer.apply_gradients(zip(gradients, u.trainable_weights))
    if res <= tol:
        break;
    if it%(itmax/10) == 0:
        print("[It. {}] Loss function value {} and residual {}.".format(it,Loss,res))
    LossHistory = LossHistory + [Loss];
```

```
  0%|          | 0/50000 [00:00<?, ?it/s]
```

```
[It. 0] Loss function value 100.32574462890625 and residual 32.43754959106445.
[It. 5000] Loss function value 0.16919295489788055 and residual 0.14852739870548248.
[It. 10000] Loss function value 0.00048557115951552987 and residual 0.
↪0019718895200639963.
[It. 15000] Loss function value 0.00021293295139912516 and residual 0.007478305138647556.
[It. 20000] Loss function value 0.00013567889982368797 and residual 0.00391441211104393.
[It. 25000] Loss function value 0.00010585739801172167 and residual 0.024814754724502563.
[It. 30000] Loss function value 8.735847222851589e-05 and residual 0.012798226438462734.
[It. 35000] Loss function value 7.515062316088006e-05 and residual 0.005434154532849789.
[It. 40000] Loss function value 6.622089131269604e-05 and residual 0.0051046437583863735.
[It. 45000] Loss function value 5.938777030678466e-05 and residual 0.
↪00028884882340207696.
```



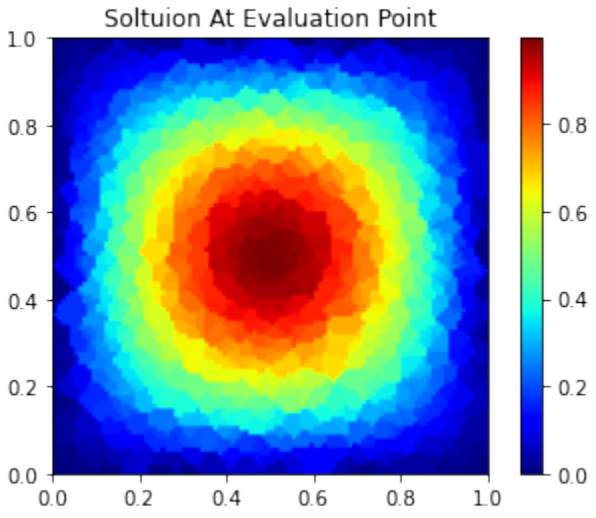
[11]:
```
from scipy.interpolate import griddata
grid_x, grid_y = np.mgrid[0:1:100j, 0:1:100j]
```

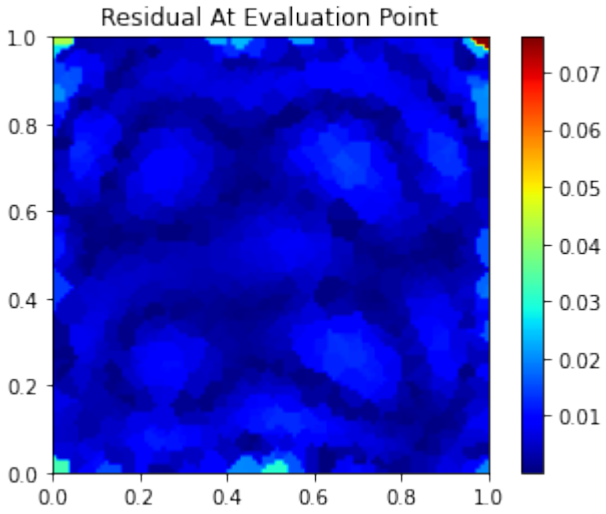
(continues on next page)

(continued from previous page)

```
def Residual(u,F,mesh):
    v = tf.Variable([[1.0,0.0] for i in range(N_EvEl)])
    with tf.autodiff.ForwardAccumulator(mesh,v) as acc:
        with tf.GradientTape() as tape:
            uh = u(mesh, training=True)
        gradu = tape.gradient(uh,mesh);
    d2u_dx2=acc.jvp(gradu)[:,0]
    v = tf.Variable([[0.0,1.0] for i in range(N_EvEl)])
    with tf.autodiff.ForwardAccumulator(mesh,v) as acc:
        with tf.GradientTape() as tape:
            uh = u(mesh, training=True)
        gradu = tape.gradient(uh,mesh);
    d2u_dy2=acc.jvp(gradu)[:,1]
    Delta = (d2u_dx2+d2u_dy2-tf.reshape(F,(N_EvEl,)))
    return tf.abs(Delta)


grid_u = griddata(mesh.numpy(), u(mesh).numpy().reshape(N_EvEl,),(grid_x, grid_y),␣
↪method='nearest')
plt.figure()
plt.imshow(grid_u.T, extent=(0,1,0,1), origin='lower',cmap="jet")
plt.colorbar()
plt.title("Soltuion At Evaluation Point")

grid_res = griddata(mesh.numpy(), Residual(u,F,mesh).numpy().reshape(N_EvEl,),(grid_x,␣
↪grid_y), method='nearest')
plt.figure()
plt.imshow(grid_res.T, extent=(0,1,0,1), origin='lower',cmap="jet")
plt.colorbar()
plt.title("Residual At Evaluation Point")
```

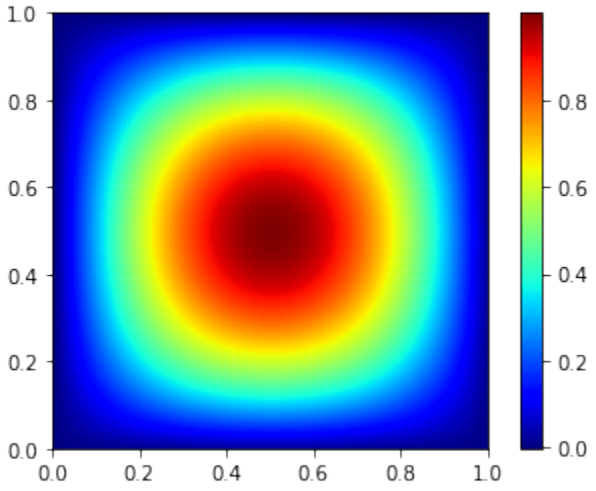
[11]:
```
Text(0.5, 1.0, 'Residual At Evaluation Point')
```





[16]:
```
x = np.arange(0.0, 1.0, 0.01)
y = np.arange(0.0, 1.0, 0.01)
X, Y = np.meshgrid(x, y)
Z = 1*X;
for i in range(len(X)):
    for j in range(len(Y)):
        Z[i,j] = u(np.array([[X[i,j],Y[i,j]]]));
import matplotlib.ticker as plticker
fig,ax = plt.subplots()
plt.imshow(Z, cmap='jet',extent=(0,1,0,1))
plt.colorbar()
```

[16]:
```
<matplotlib.colorbar.Colorbar at 0x7f1a1ad8f370>
```

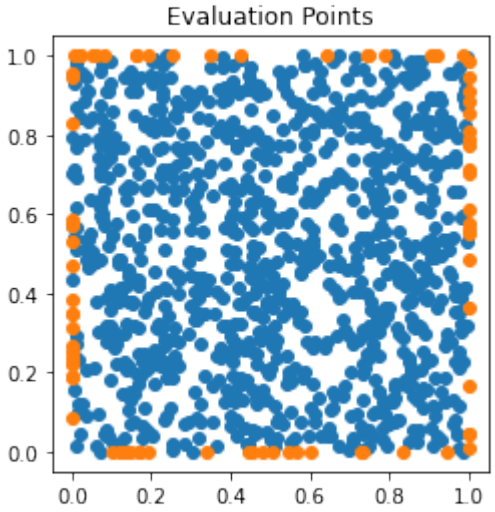


[15]:
```
import matplotlib.ticker as plticker
fig,ax = plt.subplots()
ax.scatter([P[0] for P in Pnts],[P[1] for P in Pnts]);
ax.scatter([P[0] for P in BNDPnts],[P[1] for P in BNDPnts])
ax.set_title("Evaluation Points")
```

(continues on next page)

(continued from previous page)

```
ax.set_aspect('equal', 'box')
```



[14]:
```
print("Number of NN parameters: ",util_shape_product([l.shape  for l in u.trainable_
↪weights]))
print([l.shape for l in u.trainable_weights])
```

```
Number of NN parameters:  200
[TensorShape([2, 50]), TensorShape([50]), TensorShape([50, 1])]
```

[ ]:

CHAPTER

THREE

PARTIAL DIFFERENTIAL EQUATIONS (PDE)

3.1 Eigenvalue Problem (NGS)

In this notebook we study how to solve an eigenvalue problem using NGSolve and SLEPC, in parallel.

Note *to run the code in parallel using a notebook you first need to initialize ipcluster, using the command:* ipcluster start –engines=MPI -n 4

[1]:
```
from ipyparallel import Client
c = Client()
c.ids
```

[1]:
```
[0, 1]
```

[2]:
```
%%px
#STDLIB
from math import pi

#NGSOLVE/NETGEN
from netgen.geom2d import unit_square
from ngsolve import *
from numsa.NGSlepc import *
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.rank
npro = comm.size
H = []
E = []
for k in range(2,4):
    print (rank, npro)
    print("Mesh N. {}".format(k))
    h = 2**(-k);
    E = E + [h];
    if comm.rank == 0:
        ngmesh = unit_square.GenerateMesh(maxh=h).Distribute(comm)
    else:
        ngmesh = netgen.meshing.Mesh.Receive(comm)

    mesh = Mesh(ngmesh)

    fes = H1(mesh, order=1, dirichlet=".*")
```

(continues on next page)

(continued from previous page)

```
    u = fes.TrialFunction()
    v = fes.TestFunction()

    a = BilinearForm(fes)
    a += grad(u)*grad(v)*dx

    m = BilinearForm(fes)
    m += u*v*dx

    a.Assemble()
    m.Assemble()

    EP = SLEPcEigenProblem("GHEP","krylovschur")
    EP.SpectralTransformation("sinvert")

    PC = EP.KSP.getPC();
    PC.setType("lu");
    PC.setFactorSolverType("mumps");

    EP.setOperators([a.mat,m.mat],fes.FreeDofs())
    EP.setWhich(1);

    EP.Solve()

    lam, gfur, gfui = EP.getPairs(fes)
    print([l/pi**2 for l in lam])
    E = E + [abs(lam[0]/pi**2-2)];
```

```
[stdout:1] 1 2
Mesh N. 2
[(2.240498967068547+0j)]
1 2
Mesh N. 3
[(2.0458028065043323+0j)]
```

```
[stdout:0] 0 2
Mesh N. 2
[(2.240498967068547+0j)]
0 2
Mesh N. 3
[(2.0458028065043323+0j)]
```

[3]:
```
%%px
E
```

```
Out[0:13]: [0.25, 0.24049896706854712, 0.125, 0.04580280650433233]
```

```
Out[1:13]: [0.25, 0.24049896706854712, 0.125, 0.04580280650433233]
```

[4]:
```
%%px
#STDLIB
```

(continues on next page)

(continued from previous page)

```
from math import pi

#NGSOLVE/NETGEN
from netgen.geom2d import unit_square
from ngsolve import *
import ngsolve.ngs2petsc as N2P


#MPI
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.rank
npro = comm.size
if rank == 0:
    print (rank, npro)

if comm.rank == 0:
    ngmesh = unit_square.GenerateMesh(maxh=0.05).Distribute(comm)
else:
    ngmesh = netgen.meshing.Mesh.Receive(comm)

mesh = Mesh(ngmesh)

fes = H1(mesh, order=1, dirichlet=".*")
u = fes.TrialFunction()
v = fes.TestFunction()

a = BilinearForm(fes)
a += grad(u)*grad(v)*dx
pre = Preconditioner(a, type="direct", inverse="masterinverse")

m = BilinearForm(fes)
m += u*v*dx

a.Assemble()
m.Assemble()

EP = SLEPcEigenProblem("GHEP","krylovschur")
EP.SpectralTransformation("sinvert")

PC = EP.KSP.getPC();
PC.setType("lu");
PC.setFactorSolverType("mumps");

EP.setOperators([a.mat,m.mat],fes.FreeDofs())
EP.setWhich(5);

EP.Solve()

lam, gfur, gfui = EP.getPairs(fes)
if rank == 0:
    print([l/pi**2 for l in lam])
```

(continues on next page)

(continued from previous page)

```
eig1 = GridFunction(fes); eig1.vec.data = gfur.vecs[0].data;
eig2 = GridFunction(fes); eig2.vec.data = gfur.vecs[1].data;
eig3 = GridFunction(fes); eig3.vec.data = gfur.vecs[2].data;
eig4 = GridFunction(fes); eig4.vec.data = gfur.vecs[3].data;
```

```
[stdout:0] 0 2
[(2.0063458676628505+0j), (5.039046175256581+0j), (5.0397702505874244+0j), (8.
↪102182495028751+0j), (10.152804821204073+0j)]
```

[5]:
```
from ngsolve.webgui import Draw
eig1 = c[:]["eig1"]
eig2 = c[:]["eig2"]
eig3 = c[:]["eig3"]
eig4 = c[:]["eig4"]
Draw (eig1[0])
Draw (eig2[0])
Draw (eig3[0])
Draw (eig4[0])
```

```
WebGuiWidget(value={'ngsolve_version': '6.2.2105-64-g8568b5f22', 'mesh_dim': 2, 'order2d
↪': 2, 'order3d': 2, 'd...
```

```
WebGuiWidget(value={'ngsolve_version': '6.2.2105-64-g8568b5f22', 'mesh_dim': 2, 'order2d
↪': 2, 'order3d': 2, 'd...
```

```
WebGuiWidget(value={'ngsolve_version': '6.2.2105-64-g8568b5f22', 'mesh_dim': 2, 'order2d
↪': 2, 'order3d': 2, 'd...
```

```
WebGuiWidget(value={'ngsolve_version': '6.2.2105-64-g8568b5f22', 'mesh_dim': 2, 'order2d
↪': 2, 'order3d': 2, 'd...
```

[5]:
```
BaseWebGuiScene
```

3.2 Pattern Formation PDE (FD)

[17]:
```
import numpy as np
from math import sqrt
import scipy.integrate
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IPython.display import HTML
from scipy import sparse
from scipy.sparse.linalg import eigs, spsolve
from tqdm.notebook import trange, tqdm
from nodepy.runge_kutta_method import *
```

We are interested in solving numerically the following system of reaction-diffusion PDE:

$$\begin{cases} \partial_t u = \delta_1(\partial_x^2 u + \partial_y^2 u) + \alpha u(1-\tau_1 v^2) + v(1-\tau_2 u) \\ \\ \partial_t v = \delta_2(\partial_x^2 v + \partial_y^2 v) + \beta v(1-\frac{\alpha\tau_1}{\beta}uv) + u(\gamma+\tau_2 u) \end{cases}$$

and we can separate the diffusion terms from the reactions term, obtaining,

$$\begin{cases} \partial_t u = \nabla^2 \cdot u + f(u,v) \\ \\ \partial_t v = \nabla^2 \cdot v + g(u,v) \end{cases}$$

where the functions f and g are clearly defined as:

in particular we focus our attention on solving the above coupled system of autonomous ordinary differential equations reppresenting the reaction part of the Turing model. In particular we will solve these implementing a 4th order Runge-Kutta method.

[2]:
```
class ODESol:
    def __init__(self,timesteps,timestep,U):
        self.t = timesteps;
        self.h = timestep;
        self.y = U;
def RK4(F,T,U0,arg,N):
    tt = np.linspace(T[0],T[1],N);
    h = tt[1]-tt[0];
    U = np.zeros([len(U0),N]);
    U[:,0] = U0;
    for i in trange(0,N-1):
        Y1 = U[:,i];
        Y2 = U[:,i] + 0.5*h*F(tt[i],Y1,arg);
        Y3 = U[:,i] + 0.5*h*F(tt[i]+0.5*h,Y2,arg);
        Y4 = U[:,i] + h*F(tt[i]+0.5*h,Y3,arg);
        U[:,i+1] = U[:,i]+(h/6)*(F(tt[i],Y1,arg)+2*F(tt[i]+0.5*h,Y2,arg)+2*F(tt[i]+0.5*h,
↪Y3,arg)+F(tt[i]+h,Y4,arg))
    sol = ODESol(tt,h,U);
    return sol;
```

3.2.1 Reaction

[3]:
```
Table = []
Table = Table + [{"delta1":0.00225,"delta2":0.0045,"tau1":0.02,"tau2":0.2,"alpha":0.899,
↪"beta":-0.91,"gamma":-0.899}]
Table = Table + [{"delta1":0.001,"delta2":0.0045,"tau1":0.02,"tau2":0.2,"alpha":0.899,
↪"beta":-0.91,"gamma":-0.899}]
Table = Table + [{"delta1":0.00225,"delta2":0.0045,"tau1":0.02,"tau2":0.2,"alpha":1.9,
↪"beta":-0.91,"gamma":-1.9}]

Table = Table + [{"delta1":0.00225,"delta2":0.0045,"tau1":2.02,"tau2":0.0,"alpha":2.0,
↪"beta":-0.91,"gamma":-2}]
Table = Table + [{"delta1":0.00105,"delta2":0.0021,"tau1":3.5,"tau2":0.0,"alpha":0.899,
↪"beta":-0.91,"gamma":-0.899}]
Table = Table + [{"delta1":0.00225,"delta2":0.0045,"tau1":0.02,"tau2":0.2,"alpha":1.9,
↪"beta":-0.85,"gamma":-1.9}]
Table = Table + [{"delta1":0.00225,"delta2":0.0005,"tau1":2.02,"tau2":0.0,"alpha":2.0,
↪"beta":-0.91,"gamma":-2}]
```

(continues on next page)

(continued from previous page)

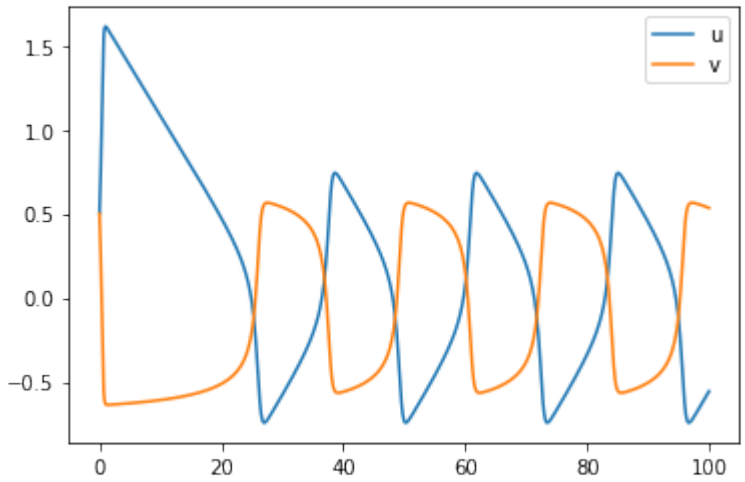
```
def Reaction(t,x,parameters):
    #The ODE is autonomus so we don't really need
    #the depende on time.
    u = x[0]; v = x[1]; #We grab the useful quantity.
    d1 = parameters["delta1"]; d2 = parameters["delta2"];
    t1 = parameters["tau1"]; t2 = parameters["tau2"];
    a = parameters["alpha"]; b = parameters["beta"]; g = parameters["gamma"];
    du = a*u*(1-t1*(v**2))+v*(1-t2*u);
    dv = b*v*(1+(a*t1/b)*(u*v))+u*(g+t2*v);
    return np.array([du,dv])
```

[4]:
```
t0 = 0.                    # Initial time
u0 = np.array([0.5,0.5])# Initial values
tfinal = 100.               # Final time
dt_output=0.1# Interval between output for plotting
N=int(tfinal/dt_output)       # Number of output times
print(N)
tt=np.linspace(t0,tfinal,N)  # Output times
ODE = RK4(Reaction,[t0,tfinal],u0,Table[6],N);
uu=ODE.y
plt.plot(tt,uu[0,:],tt,uu[1,:])
plt.legend(["u","v"])
```

```
1000
```

```
  0%|          | 0/999 [00:00<?, ?it/s]
```

[4]:
```
<matplotlib.legend.Legend at 0x7f32bbb97940>
```



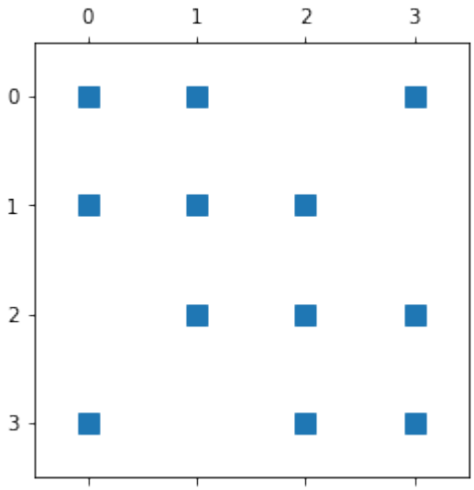
3.2.2 Diffusion

[5]:
```
def laplacian_1D(m):
    em = np.ones(m)
    e1=np.ones(m-1)
    A = (sparse.diags(-2*em,0)+sparse.diags(e1,-1)+sparse.diags(e1,1))/((2/(m+1))**2);
    A = A.tocsr();
    A[0,-1]=1/((2/(m+1))**2);
    A[-1,0]=1/((2/(m+1))**2);
    return A;
```

[6]:
```
plt.spy(laplacian_1D(4))
```

```
/home/uzerbinati/.local/lib/python3.8/site-packages/scipy/sparse/_index.py:82:␣
↪SparseEfficiencyWarning: Changing the sparsity structure of a csr_matrix is expensive.␣
↪lil_matrix is more efficient.
  self._set_intXint(row, col, x.flat[0])
```

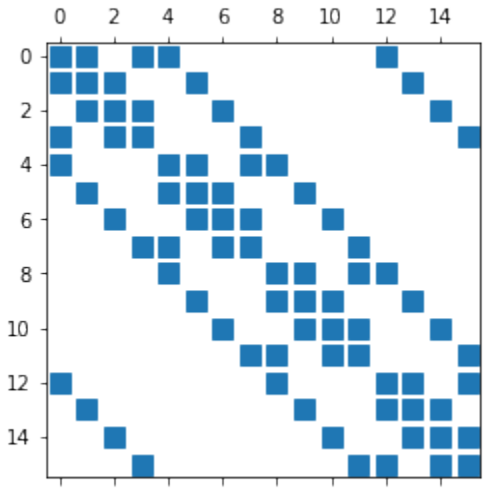
[6]:
```
<matplotlib.lines.Line2D at 0x7f32bb1c5d90>
```



[7]:
```
def laplacian_2D(m):
    I = np.eye(m)
    A = laplacian_1D(m)
    return sparse.kron(A,I) + sparse.kron(I,A)
```

[8]:
```
plt.spy(laplacian_2D(4))
```

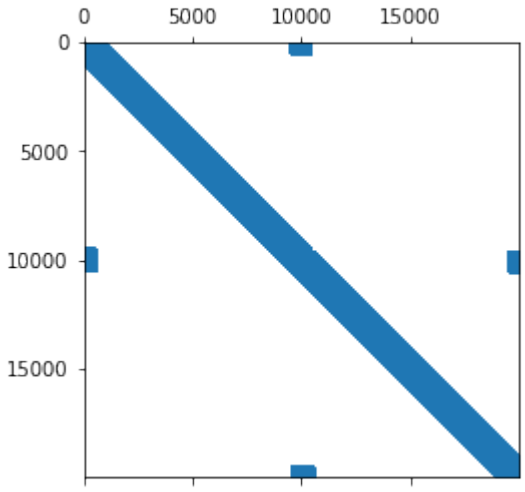
[8]:
```
<matplotlib.lines.Line2D at 0x7f32bb1ad6d0>
```



[9]:
```
m=100
x=np.linspace(-1,1,m+2); x=x[1:-1]
y=np.linspace(-1,1,m+2); y=y[1:-1]
print(x[1]-x[0])
X,Y=np.meshgrid(x,y)
A=laplacian_2D(m)
plt.spy(sparse.bmat([[2*A,None],[None,1*A]]))
```

```
0.01980198019801982
```

[9]:
```
<matplotlib.lines.Line2D at 0x7f32bb12d5b0>
```



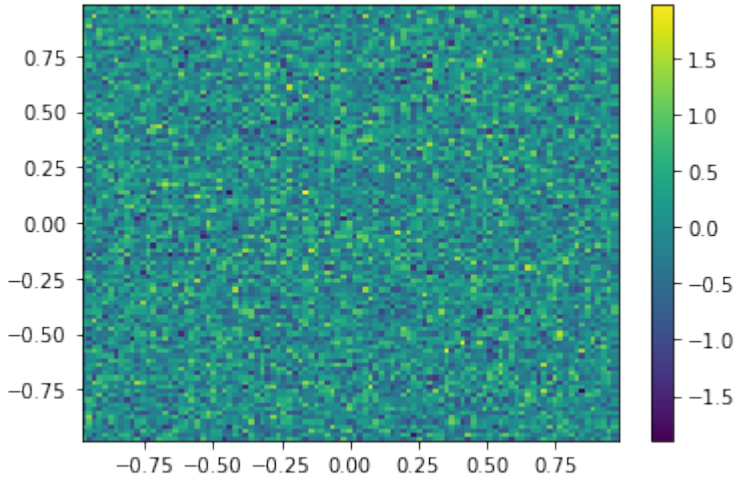
[10]:
```
def Diffusion(t,x,parameters):
    B = sparse.bmat([[parameters["delta1"]*A,None],[None,parameters["delta2"]*A]]);
    return B*x;
#Generating the initial data
mu, sigma = 0, 0.5 # mean and standard deviation
u0 = np.random.normal(mu, sigma,m**2);
v0 = np.random.normal(mu, sigma,m**2)
#Plotting the initial data
```

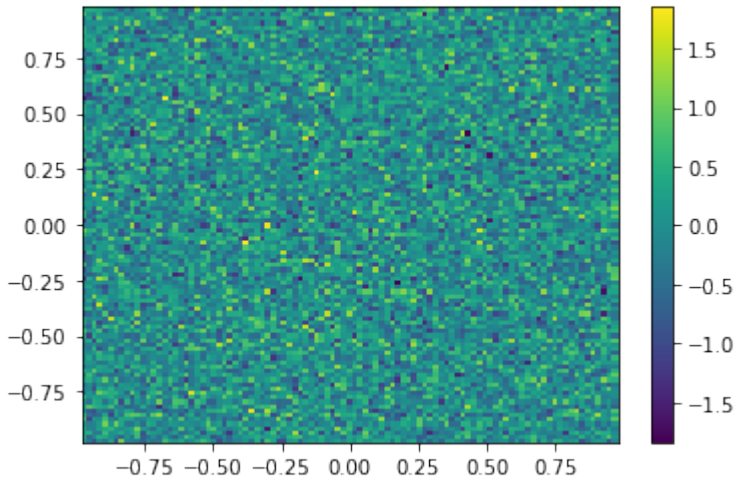
(continues on next page)

(continued from previous page)

```
plt.figure()
U0=u0.reshape([m,m])
plt.pcolor(X,Y,U0)
plt.colorbar();
plt.figure()
V0=v0.reshape([m,m])
plt.pcolor(X,Y,V0)
plt.colorbar();
```

```
<ipython-input-10-5d9199957f96>:11: MatplotlibDeprecationWarning: shading='flat' when X␣
↪and Y have the same dimensions as C is deprecated since 3.3.  Either specify the␣
↪corners of the quadrilaterals with X and Y, or pass shading='auto', 'nearest' or
↪'gouraud', or set rcParams['pcolor.shading'].  This will become an error two minor␣
↪releases later.
  plt.pcolor(X,Y,U0)
<ipython-input-10-5d9199957f96>:15: MatplotlibDeprecationWarning: shading='flat' when X␣
↪and Y have the same dimensions as C is deprecated since 3.3.  Either specify the␣
↪corners of the quadrilaterals with X and Y, or pass shading='auto', 'nearest' or
↪'gouraud', or set rcParams['pcolor.shading'].  This will become an error two minor␣
↪releases later.
  plt.pcolor(X,Y,V0)
```





[11]:
```
TIndex = 0;
t0 = 0.0                    # Initial time
tfinal = 0.1              # Final time
dt_output=0.021           # Interval between output for plotting
N=int(tfinal/dt_output)       # Number of output times
print("CFL suggested time step",(2.5*(0.0198)**2)/(8*max(Table[TIndex]["delta1"],
↪Table[TIndex]["delta2"])));
#ODE = scipy.integrate.solve_ivp(Diffusion,[t0,tfinal],np.append(u0,v0,axis=0),
↪args=[Table[TIndex]],method='RK45',t_eval=tt,atol=1.e-10,rtol=1.e-10);
ODE = RK4(Diffusion,[t0,tfinal],np.append(u0,v0,axis=0),Table[TIndex],N);
uu=ODE.y
print("Step size of the Method",ODE.h);
print(ODE.h*eigs(sparse.bmat([[Table[TIndex]["delta1"]*A,None],[None,Table[TIndex][
↪"delta2"]*A]]),k=10)[0])
plt.figure()
RK44 = loadRKM('RK44')
RK44.plot_stability_region(bounds=[-5,1,-5,5])
plt.plot(ODE.h*eigs(sparse.bmat([[Table[TIndex]["delta1"]*A,None],[None,Table[TIndex][
↪"delta2"]*A]]),k=10)[0].real,ODE.h*eigs(sparse.bmat([[Table[TIndex]["delta1"]*A,None],
↪[None,Table[TIndex]["delta2"]*A]]),k=10)[0].imag,"b*")
```

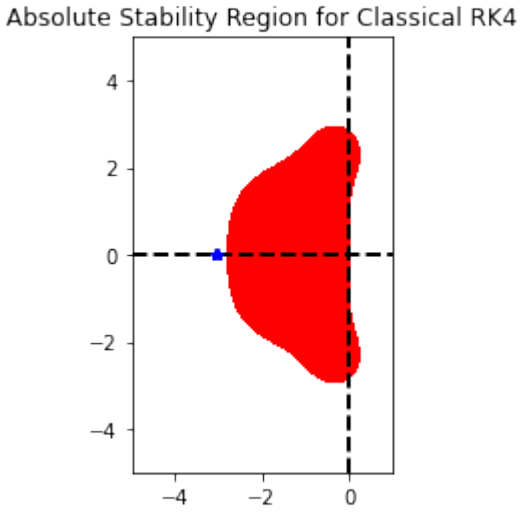
```
CFL suggested time step 0.02722500000000001
```

```
  0%|          | 0/3 [00:00<?, ?it/s]
```

```
Step size of the Method 0.03333333333333333
[-3.0603    +0.j -3.0587903 +0.j -3.0587903 +0.j -3.0572806 +0.j
 -3.0572806 +0.j -3.0587903 +0.j -3.0587903 +0.j -3.0572806 +0.j
 -3.0572806 +0.j -3.05426716+0.j]
```

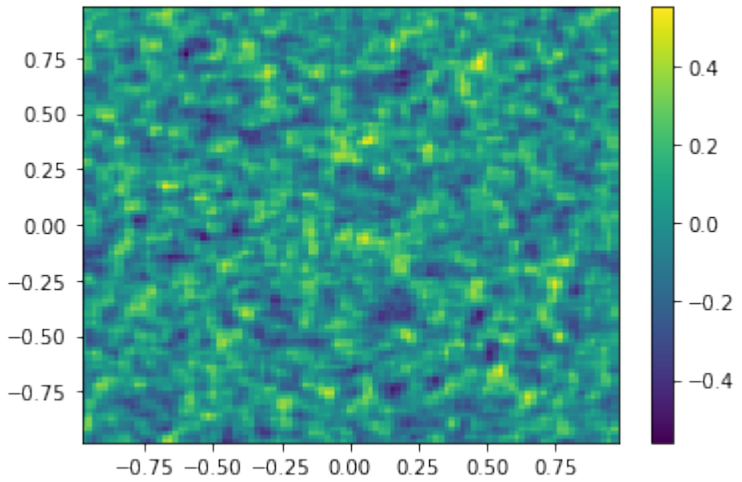
[11]:
```
[<matplotlib.lines.Line2D at 0x7f32ba2121f0>]
```

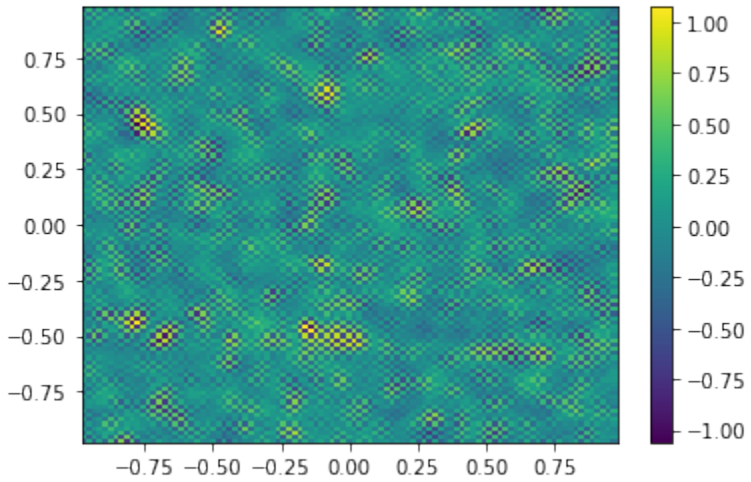
```
<Figure size 432x288 with 0 Axes>
```



[12]:
```
ut = uu[0:m**2,-1];
vt = uu[m**2:,-1];
Ut=ut.reshape([m,m])
plt.pcolor(X,Y,Ut)
plt.colorbar();
plt.figure()
Vt=vt.reshape([m,m])
plt.pcolor(X,Y,Vt)
plt.colorbar();
```

```
<ipython-input-12-14a28ac5572b>:4: MatplotlibDeprecationWarning: shading='flat' when X␣
↪and Y have the same dimensions as C is deprecated since 3.3.  Either specify the␣
↪corners of the quadrilaterals with X and Y, or pass shading='auto', 'nearest' or
↪'gouraud', or set rcParams['pcolor.shading'].  This will become an error two minor␣
↪releases later.
  plt.pcolor(X,Y,Ut)
<ipython-input-12-14a28ac5572b>:8: MatplotlibDeprecationWarning: shading='flat' when X␣
↪and Y have the same dimensions as C is deprecated since 3.3.  Either specify the␣
↪corners of the quadrilaterals with X and Y, or pass shading='auto', 'nearest' or
↪'gouraud', or set rcParams['pcolor.shading'].  This will become an error two minor␣
↪releases later.
  plt.pcolor(X,Y,Vt)
```





3.2.3 Reaction-Diffusion

[13]:
```
def ReactionDiffusion(t,x,parameters):
    u = x[0:m**2]; v = x[m**2:]; #We grab the useful quantity.
    d1 = parameters["delta1"]; d2 = parameters["delta2"];
    t1 = parameters["tau1"]; t2 = parameters["tau2"];
    a = parameters["alpha"]; b = parameters["beta"]; g = parameters["gamma"];
    #Diffusion
    B = sparse.bmat([[parameters["delta1"]*A,None],[None,parameters["delta2"]*A]]);
    #Reaction
    du = a*u*(1-t1*(v**2))+v*(1-t2*u);
    dv = b*v*(1+(a*t1/b)*(u*v))+u*(g+t2*v);
    b = np.append(du,dv,axis=0);
    return B*x+b;
TIndex = 0;
t0 = 0.0                    # Initial time
tfinal = 150               # Final time
dt_output=0.0065           # Interval between output for plotting
```

(continues on next page)

(continued from previous page)

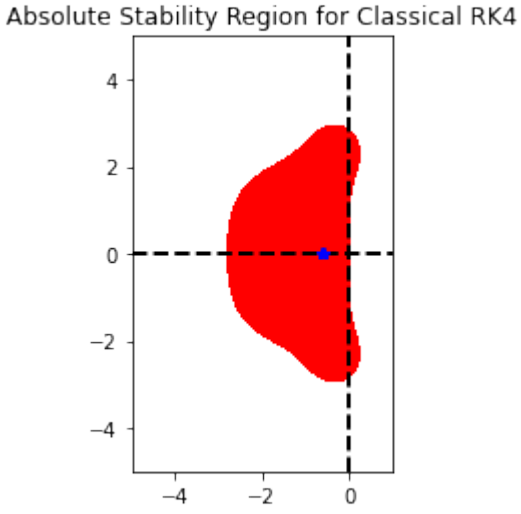
```
N=int(tfinal/dt_output)       # Number of output times
#ODE = scipy.integrate.solve_ivp(Diffusion,[t0,tfinal],np.append(u0,v0,axis=0),
↪args=[Table[TIndex]],method='RK45',t_eval=tt,atol=1.e-10,rtol=1.e-10);
ODE = RK4(ReactionDiffusion,[t0,tfinal],np.append(u0,v0,axis=0),Table[TIndex],N);
uu=ODE.y
print("Step size of the Method",ODE.h);
print(ODE.h*eigs(sparse.bmat([[Table[TIndex]["delta1"]*A,None],[None,Table[TIndex][
↪"delta2"]*A]]),k=10)[0])
plt.figure()
RK44 = loadRKM('RK44')
RK44.plot_stability_region(bounds=[-5,1,-5,5])
plt.plot(ODE.h*eigs(sparse.bmat([[Table[TIndex]["delta1"]*A,None],[None,Table[TIndex][
↪"delta2"]*A]]),k=10)[0].real,ODE.h*eigs(sparse.bmat([[Table[TIndex]["delta1"]*A,None],
↪[None,Table[TIndex]["delta2"]*A]]),k=10)[0].imag,"b*")
```

```
  0%|          | 0/23075 [00:00<?, ?it/s]
```

```
Step size of the Method 0.0065005417118093175
[-0.59680823+0.j -0.59651382+0.j -0.59651382+0.j -0.5962194 +0.j
 -0.5962194 +0.j -0.59651382+0.j -0.59651382+0.j -0.5962194 +0.j
 -0.5962194 +0.j -0.59563173+0.j]
```

[13]:
```
[<matplotlib.lines.Line2D at 0x7f32b96476d0>]
```

```
<Figure size 432x288 with 0 Axes>
```



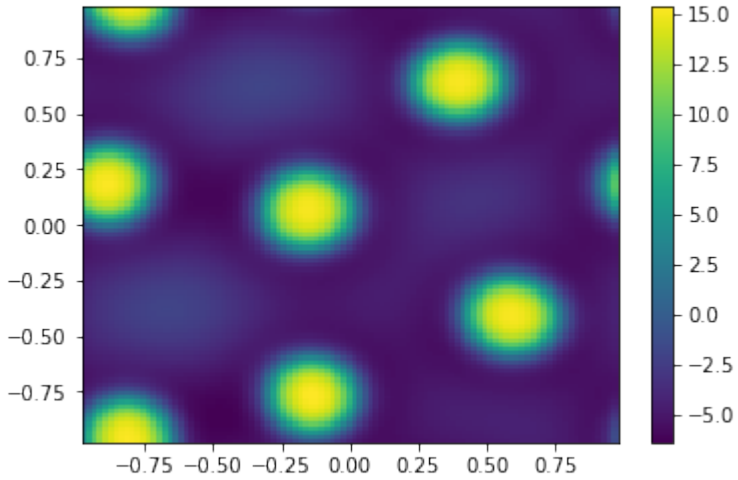
[14]:
```
ut = uu[0:m**2,-1];
vt = uu[m**2:,-1];
Ut=ut.reshape([m,m])
plt.pcolor(X,Y,Ut)
plt.colorbar();
plt.figure()
Vt=vt.reshape([m,m])
plt.pcolor(X,Y,Vt)
plt.colorbar();
```

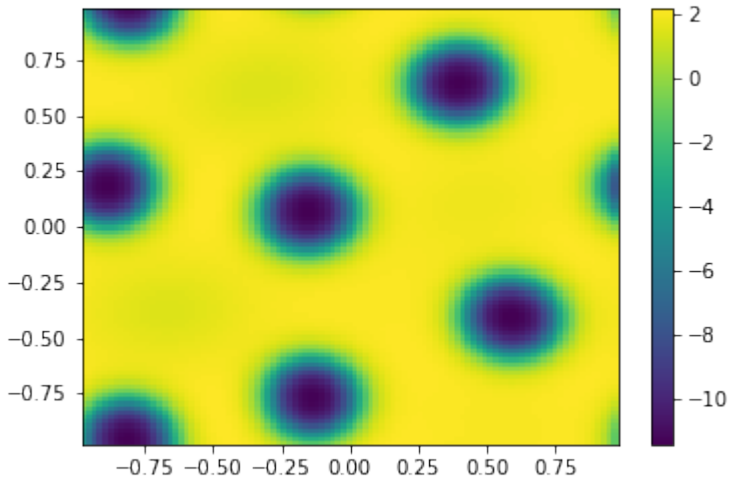
```
<ipython-input-14-14a28ac5572b>:4: MatplotlibDeprecationWarning: shading='flat' when X␣
↪and Y have the same dimensions as C is deprecated since 3.3.  Either specify the␣
↪corners of the quadrilaterals with X and Y, or pass shading='auto', 'nearest' or␣
↪'gouraud', or set rcParams['pcolor.shading'].  This will become an error two minor␣
↪releases later.
```

(continues on next page)

(continued from previous page)

```
  plt.pcolor(X,Y,Ut)
<ipython-input-14-14a28ac5572b>:8: MatplotlibDeprecationWarning: shading='flat' when X␣
↪and Y have the same dimensions as C is deprecated since 3.3.  Either specify the␣
↪corners of the quadrilaterals with X and Y, or pass shading='auto', 'nearest' or
↪'gouraud', or set rcParams['pcolor.shading'].  This will become an error two minor␣
↪releases later.
  plt.pcolor(X,Y,Vt)
```





3.2.4 Efficient Reaction Diffusion

[15]:
```
def Reaction(t,x,parameters):
    u = x[0:m**2]; v = x[m**2:]; #We grab the useful quantity.
    d1 = parameters["delta1"]; d2 = parameters["delta2"];
    t1 = parameters["tau1"]; t2 = parameters["tau2"];
    a = parameters["alpha"]; b = parameters["beta"]; g = parameters["gamma"];
    #Reaction
    du = a*u*(1-t1*(v**2))+v*(1-t2*u);
    dv = b*v*(1+(a*t1/b)*(u*v))+u*(g+t2*v);
```

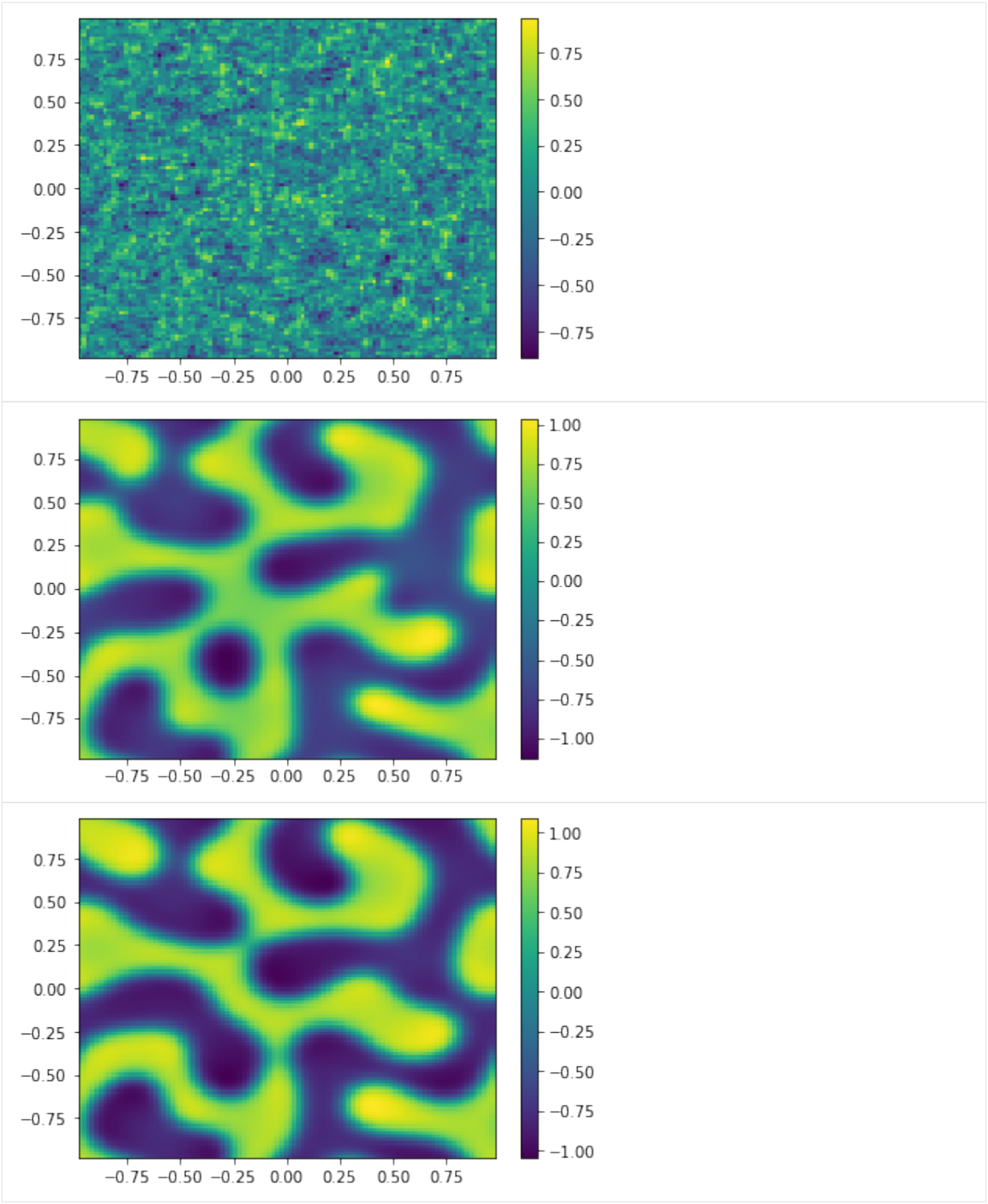
(continues on next page)

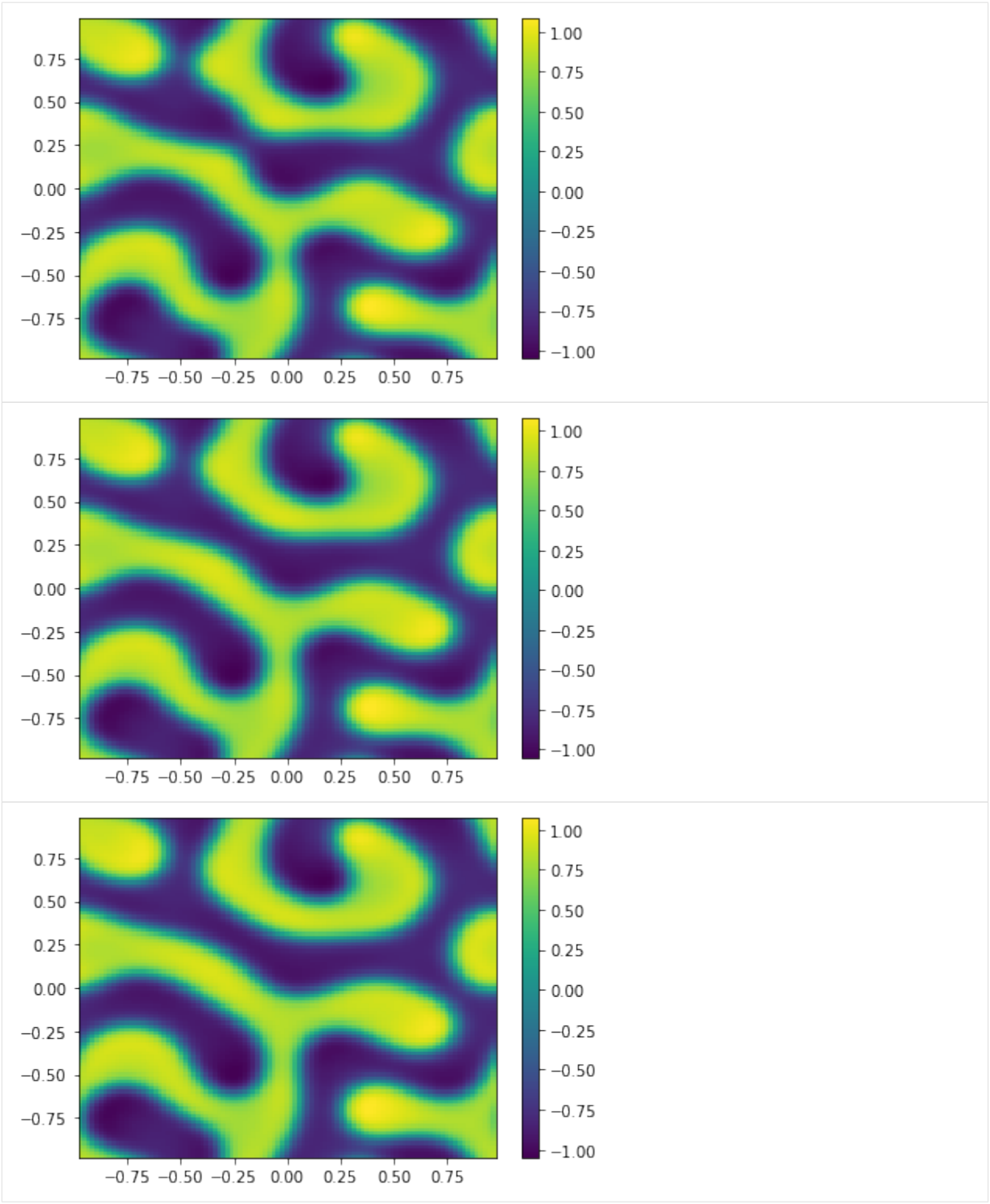
(continued from previous page)

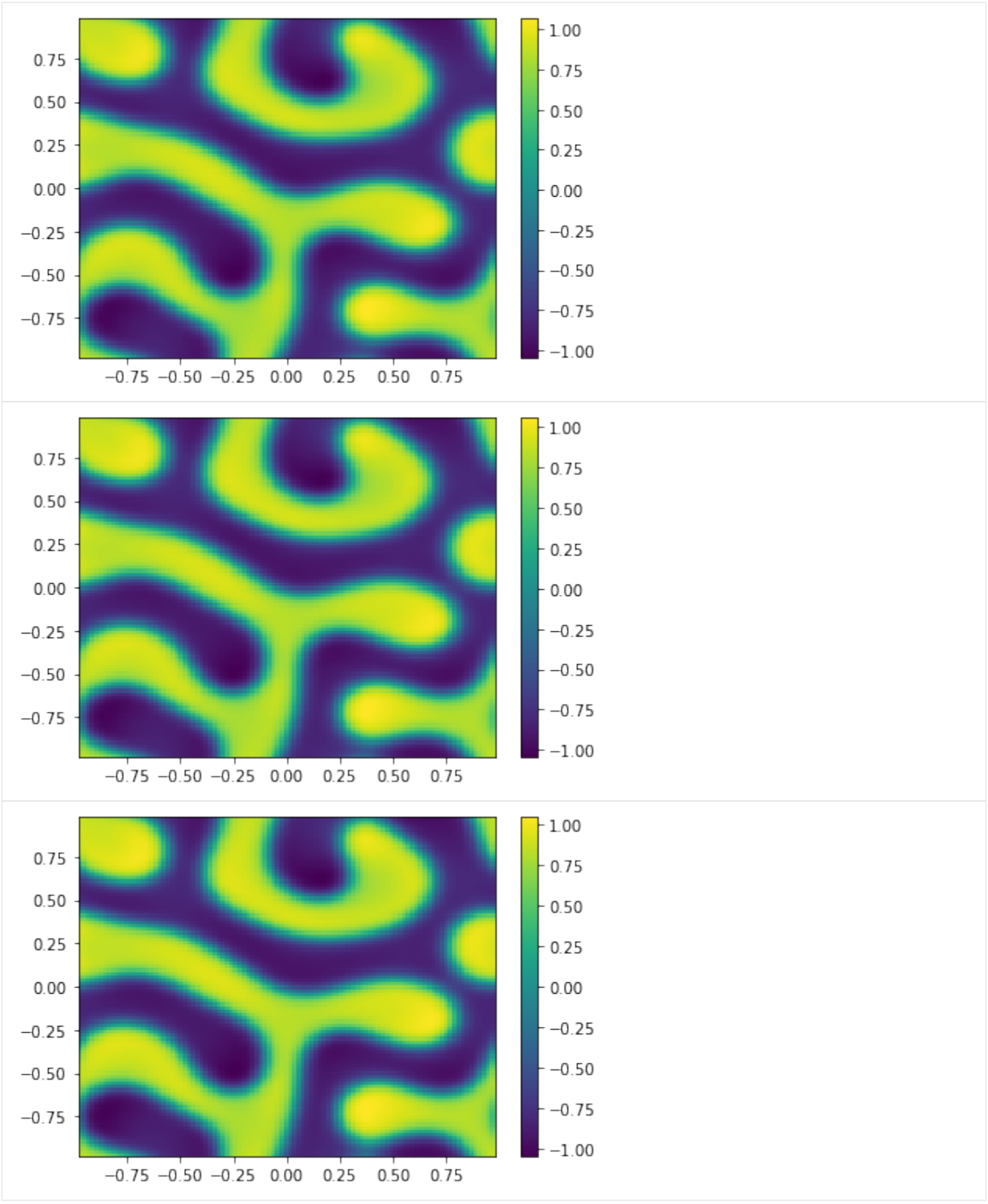
```
    b = np.append(du,dv,axis=0);
    return b;
def SplitSolver(F,T,U0,arg,N):
    tt = np.linspace(T[0],T[1],N);
    h = tt[1]-tt[0];
    U = np.zeros([len(U0),N]);
    U[:,0] = U0;
    for i in trange(0,N-1):
        B = sparse.bmat([[arg["delta1"]*A,None],[None,arg["delta2"]*A]]);
        UStar = spsolve((sparse.identity(B.shape[0])-h*B),U[:,i])
        Y1 = UStar;
        Y2 = UStar + 0.5*h*F(tt[i],Y1,arg);
        Y3 = UStar + 0.5*h*F(tt[i]+0.5*h,Y2,arg);
        Y4 = UStar + h*F(tt[i]+0.5*h,Y3,arg);
        U[:,i+1] = UStar+(h/6)*(F(tt[i],Y1,arg)+2*F(tt[i]+0.5*h,Y2,arg)+2*F(tt[i]+0.5*h,
↪Y3,arg)+F(tt[i]+h,Y4,arg))
        if (i%100==0):
            Ut=U[0:m**2,i+1].reshape([m,m])
            plt.pcolor(X,Y,Ut)
            plt.colorbar();
            plt.show()
    sol = ODESol(tt,h,U);
    return sol;
TIndex = 3;
t0 = 0.0                    # Initial time
tfinal = 150               # Final time
dt_output=0.1             # Interval between output for plotting
N=int(tfinal/dt_output)       # Number of output times
#ODE = scipy.integrate.solve_ivp(Diffusion,[t0,tfinal],np.append(u0,v0,axis=0),
↪args=[Table[TIndex]],method='RK45',t_eval=tt,atol=1.e-10,rtol=1.e-10);
ODE = SplitSolver(Reaction,[t0,tfinal],np.append(u0,v0,axis=0),Table[TIndex],N);
uu=ODE.y
ut = uu[0:m**2,-1];
vt = uu[m**2:,-1];
Ut=ut.reshape([m,m])
plt.pcolor(X,Y,Ut)
plt.colorbar();
plt.figure()
Vt=vt.reshape([m,m])
plt.pcolor(X,Y,Vt)
plt.colorbar();
```

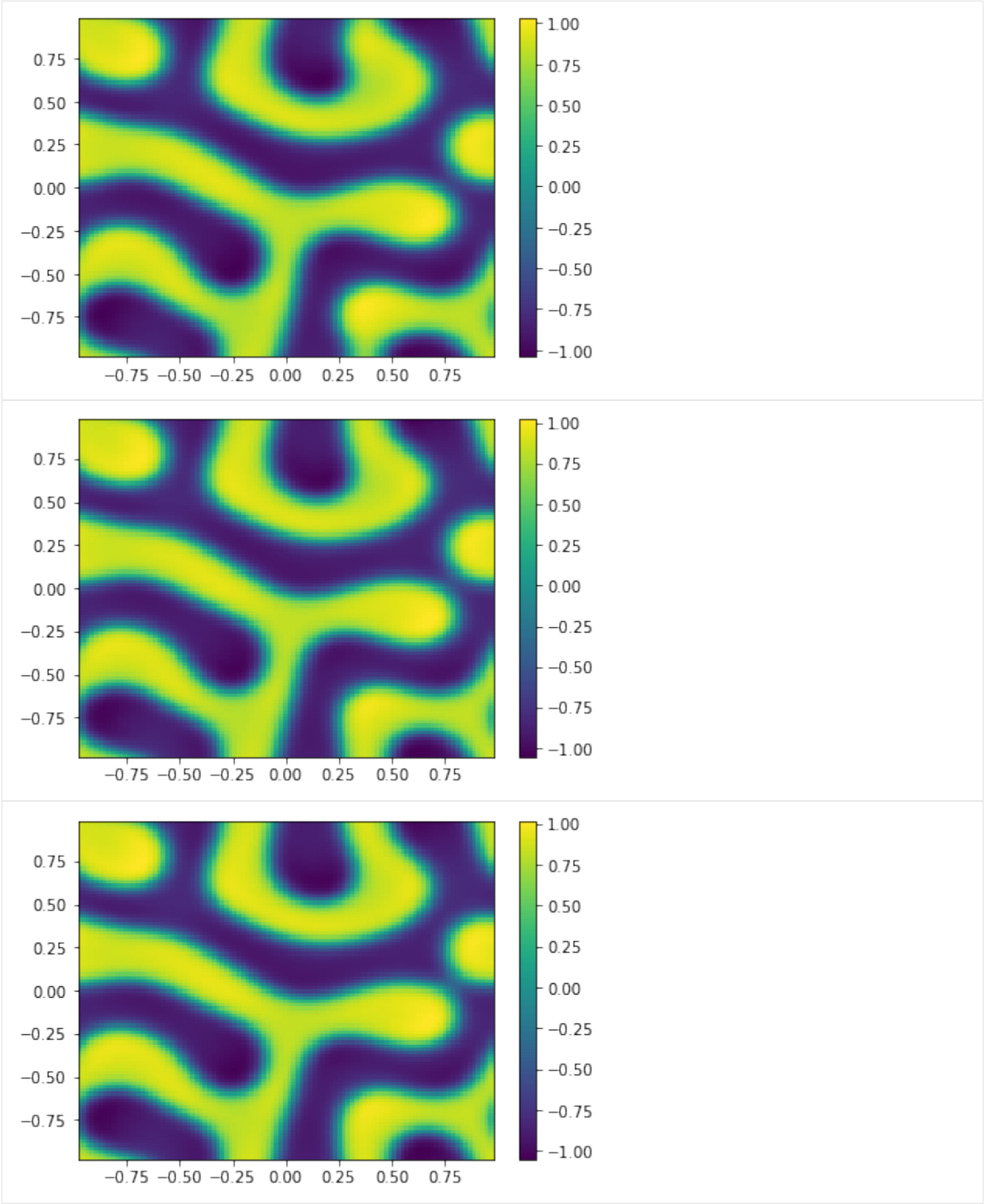
```
  0%|          | 0/1499 [00:00<?, ?it/s]
```

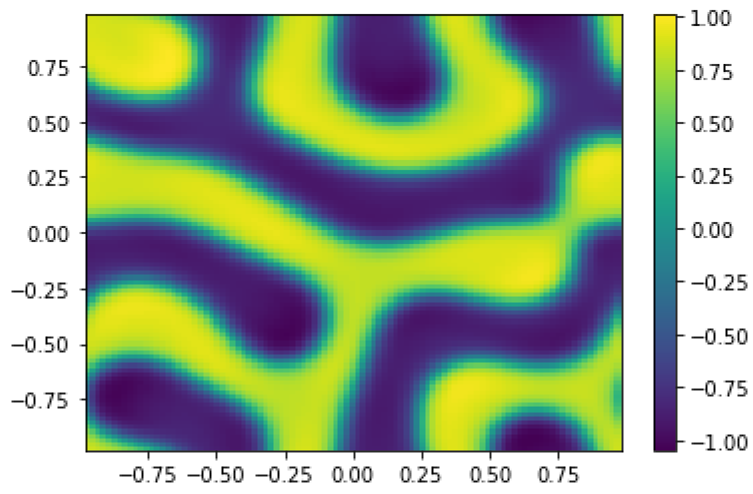
```
<ipython-input-15-45dc21ecdffc>:26: MatplotlibDeprecationWarning: shading='flat' when X␣
↪and Y have the same dimensions as C is deprecated since 3.3.  Either specify the␣
↪corners of the quadrilaterals with X and Y, or pass shading='auto', 'nearest' or
↪'gouraud', or set rcParams['pcolor.shading'].  This will become an error two minor␣
↪releases later.
  plt.pcolor(X,Y,Ut)
```

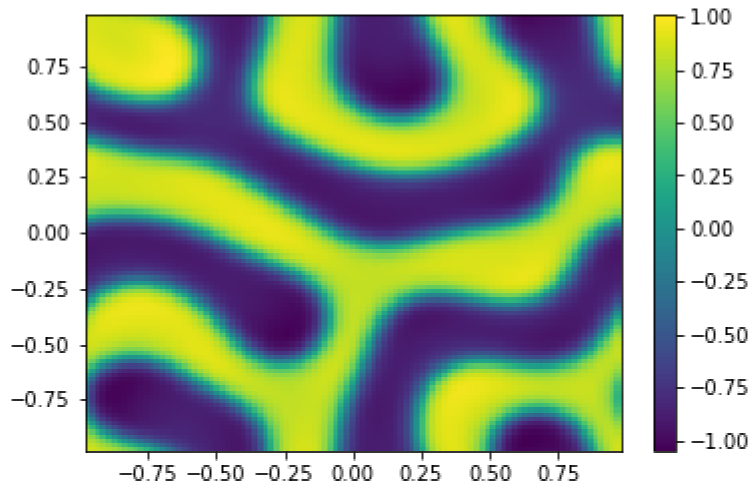

0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−0.75 −0.50 −0.25 0.00 0.25 0.50 0.75
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00

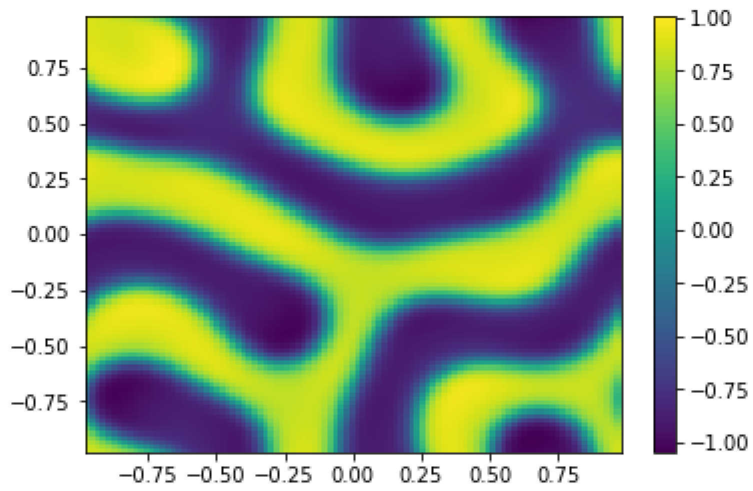

0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−0.75 −0.50 −0.25 0.00 0.25 0.50 0.75
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00


0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−0.75 −0.50 −0.25 0.00 0.25 0.50 0.75
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00


0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−0.75 −0.50 −0.25 0.00 0.25 0.50 0.75
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00





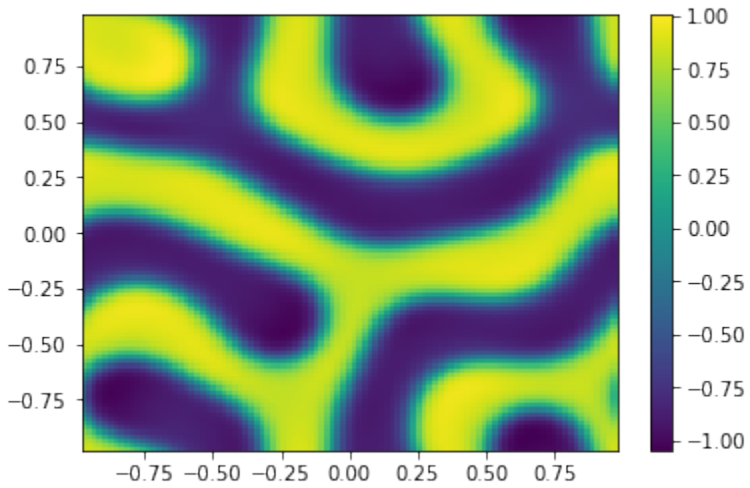


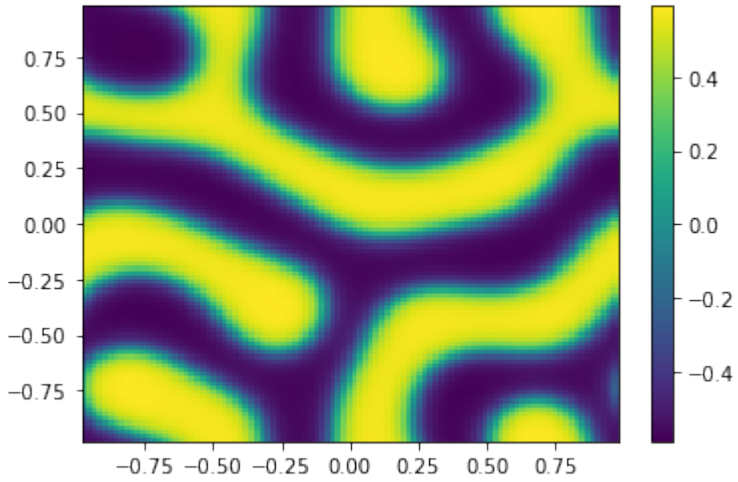

```
<ipython-input-15-45dc21ecdffc>:42: MatplotlibDeprecationWarning: shading='flat' when X␣
↪and Y have the same dimensions as C is deprecated since 3.3.  Either specify the␣
↪corners of the quadrilaterals with X and Y, or pass shading='auto', 'nearest' or
↪'gouraud', or set rcParams['pcolor.shading'].  This will become an error two minor␣
↪releases later.
```

(continues on next page)

(continued from previous page)

```
  plt.pcolor(X,Y,Ut)
<ipython-input-15-45dc21ecdffc>:46: MatplotlibDeprecationWarning: shading='flat' when X␣
↪and Y have the same dimensions as C is deprecated since 3.3.  Either specify the␣
↪corners of the quadrilaterals with X and Y, or pass shading='auto', 'nearest' or
↪'gouraud', or set rcParams['pcolor.shading'].  This will become an error two minor␣
↪releases later.
  plt.pcolor(X,Y,Vt)
```





CHAPTER

FOUR

INDICES

- genindex
- modindex
- search