helmholtz

December 28, 2022

1 Numerical solution of the Helmholtz equation

1.1 Description

1.1.1 PDE

We will try to solve the following pde:

$\Delta\psi(x,y) + k^2\psi(x,y) = -f(x,y)$ on $\Omega = [0,1]^2$

where $f(x,y) = 2k^2e^{jk(x+y)} - k^2e^{jk(x+y)} = k^2e^{jk(x+y)}$

1.1.2 Boundary conditions

$\psi(0,y) = e^{jky}, \quad \psi(1,y) = e^{jk(1+y)}, \quad \psi(x,0) = e^{jkx} \quad$ and $ (x,1)=e^{jk(x+1)} $

1.1.3 Analytical solution

The solution ψ_{th} should be $\psi_{th}(x,y) = e^{jk(x+y)}$

1.1.4 Approximated solution

We want to find a approximated solution $\psi_a(x,y) = N(x,y)$ such that described in the second Lagaris's paper

1.1.5 Loss function

The loss to minimize here is $\mathcal{L} = \frac{1}{N}||\Delta\psi_a(x,y) + k^2\psi_a(x,y) + f(x,y)||_2^2 + \frac{1}{M}\eta||\psi_a|_{\partial\Omega} - b||_2^2$,

where N and M are the size of the training data inside the domain and boundary. b is the value of each point at the boundary

2 Libraries

[30]:
```
import jax, optax
import pickle
import functools
import matplotlib.pyplot
import numpy
%matplotlib inline
```

```
# Set and verify device
jax.config.update('jax_platform_name', 'gpu')
jax.config.update("jax_enable_x64", True)
print(jax.lib.xla_bridge.get_backend().platform)
```

gpu

3 Parameters

[31]:
```
# Neural network parameters
parameters = {}
parameters['seed'] = 351
parameters['n_features'] = 2        # Input dimension (x1, x2)
parameters['n_targets'] = 2         # Output dimension. It's a complex number␣
↪(y1 + j*y2)
parameters['hidden_layers'] = [50, 50, 50, 50, 50]   # Hidden layers structure
parameters['layers'] = [parameters['n_features']] + parameters['hidden_layers']␣
↪+ [parameters['n_targets']]
parameters['eta'] = 1.0

# Training parameters
parameters['learning_rate'] = optax.linear_schedule(0.005, 0.00001,␣
↪transition_steps = 50, transition_begin = 5000)
parameters['optimizer'] = optax.adam(parameters['learning_rate'])
parameters['maximum_num_epochs'] = 50000
parameters['report_steps'] = 1000
parameters['options'] = 1           # 1: we start a new training. 2: We␣
↪continue the last training.
                                    # Other cases: We just load the last␣
↪training

# Data parameters
parameters['n_inside'] = 100        # number of points inside the domain
parameters['n_bound'] = 60          # number of points at the boundary
parameters['domain_bounds'] = jax.numpy.column_stack(([0.0, 0.0], [1.0, 1.0])) ␣
↪ # minimal and maximal value of each axis (x, y)
```

4 Neural network

[32]:
```
class MLP:
    """
        Create a multilayer perceptron and initialize the neural network
    Inputs :
        A SEED number and the layers structure
    """
```

```python
    def __init__(self, key, layers):
        self.key = key
        self.keys = jax.random.split(self.key,len(layers))
        self.layers = layers
        self.params = []

    def MLP_create(self):
        """
        Initialize the MLP weigths and bias
        Parameters
        ----------
        Returns
        -------
        params : list of parameters[[w1,b1],...,[wn,bn]]
            -- weights and bias
        """
        for layer in range(0, len(self.layers)-1):
            in_size,out_size = self.layers[layer], self.layers[layer+1]
            weights = jax.nn.initializers.glorot_normal()(self.keys[layer],␣
↪(out_size, in_size), jax.numpy.float32)
            bias = jax.nn.initializers.lecun_normal()(self.keys[layer],␣
↪(out_size, 1), jax.numpy.float32).reshape((out_size, ))
            self.params.append((weights, bias))
        return self.params


    @functools.partial(jax.jit, static_argnums=(0,))
    def NN_evaluation(self, params, inputs):
        """
        Evaluate a position (x,y) using the neural network
        Parameters
        ----------
        params : list of parameters[[w1,b1],...,[wn,bn]]
            -- weights and bias
        inputs : jax.numpy.ndarray[[batch_size,batch_size]]
            -- points in the domain
        Returns
        -------
        output : jax.numpy.array[batch_size]
            -- neural network output
        """
        for layer in range(0, len(params)-1):
            weights, bias = params[layer]
            inputs = jax.nn.tanh(jax.numpy.add(jax.numpy.dot(inputs, weights.
↪T), bias))
        weights, bias = params[-1]
```

```
        real_and_imaginary_layers = jax.numpy.dot(inputs, weights.T)+bias  #␣
↪The first output of the NN is the real part, the second is the imaginary part
        output = jax.lax.complex(real_and_imaginary_layers[0],␣
↪real_and_imaginary_layers[1])

        return output
```

5 Operators

[33]:
```
class PDE_operators:
    """
        Class with the operators used to solve the PDE
    Input:
        A function that we want to compute the respective operator
    """
    def __init__(self, function):
        self.function = function

    @functools.partial(jax.jit, static_argnums=(0,))
    def laplacian_2d(self, params, inputs):
        """
        Compute the two dimensional laplacian.
        Parameters
        ----------
        params : list of parameters[[w1,b1],...,[wn,bn]]
            -- weights and bias
        inputs : jax.numpy.ndarray[[batch_size,batch_size]]
            -- coordinates (x, y)
        Returns
        -------
        laplacian : jax.numpy.ndarray[batch_size]
            -- numerical values of the laplacian applied to the inputs
        """
        fun = lambda params,x, y: self.function(params, x, y)
        @functools.partial(jax.jit)
        def action(params,x, y):
            u_xx = jax.jacfwd(jax.jacfwd(fun, 1), 1)(params,x, y)
            u_yy = jax.jacfwd(jax.jacfwd(fun, 2), 2)(params,x, y)
            return u_xx + u_yy
        vec_fun = jax.vmap(action, in_axes = (None, 0, 0))
        laplacian = vec_fun(params, inputs[:,0], inputs[:,1]).flatten()
        return laplacian
```

6 Physics Informed Neural Network

[34]:
```
class PINN:
    """
    Solve a PDE using Physics Informed Neural Networks
    Input:
        The evaluation function of the neural network and the optimizer␣
↪selected to do gradient descent
    """
    def __init__(self, NN_evaluation, optimizer):
        self.NN_evaluation = NN_evaluation
        self.optimizer = optimizer

        self.operators = PDE_operators(self.spatial_solution2d)
        self.laplacian2d = self.operators.laplacian_2d

        self.k_coeff = 0.5  # Wavenumber


    @functools.partial(jax.jit, static_argnums = (0, ))
    def spatial_solution2d(self, params, inputX, inputY):
        """
        Compute the complex solution of the PDE on the points (x, y)
        Parameters
        ----------
        params : list of parameters[[w1,b1],...,[wn,bn]]
            -- weights and bias
        inputX : jax.numpy.array[batch_size]
            -- points on the x-axis of the domain
        inputY : jax.numpy.array[batch_size]
            -- points on the y-axis of the domain
        Returns
        -------
        applied_solution : jax.numpy.array[batch_size]
            -- PINN solution applied to inputs. a complex array
        """
        inputs = jax.numpy.column_stack((inputX, inputY))
        NN = jax.vmap(functools.partial(jax.jit(self.NN_evaluation),␣
↪params))(inputs)

        return NN


    @functools.partial(jax.jit, static_argnums=(0,))
    def loss_boundary(self, params, inputs, values):
        """
        Compute the loss function at the boundary
```

```
        Parameters
        ----------
        params : list of parameters[[w1,b1],...,[wn,bn]]
            -- weights and bias
        inputs : jax.numpy.ndarray[[M, M]]
            -- (x,y) points from boundary
        values : jax.numpy.ndarray[[M, M]]
            -- values of the couple (x,y) in the boundary
        Returns
        -------
        loss_bound : a float.64
            -- loss function applied to inputs
        """
        exact_bound = values
        preds_bound = self.spatial_solution2d(params, inputs[:,0], inputs[:,1])
        loss_bound = (jax.numpy.linalg.norm(preds_bound - exact_bound)**2)/
↪inputs.shape[0]

        return loss_bound


    # Definition of the pde mentioned above
    @functools.partial(jax.jit, static_argnums = (0, ))
    def loss_residual(self, params, inputs):
        """
        Compute the residual of the pde
        Parameters
        ----------
        params : list of parameters[[w1,b1],...,[wn,bn]]
            -- weights and bias
        inputs : jax.numpy.ndarray[[N, N]]
            -- (x,y) points from domain
        Returns
        -------
        loss_residual : a float.64
            -- loss function applied to inputs
        """
        j_number = jax.lax.complex(0.0,1.0)
        exact_pde_values = self.k_coeff**2*jax.numpy.exp(j_number*self.
↪k_coeff*(inputs[:,0] + inputs[:,1]))
        pred_pde_values = self.laplacian2d(params, inputs) + self.
↪k_coeff**2*self.spatial_solution2d(params, inputs[:,0], inputs[:,1])
        loss_res = (jax.numpy.linalg.norm(pred_pde_values +␣
↪exact_pde_values)**2)/inputs.shape[0]

        return loss_res
```

```
    # Definition of the loss function mentioned above
    @functools.partial(jax.jit, static_argnums = (0, ))
    def loss_function(self, params, inside_points, boundary_points,␣
↪boundary_values):
        """
        Compute the sum of each loss function
        Parameters
        ----------
        params : list of parameters[[w1,b1],...,[wn,bn]]
            -- weights and bias
        inside_points : jax.numpy.ndarray[[N,N]]
            -- (x,y) points from the domain
        boundary_points : jax.numpy.ndarray[[M,M]]
            -- (x,y) points from boundary
        Returns
        -------
        loss : a float.64
            -- loss function applied to inputs
        losses : numpy.array(loss_residual, loss_b, loss_i)
            -- current values of each loss function
        """
        loss_res = self.loss_residual(params, inside_points)
        loss_bound = self.loss_boundary(params, boundary_points,␣
↪boundary_values)
        loss_sum = loss_res + parameters['eta']*loss_bound
        losses = jax.numpy.array([loss_res, loss_bound])

        return loss_sum, losses


    # Make one train step
    @functools.partial(jax.jit, static_argnums = (0, ))
    def train_step(self, params, opt_state, inside_points, boundary_points,␣
↪boundary_values):
        """
        Make just one step of the training
        Parameters
        ----------
        params : list of parameters[[w1,b1],...,[wn,bn]]
            -- weights and bias
        opt_state : a tuple given by optax
            -- state(hystorical) of the gradient descent
        inside : jax.numpy.ndarray[[batch_size, batch_size,batch_size]]
            -- (x,y,t) points from domain
        bound : jax.numpy.ndarray[[batch_size, batch_size,batch_size]]
            -- (x,y) points from boundary
```

```
        Returns
        -------
        loss : a float.64
            -- loss function applied to inputs
        new_params : list of parameters[[w1,b1],...,[wn,bn]]
            -- weights and bias updated
        opt_state : a tuple given by optax
            -- update the state(hystorical) of the gradient descent
        losses : jax.numpy.array with the values [loss_resi, loss_bound]
            -- current values of each loss function
        """
        (loss,losses), gradient = jax.value_and_grad(self.loss_function,␣
 ↪has_aux=True)(params, inside_points, boundary_points, boundary_values)
        updates, new_opt_state = self.optimizer.update(gradient, opt_state)
        new_params = optax.apply_updates(params, updates)

        return loss, new_params, new_opt_state, losses
```

7 Analytical solution

[35]:
```
@jax.jit
def analytical_solution(x, y, k = 0.5):
    """
    Compute the analytical solution
    Parameters
    ----------
    inputX : jax.numpy.ndarray[batch_size]
        -- points in the axis x
    inputY : jax.numpy.ndarray[batch_size]
        -- points in the axis y
    Returns
    -------
    sol : jax.numpy.array[batch_size]
        -- analytical solution applied to inputs
    """
    j_number = jax.lax.complex(0.0, 1.0)
    sol = jax.numpy.exp(j_number*k*(x + y))
    return sol
```

8 Dataset creation

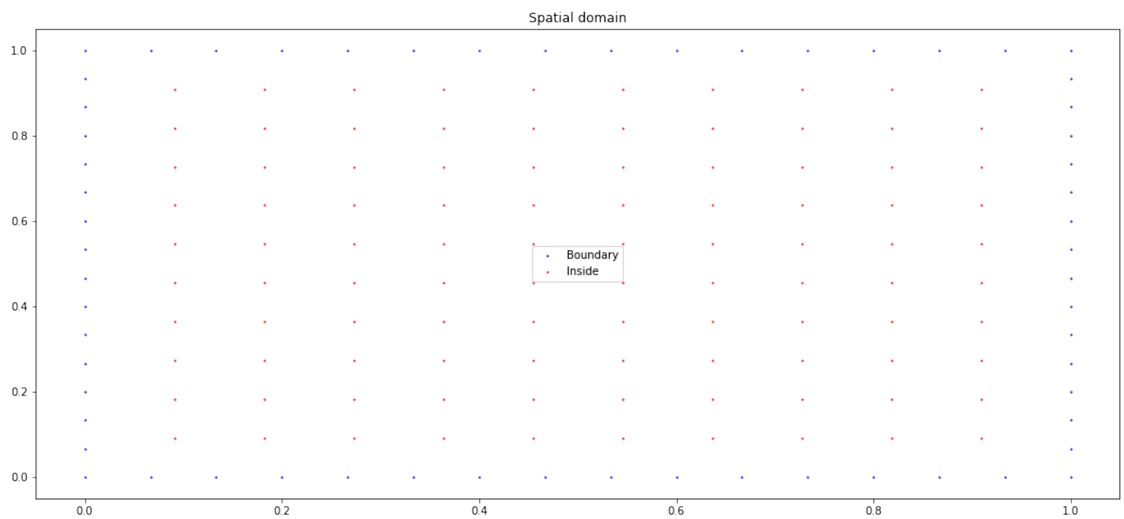
[36]:
```
### Inside data
x = jax.numpy.linspace(parameters['domain_bounds'][0,0],␣
 ↪parameters['domain_bounds'][0,1], int(jax.numpy.
 ↪sqrt(parameters['n_inside']))+1, endpoint=False)[1:]
```

```
y = jax.numpy.linspace(parameters['domain_bounds'][1,0],␣
↪parameters['domain_bounds'][1,1], int(jax.numpy.
↪sqrt(parameters['n_inside']))+1, endpoint=False)[1:]
x, y = jax.numpy.meshgrid(x, y)
x, y = x.flatten(), y.flatten()
XY_inside = jax.numpy.column_stack((x, y))

#### Boundary data ---- 4 edges
### Left
x = jax.numpy.
↪linspace(parameters['domain_bounds'][0,0],parameters['domain_bounds'][0,0],1,␣
↪endpoint = False)
y = jax.numpy.
↪linspace(parameters['domain_bounds'][1,0],parameters['domain_bounds'][1,1],parameters['n_bo
↪/4, endpoint = False)
x, y = jax.numpy.meshgrid(x,y)
x, y = x.flatten(), y.flatten()
xy_left = jax.numpy.column_stack((x, y))


### Behind
x = jax.numpy.
↪linspace(parameters['domain_bounds'][0,0],parameters['domain_bounds'][0,1],parameters['n_bo
↪/4, endpoint = False)
y = jax.numpy.
↪linspace(parameters['domain_bounds'][1,1],parameters['domain_bounds'][1,1],1,␣
↪endpoint = False)
x, y = jax.numpy.meshgrid(x, y)
x, y = x.flatten(), y.flatten()
xy_behind = jax.numpy.column_stack((x, y))


### Right
x = jax.numpy.
↪linspace(parameters['domain_bounds'][0,1],parameters['domain_bounds'][0,1],1,␣
↪endpoint = False)
y = jax.numpy.
↪linspace(parameters['domain_bounds'][1,1],parameters['domain_bounds'][1,0],parameters['n_bo
↪/4, endpoint = False)
x, y = jax.numpy.meshgrid(x, y)
x, y = x.flatten(), y.flatten()
xy_right = jax.numpy.column_stack((x, y))


### Front
```

```
x = jax.numpy.
 ↪linspace(parameters['domain_bounds'][0,1],parameters['domain_bounds'][0,0],parameters['n_bo
 ↪/4, endpoint = False)
y = jax.numpy.
 ↪linspace(parameters['domain_bounds'][1,0],parameters['domain_bounds'][1,0],1,␣
 ↪endpoint = False)
x, y = jax.numpy.meshgrid(x,y)
x, y = x.flatten(), y.flatten()
xy_front = jax.numpy.column_stack((x, y))

XY_bound = jax.numpy.concatenate((xy_left, xy_behind, xy_right, xy_front))
XY_bound_values = analytical_solution(XY_bound[:,0], XY_bound[:,1])
```

9 Dataset plot

[45]:
```
fig, ax = matplotlib.pyplot.subplots()
fig.set_size_inches(18, 8.0)
title = ax.set_title('Spatial domain')
graph = matplotlib.pyplot.scatter(XY_bound[:,0],XY_bound[:,1],color='b',s=1)
graph = matplotlib.pyplot.scatter(XY_inside[:,0],XY_inside[:,1],color='r',s=1)
__ = ax.legend(['Boundary','Inside'])
matplotlib.pyplot.savefig('/content/gdrive/My Drive/Colab/PINNs/Helmholtz/
 ↪Images/domain.png', bbox_inches = 'tight', facecolor='white')
matplotlib.pyplot.show()

print(f"Number of points inside the domain: {XY_inside.shape[0]}")
print(f"Number of points at the boundary: {XY_bound.shape[0]}")
```




```
Number of points inside the domain: 100
```

```
Number of points at the boundary: 60
```

10 Model initialization

[38]:
```
key = jax.random.PRNGKey(parameters['seed'])
NN_MLP = MLP(key, parameters['layers'])
params = NN_MLP.MLP_create()            # Create the MLP
NN_eval = NN_MLP.NN_evaluation            # Evaluation function
solver = PINN(NN_eval, parameters['optimizer'])
opt_state = parameters['optimizer'].init(params)
```

11 Training

[39]:
```
loss_history = []
loss_residual = []               # residual loss
loss_boundary = []               # boundary loss

print("Training start")
if parameters['options'] == 1:             # start a new training
    # Main loop to solve the PDE
    for ibatch in range(parameters['maximum_num_epochs']+1):

        loss, params, opt_state, losses = solver.train_step(params,opt_state,␣
 ↪XY_inside, XY_bound, XY_bound_values)

        loss_residual.append(float(losses[0]))
        loss_boundary.append(float(losses[1]))
        losssum = jax.numpy.sum(losses)
        loss_history.append(float(losssum))

        if ibatch%parameters['report_steps']==parameters['report_steps']-1:
            print("Epoch n°{}: ".format(ibatch+1), losssum.item())

        if losssum<=numpy.min(loss_history): # save if the current state is the␣
 ↪best
            pickle.dump(params, open("/content/gdrive/My Drive/Colab/PINNs/
 ↪Helmholtz/Checkpoints/params_helmholtz", "wb"))
            pickle.dump(opt_state, open("/content/gdrive/My Drive/Colab/PINNs/
 ↪Helmholtz/Checkpoints/opt_state_helmholtz", "wb"))
            pickle.dump(loss_history, open("/content/gdrive/My Drive/Colab/
 ↪PINNs/Helmholtz/Checkpoints/loss_history_helmholtz", "wb"))
            pickle.dump(loss_residual, open("/content/gdrive/My Drive/Colab/
 ↪PINNs/Helmholtz/Checkpoints/loss_residual_helmholtz", "wb"))
            pickle.dump(loss_boundary, open("/content/gdrive/My Drive/Colab/
 ↪PINNs/Helmholtz/Checkpoints/loss_boundary_helmholtz", "wb"))
```

```
elif parameters['options'] == 2:      # continue the last training
    params = pickle.load(open("/content/gdrive/My Drive/Colab/PINNs/Helmholtz/
↪Checkpoints/params_helmholtz", "rb"))
    opt_state = pickle.load(open("/content/gdrive/My Drive/Colab/PINNs/
↪Helmholtz/Checkpoints/opt_state_helmholtz", "rb"))
    loss_history = pickle.load(open("/content/gdrive/My Drive/Colab/PINNs/
↪Helmholtz/Checkpoints/loss_history_helmholtz", "rb"))
    loss_residual = pickle.load(open("/content/gdrive/My Drive/Colab/PINNs/
↪Helmholtz/Checkpoints/loss_residual_helmholtz", "rb"))
    loss_boundary = pickle.load(open("/content/gdrive/My Drive/Colab/PINNs/
↪Helmholtz/Checkpoints/loss_boundary_helmholtz", "rb"))
    iepoch = len(loss_history)

    # Main loop to solve the PDE
    for ibatch in range(iepoch, parameters['maximum_num_epochs']+1):

        loss, params, opt_state, losses = solver.train_step(params,opt_state,␣
↪XY_inside, XY_bound, XY_bound_values)

        loss_residual.append(float(losses[0]))
        loss_boundary.append(float(losses[1]))
        losssum = jax.numpy.sum(losses)
        loss_history.append(float(losssum))

        if ibatch%parameters['report_steps']==parameters['report_steps']-1:
            print("Epoch n°{}: ".format(ibatch+1), losssum.item())

        if losssum<=numpy.min(loss_history): # save if the current state is the␣
↪best
            pickle.dump(params, open("/content/gdrive/My Drive/Colab/PINNs/
↪Helmholtz/Checkpoints/params_helmholtz", "wb"))
            pickle.dump(opt_state, open("/content/gdrive/My Drive/Colab/PINNs/
↪Helmholtz/Checkpoints/opt_state_helmholtz", "wb"))
            pickle.dump(loss_history, open("/content/gdrive/My Drive/Colab/
↪PINNs/Helmholtz/Checkpoints/loss_history_helmholtz", "wb"))
            pickle.dump(loss_residual, open("/content/gdrive/My Drive/Colab/
↪PINNs/Helmholtz/Checkpoints/loss_residual_helmholtz", "wb"))
            pickle.dump(loss_boundary, open("/content/gdrive/My Drive/Colab/
↪PINNs/Helmholtz/Checkpoints/loss_boundary_helmholtz", "wb"))

else:
    params = pickle.load(open("/content/gdrive/My Drive/Colab/PINNs/Helmholtz/
↪Checkpoints/params_helmholtz", "rb"))
```

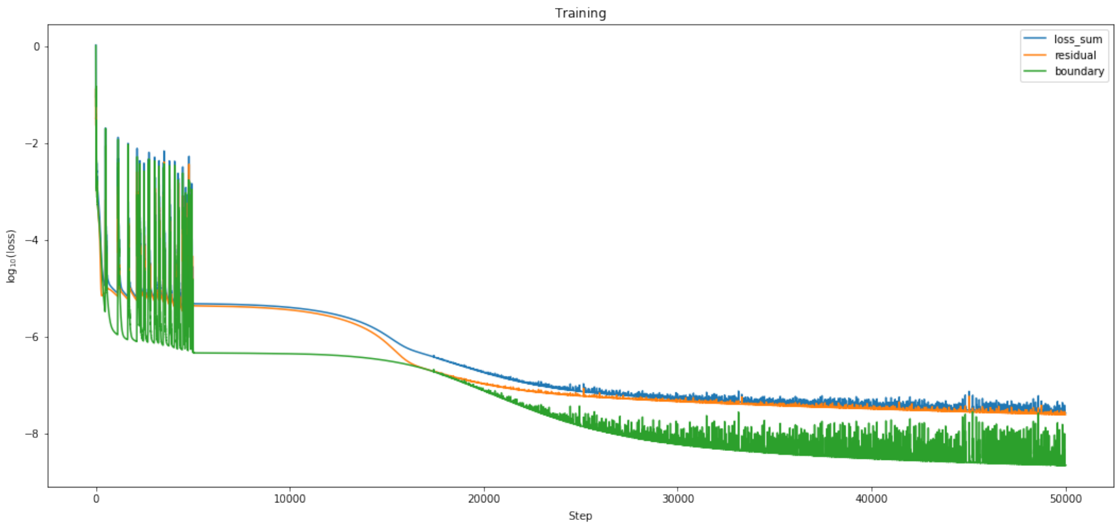
```
    opt_state = pickle.load(open("/content/gdrive/My Drive/Colab/PINNs/
↪Helmholtz/Checkpoints/opt_state_helmholtz", "rb"))
    loss_history = pickle.load(open("/content/gdrive/My Drive/Colab/PINNs/
↪Helmholtz/Checkpoints/loss_history_helmholtz", "rb"))
    loss_residual = pickle.load(open("/content/gdrive/My Drive/Colab/PINNs/
↪Helmholtz/Checkpoints/loss_residual_helmholtz", "rb"))
    loss_boundary = pickle.load(open("/content/gdrive/My Drive/Colab/PINNs/
↪Helmholtz/Checkpoints/loss_boundary_helmholtz", "rb"))
```

```
Training start
Epoch n°1000:  8.719329066262856e-06
Epoch n°2000:  7.262592146136989e-06
Epoch n°3000:  6.288189073918382e-06
Epoch n°4000:  5.190959847847501e-06
Epoch n°5000:  7.2603547478888216e-06
Epoch n°6000:  4.7037527487899885e-06
Epoch n°7000:  4.6165026925892476e-06
Epoch n°8000:  4.4816285350428966e-06
Epoch n°9000:  4.278457898050031e-06
Epoch n°10000:  3.98561100676301e-06
Epoch n°11000:  3.5838059198060872e-06
Epoch n°12000:  3.0648174260013897e-06
Epoch n°13000:  2.439774102313495e-06
Epoch n°14000:  1.740597088882981e-06
Epoch n°15000:  1.045671426900509e-06
Epoch n°16000:  5.889909550033537e-07
Epoch n°17000:  4.2492005548973985e-07
Epoch n°18000:  3.174230519954772e-07
Epoch n°19000:  2.538926256728737e-07
Epoch n°20000:  1.8009248855566928e-07
Epoch n°21000:  1.4058818399830672e-07
Epoch n°22000:  1.1328217071142239e-07
Epoch n°23000:  9.484011991735067e-08
Epoch n°24000:  8.909157476719348e-08
Epoch n°25000:  7.31015948768099e-08
Epoch n°26000:  6.721062855412622e-08
Epoch n°27000:  6.147054200165567e-08
Epoch n°28000:  6.977124516878786e-08
Epoch n°29000:  5.4096862715422166e-08
Epoch n°30000:  6.462058845698156e-08
Epoch n°31000:  5.8309963671733466e-08
Epoch n°32000:  5.09763235962723e-08
Epoch n°33000:  4.4703542099466226e-08
Epoch n°34000:  4.371604365706995e-08
Epoch n°35000:  4.272553624995411e-08
Epoch n°36000:  3.988090010733721e-08
Epoch n°37000:  3.952555354543024e-08
```

```
Epoch n°38000:  3.9786382558319834e-08
Epoch n°39000:  3.604430799084907e-08
Epoch n°40000:  3.505339854843323e-08
Epoch n°41000:  3.421210882867841e-08
Epoch n°42000:  3.579200483315895e-08
Epoch n°43000:  3.262067350519102e-08
Epoch n°44000:  3.306898275242547e-08
Epoch n°45000:  3.011082389611286e-08
Epoch n°46000:  2.9320920560166363e-08
Epoch n°47000:  2.8742052441719195e-08
Epoch n°48000:  2.7866088797395117e-08
Epoch n°49000:  2.7301029774833818e-08
Epoch n°50000:  2.634568178630608e-08
```

12 Loss function plot

[40]:
```python
fig, ax = matplotlib.pyplot.subplots(1, 1)
fig.set_size_inches(18, 8.0)
__ = ax.plot(numpy.log10(loss_history))
__ = ax.plot(numpy.log10(numpy.array(loss_residual)))
__ = ax.plot(numpy.log10(numpy.array(loss_boundary)))
xlabel = ax.set_xlabel(r'${\rm Step}$')
ylabel = ax.set_ylabel(r'$\log_{10}{\rm (loss)}$')
title = ax.set_title(r'${\rm Training}$')
ax.legend(['loss_sum','residual','boundary'])
matplotlib.pyplot.savefig('/content/gdrive/My Drive/Colab/PINNs/Helmholtz/
↪Images/loss_function.png', bbox_inches = 'tight', facecolor='white')
matplotlib.pyplot.show()
```

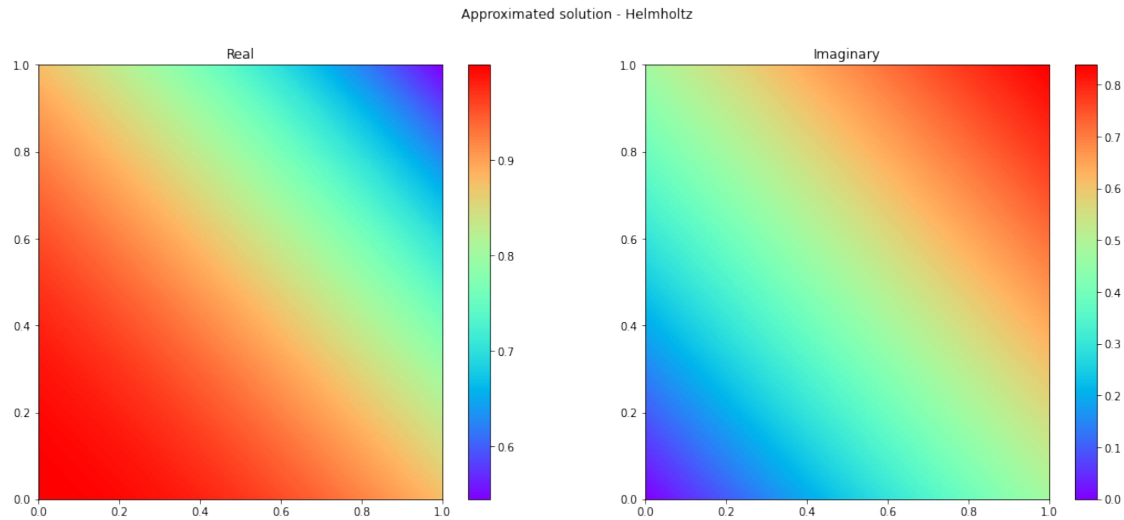


13 Load best params of the training

[41]:
```
params = pickle.load(open("/content/gdrive/My Drive/Colab/PINNs/Helmholtz/
↪Checkpoints/params_helmholtz", "rb"))
```

14 Approximated solution plot

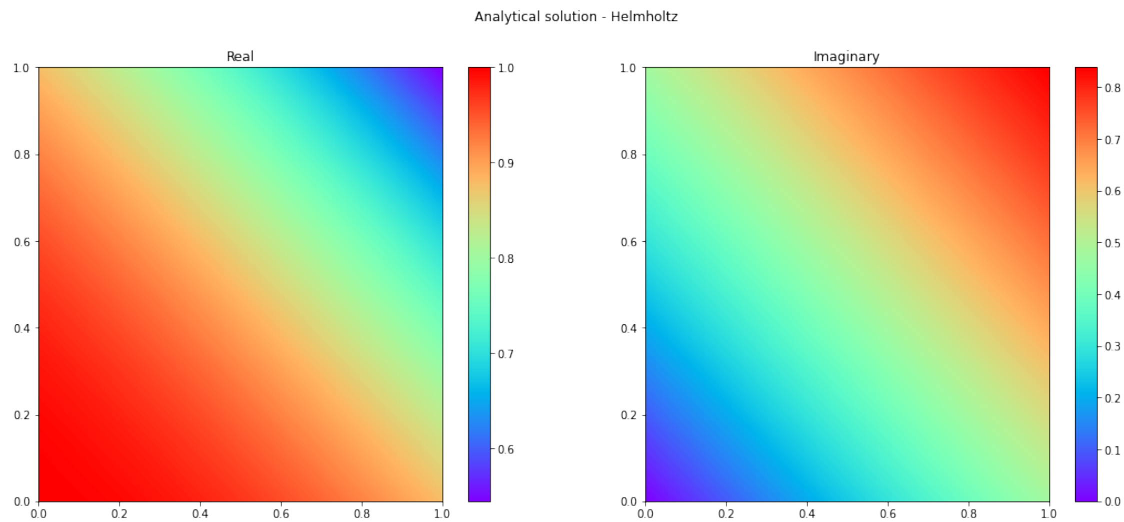
[42]:
```
npoints = 200
real_values = numpy.zeros((npoints, npoints))
imag_values = numpy.zeros((npoints, npoints))

x, y = numpy.meshgrid(numpy.linspace(parameters['domain_bounds'][0,0],␣
↪parameters['domain_bounds'][0,1], npoints), numpy.
↪linspace(parameters['domain_bounds'][1,0], parameters['domain_bounds'][1,1],␣
↪npoints))

fig, ax = matplotlib.pyplot.subplots(1,2)
fig.set_size_inches(18, 7.2)
title = fig.suptitle('Approximated solution - Helmholtz')

for i in range(npoints):
    print("Plotting: {} out of {}".format(i+1, npoints), end='\r')
    real_values[i,:] = jax.numpy.real(functools.partial(solver.
↪spatial_solution2d, params)(x[i,:], y[i,:]))
    imag_values[i,:] = jax.numpy.imag(functools.partial(solver.
↪spatial_solution2d, params)(x[i,:], y[i,:]))

title = ax[0].set_title('Real')
graph = ax[0].pcolormesh(x, y, real_values, cmap = 'rainbow')
matplotlib.pyplot.colorbar(graph, ax=ax[0])

title = ax[1].set_title('Imaginary')
graph = ax[1].pcolormesh(x, y, imag_values, cmap = 'rainbow')
matplotlib.pyplot.colorbar(graph, ax=ax[1])


matplotlib.pyplot.savefig('/content/gdrive/My Drive/Colab/PINNs/Helmholtz/
↪Images/approximated_helmholtz.png', facecolor = 'white', bbox_inches =␣
↪'tight')
matplotlib.pyplot.show()
```



15 Analytical solution plot

[43]:
```
npoints = 200
real_values = numpy.zeros((npoints, npoints))
imag_values = numpy.zeros((npoints, npoints))

x, y = numpy.meshgrid(numpy.linspace(parameters['domain_bounds'][0,0],␣
↪parameters['domain_bounds'][0,1], npoints), numpy.
↪linspace(parameters['domain_bounds'][1,0], parameters['domain_bounds'][1,1],␣
↪npoints))

fig, ax = matplotlib.pyplot.subplots(1,2)
fig.set_size_inches(18, 7.2)
title = fig.suptitle('Analytical solution - Helmholtz')

for i in range(npoints):
    print("Plotting: {} out of {}".format(i+1, npoints), end='\r')
    real_values[i,:] = jax.numpy.real(functools.
↪partial(analytical_solution)(x[i,:], y[i,:]))
    imag_values[i,:] = jax.numpy.imag(functools.
↪partial(analytical_solution)(x[i,:], y[i,:]))

title = ax[0].set_title('Real')
graph = ax[0].pcolormesh(x, y, real_values, cmap = 'rainbow')
matplotlib.pyplot.colorbar(graph, ax=ax[0])

title = ax[1].set_title('Imaginary')
graph = ax[1].pcolormesh(x, y, imag_values, cmap = 'rainbow')
```

```
matplotlib.pyplot.colorbar(graph, ax=ax[1])


matplotlib.pyplot.savefig('/content/gdrive/My Drive/Colab/PINNs/Helmholtz/
↪Images/analytical_helmholtz.png', facecolor = 'white', bbox_inches = 'tight')
matplotlib.pyplot.show()
```



16 Squared error plot

[44]:

```
npoints = 200
values = numpy.zeros((npoints, npoints))

x, y = numpy.meshgrid(numpy.linspace(parameters['domain_bounds'][0,0],␣
↪parameters['domain_bounds'][0,1], npoints), numpy.
↪linspace(parameters['domain_bounds'][1,0], parameters['domain_bounds'][1,1],␣
↪npoints))

fig, ax = matplotlib.pyplot.subplots()
fig.set_size_inches(18, 8.0)
title = ax.set_title('Squared error - Helmholtz')

for i in range(npoints):
    print("Plotting: {} out of {}".format(i+1, npoints), end='\r')
    real_squared_error = (jax.numpy.real(functools.partial(solver.
↪spatial_solution2d, params)(x[i,:], y[i,:])) - jax.numpy.real(functools.
↪partial(analytical_solution)(x[i,:], y[i,:])))**2
```

```
    imag_squared_error = (jax.numpy.imag(functools.partial(solver.
↪spatial_solution2d, params)(x[i,:], y[i,:])) - jax.numpy.imag(functools.
↪partial(analytical_solution)(x[i,:], y[i,:])))**2
    values[i,:] = real_squared_error + imag_squared_error

print("MSE: ", numpy.mean(values.flatten()))
graph = matplotlib.pyplot.pcolormesh(x, y, values, cmap = 'rainbow')
matplotlib.pyplot.colorbar()
matplotlib.pyplot.savefig('/content/gdrive/My Drive/Colab/PINNs/Helmholtz/
↪Images/squared_error_helmholtz.png', facecolor = 'white', bbox_inches =␣
↪'tight')
matplotlib.pyplot.show()
```

MSE: 6.002742686325822e-10