Supplementary Materials for

Deep learning spatial phase unwrapping: a comparative review

Kaiqiang Wang,[a,b] Qian Kemao,[c,*] Jianglei Di,[a,b,d,*] and Jianlin Zhao[a,b,*]

[a]Northwestern Polytechnical University, School of Physical Science and Technology, Shaanxi Key Laboratory of Optical Information Technology, Xi'an, China
[b]Ministry of Industry and Information Technology, Key Laboratory of Light Field Manipulation and Information Acquisition, Xi'an, China
[c]Nanyang Technological University, School of Computer Science and Engineering, Singapore
[d]Guangdong University of Technology, Guangdong Provincial Key Laboratory of Photonics Information Technology, Guangzhou, China
*Address all correspondence to Qian Kemao, mkmqian@ntu.edu.sg; Jianglei Di, jiangleidi@nwpu.edu.cn; Jianlin Zhao, jlzhao@nwpu.edu.cn

S1. Structure of Res-UNet

The Res-UNet are inspired by U-Net [48], residual block [24, 47] and Inception module [51, 78]. As shown in Fig. S1(a), it consists of an encoding path (left), a decoding path (right) and a bridge path (middle). The encoding and decoding paths each contain four residual blocks, while the residual block of the encoding path is followed by max pooling for downsampling and the residual block of the decoding path is preceded by transposed convolution for upsampling. As shown in Fig. S1(b), the Inception module is inserted into residual block, which includes branch 0, branch 1, branch 2, branch 3 and branch 4.

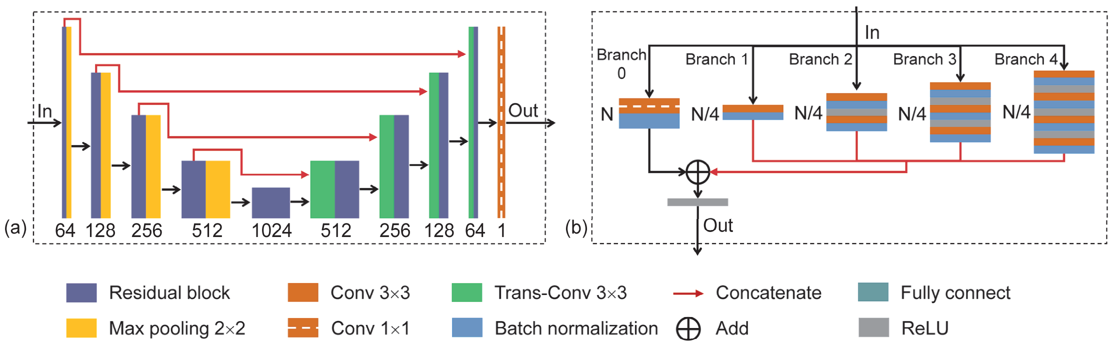

Fig. S1 Structure of (a) Res-UNet and (b) residual block.

S2. Comparison of D_RME and D_RME0

To verify the quality of datasets with different h distributions, we trained Res-UNet by D_RME and D_RME0 as shown in Table S1. Then, the two trained networks (RME-Net and RME0-Net) were tested, whose $RMSE_m$ and $RMSE_{sd}$ are shown in Table S2. It can be seen that the $RMSE_m$ of RME-Net is significantly lower than that of RME0-Net, which indicates that assigning more data with high h to the training dataset can improve the performance of the neural network when other factors (such as the generation method and number of datasets) are the same.

Table S1 Summary of D_RME and D_RME0.

Datasets	sizes	Proportion of h in 10-30	Proportion of h in 30-35	Proportion of h in 35-40
Training part of D_RME	20,000	50%	20%	30%
Testing part of D_RME	2,000	2/3	1/6	1/6
D_RME0 for training	20,000	2/3	1/6	1/6

Table S2 Accuracy estimation of RME-Net and RME0-Net.

		D_RME	D_GFS	D_ZPS	D_RDR
$RMSE_m$	RME-Net	0.0910	0.0982	0.1336	0.1103
	RME0-Net	0.1766	0.1798	0.2019	0.1624
$RMSE_{sd}$	RME-Net	0.0507	0.1037	0.2320	0.1001
	RME0-Net	0.0652	0.1591	0.2265	0.0739

S3. $RMSE_m$ and $RMSE_{sd}$ of the congruence results

To verify the effect of congruence operation, we calculated $RMSE_m$ and $RMSE_{sd}$ for the networks and their congruence results, as shown in Table S3. $RMSE_m$ for almost all the results decreases significantly after the congruence operation, except for ZPS-Net, because ZPS-Net has low raw accuracy on the non-ZPS testing datasets.

Table S3 Accuracy estimation of RME-Net, GSF-Net and ZPS-Net. "-C" represents the congruence results.

		D_RME	D_RME-C	D_GFS	D_GFS-C	D_ZPS	D_RME-C	D_RDR	D_RME-C
$RMSE_m$	RME-Net	0.0910	0.0002	0.0982	0.0069	0.1336	0.0454	0.1103	0.0093
	GSF-Net	0.2263	0.1439	0.0985	0.0007	0.1133	0.0174	0.1184	0.0025
	ZPS-Net	2.5148	2.6141	0.4221	0.3862	0.0821	0.0001	0.8245	0.8307
$RMSE_{sd}$	RME-Net	0.0507	0.0092	0.1037	0.1065	0.2320	0.2455	0.1003	0.0593
	GSF-Net	0.4571	0.5280	0.0234	0.0175	0.1077	0.1278	0.1557	0.1900
	ZPS-Net	2.8249	2.9398	0.6252	0.7390	0.0220	0.0016	1.1405	1.2896

S4. Comparison of D_RME and D_RME1

To compare the quality of D_RME and D_RME1, we calculated $RMSE_m$ for RME-Net and RME1-Net, as shown in Table S4. It can be seen that $RMSE_m$ of RME1-Net is almost half that of RME-Net.

Table S4 Accuracy estimation of RME-Net and RME0-Net.

		D_RME	D_GFS	D_ZPS	D_RDR
$RMSE_m$	RME-Net	0.0910	0.0982	0.1336	0.1103
	RME1-Net	0.0515	0.0468	0.0649	0.0667

S5. A demonstration of dRG phase unwrapping method

In order to enable readers to get started quickly and deeply understand the deep learning-based phase unwrapping method, we provide a detailed demonstration of dRG here, including dataset generation, neural network making, training and testing. All the codes are available in a Github repository (https://github.com/kqwang/Phase_unwrapping_by_U-Net).

S5.1 Dataset generation

Here we use RME to generate the dataset. On the one hand, the parameters (phase size and *h* value range, etc.) are appropriately relaxed to improve the applicability. On the other hand, Gaussian noise is added to improve the anti-noise performance. Readers can further adjust the parameters according to actual needs.

The core parts of dataset generation codes (*dataset_generation.m*) are mainly explained in Fig. S2:

(a) Set all required parameters, which can be adjusted according to the actual needs of readers;
(b) Get initial absolute phase by enlarging a small random matrix;
(c) Set the height *h* so that 50% of the data is within 2/3 of *h*, 20% of the data is between 2/3 of *h* and 5/6 of *h*, and 30% of the data is between 5/6 of *h* and *h*;
(d) Normalize the initial absolute phase to 0-*h* as network ground truth;
(e) Add Gaussian noise with a standard deviation of 0-*noise_max* to the absolute phase (The default value of *noise_max* is 0);
(f) Calculate the wrapped phase from the noisy absolute phase as network input.

```
% parameters
size_min = 2; % minimum size of initial matrix
size_max = 8; % maximum size of initial matrix
height_min = 10; % minimum value of height h
height_max = 40; % maximum value of height h
h_h = height_max - height_min;
train_num = 20000; % data number of training dataset
test_num = 2000; % data number of testing dataset
phase_size = 128; % size of the wrapped and absolute phase
noise_max = 0; % maximum value of the standard deviation of the noise
```

(a)

```
% initial absolute phase
size_xy = randi([size_min,size_max]); % get size of initial matrix
initial_matrix = rand(size_xy,size_xy); % get initial matrix
phase_size_pre = phase_size * 1.25; % get size of pre-enlarged absolute phase
pre_enlarged_ap = imresize(initial_matrix,...
    [phase_size_pre phase_size_pre]);  % get pre-enlarged absolute phase
initial_ap = pre_enlarged_ap((phase_size_pre/2-phase_size/2+1):...
    (phase_size_pre/2+phase_size/2),(phase_size_pre/2-phase_size/2+1)...
    :(phase_size_pre/2+phase_size/2)); % get initial absolute phase
```

(b)

(c)

```
% set height h
if jj <= train_num*0.5
    h(jj) = unifrnd(height_min,(height_min+2/3*h_h), 1, 1);
else if jj <= train_num*0.7
        h(jj) = unifrnd((height_min+2/3*h_h),(height_min+5/6*h_h), 1, 1);
    else
        h(jj) = unifrnd((height_min+5/6*h_h),(height_min+h_h), 1, 1);
    end
end
```

(d)

```
% normalize the absolute phase to [0,h]
ymax=h(jj);ymin=0;
xmax = max(max(initial_ap));
xmin = min(min(initial_ap));
ap = (ymax-ymin)*(initial_ap-xmin)/(xmax-xmin) + ymin;
```

(e)

```
% add noise to absolute phase
sd_noise(jj)=unifrnd (0, noise_max, 1, 1);
Gauss_RP = randn(128,128)*sd_noise(jj);
ap_N = ap + Gauss_RP;
```

(f)

```
% get wrapped phase
R = real(exp(1i*ap_N));
I = imag(exp(1i*ap_N));
wp = atan2(I,R);
```

Fig. S2 Core parts of the dataset-generation codes.

All the datasets have been uploaded to the figshare (https://figshare.com/s/685e972475221aa3b4c4), as shown in Fig. S3. The datasets generated by *dataset_generation.m* are as following:

- train_in: The wrapped phase as input of the training dataset is in this folder and named *000001.mat* to *020000.mat*;
- train_gt: The absolute phase as ground truth of the training dataset is in this folder and named *000001.mat* to *020000.mat*;
- test_in: The wrapped phase as input of the testing dataset is in this folder and named *000001.mat* to *002000.mat*;
- test_gt: The absolute phase as ground truth of the testing dataset is in this folder and named *000001.mat* to *002000.mat*.

In addition, we provide anther dataset for testing. It is a noise-free testing dataset of real objects, which includes: candle flames, pits of different arrangements, grooves of different shapes and tables of different shapes:

- test_in_real: The wrapped phase as input of the real testing dataset is in this folder and named *000001.mat* to *000421.mat*;
- test_gt_real: The absolute phase as ground truth of the real testing dataset is in this folder and named *000001.mat* to *000421.mat*;

test_gt
test_gt_real
test_in
test_in_real
train_gt
train_in

Fig. S3 Datasets for phase unwrapping.

S5.2 Neural network making

According to the structure in Fig. S1, we built the neural network in the file *Network.py*, whose core part (residual block) is shown as Fig. S4. The codes of branches in the residual block are shown in Fig. S5.

```python
' Residual block with Inception module '
class ResB(nn.Module):
    def __init__(self, in_ch, out_ch):
        super(ResB, self).__init__()
        self.branch0 = Branch0(in_ch, out_ch)
        self.branch1 = Branch1(in_ch, out_ch // 4)
        self.branch2 = Branch2(in_ch, out_ch // 4)
        self.branch3 = Branch3(in_ch, out_ch // 4)
        self.branch4 = Branch4(in_ch, out_ch // 4)
        self.rl = nn.LeakyReLU(inplace=True)
    def forward(self, x):
        x0 = self.branch0(x)
        x1 = self.branch1(x)
        x2 = self.branch2(x)
        x3 = self.branch3(x)
        x4 = self.branch4(x)
        x5 = torch.cat((x1, x2, x3, x4), dim=1)
        x6 =  x0 + x5        cat
        x7= self.rl(x6)
        return  x7
```

Fig. S4 Codes of the residual block.

```python
' Branch0 block '
class Branch0(nn.Module):
    def __init__(self, in_ch, out_ch):
        super(Branch0, self).__init__()
        self.conv0 = nn.Conv2d(in_ch, out_ch, kernel_size=1, padding=0)
        self.bt0 = nn.BatchNorm2d(out_ch)
    def forward(self, x):
        x0 = self.conv0(x)
        x0 = self.bt0(x0)
        return x0
```

(a)

```python
' Branch1 block '
class Branch1(nn.Module):
    def __init__(self, in_ch, out_ch):
        super(Branch1, self).__init__()
        self.conv1 = nn.Conv2d(in_ch, out_ch, kernel_size=3, padding=1)
        self.bt1 = nn.BatchNorm2d(out_ch)
    def forward(self, x):
        x1 = self.conv1(x)
        x1 = self.bt1(x1)
        return x1
```

(b)

```python
' Branch2 block '
class Branch2(nn.Module):
    def __init__(self, in_ch, out_ch):
        super(Branch2, self).__init__()
        self.conv2_1 = nn.Conv2d(in_ch, out_ch, kernel_size=3, padding=1)
        self.bt2_1 = nn.BatchNorm2d(out_ch)
        self.rl2_1 = nn.LeakyReLU(inplace=True)
        self.conv2_2 = nn.Conv2d(out_ch, out_ch, kernel_size=3, padding=1)
        self.bt2_2 = nn.BatchNorm2d(out_ch)
    def forward(self, x):
        x2 = self.conv2_1(x)
        x2 = self.bt2_1(x2)
        x2 = self.rl2_1(x2)
        x2 = self.conv2_2(x2)
        x2 = self.bt2_2(x2)
        return x2
```

(c)

```python
' Branch3 block '
class Branch3(nn.Module):
    def __init__(self, in_ch, out_ch):
        super(Branch3, self).__init__()
        self.conv3_1 = nn.Conv2d(in_ch, out_ch, kernel_size=3, padding=1)
        self.bt3_1 = nn.BatchNorm2d(out_ch)
        self.rl3_1 = nn.LeakyReLU(inplace=True)
        self.conv3_2 = nn.Conv2d(out_ch, out_ch, kernel_size=3, padding=1)
        self.bt3_2 = nn.BatchNorm2d(out_ch)
        self.rl3_2 = nn.LeakyReLU(inplace=True)
        self.conv3_3 = nn.Conv2d(out_ch, out_ch, kernel_size=3, padding=1)
        self.bt3_3 = nn.BatchNorm2d(out_ch)
    def forward(self, x):
        x3 = self.conv3_1(x)
        x3 = self.bt3_1(x3)
        x3 = self.rl3_1(x3)
        x3 = self.conv3_2(x3)
        x3 = self.bt3_2(x3)
        x3 = self.rl3_2(x3)
        x3 = self.conv3_3(x3)
        x3 = self.bt3_3(x3)
        return x3
```

(d)

```python
' Branch4 block '
class Branch4(nn.Module):
    def __init__(self, in_ch, out_ch):
        super(Branch4, self).__init__()
        self.conv4_1 = nn.Conv2d(in_ch, out_ch, kernel_size=3, padding=1)
        self.bt4_1 = nn.BatchNorm2d(out_ch)
        self.rl4_1 = nn.LeakyReLU(inplace=True)
        self.conv4_2 = nn.Conv2d(out_ch, out_ch, kernel_size=3, padding=1)
        self.bt4_2 = nn.BatchNorm2d(out_ch)
        self.rl4_2 = nn.LeakyReLU(inplace=True)
        self.conv4_3 = nn.Conv2d(out_ch, out_ch, kernel_size=3, padding=1)
        self.bt4_3 = nn.BatchNorm2d(out_ch)
        self.rl4_3 = nn.LeakyReLU(inplace=True)
        self.conv4_4 = nn.Conv2d(out_ch, out_ch, kernel_size=3, padding=1)
        self.bt4_4 = nn.BatchNorm2d(out_ch)
    def forward(self, x):
        x4 = self.conv4_1(x)
        x4 = self.bt4_1(x4)
        x4 = self.rl4_1(x4)
        x4 = self.conv4_2(x4)
        x4 = self.bt4_2(x4)
        x4 = self.rl4_2(x4)
        x4 = self.conv4_3(x4)
        x4 = self.bt4_3(x4)
        x4 = self.rl4_3(x4)
        x4 = self.conv4_4(x4)
        x4 = self.bt4_4(x4)
        return x4
```

(e)

Fig. S5 Codes of branches in the residual block. (a) branch 0; (b) branch 1; (c) branch 2; (d) branch 3; (e) branch 4.

S5.3 Neural network training and testing

Before training and testing the neural network, we need to build a python environment and install the following packages: torch 1.0.1, numpy 1.16.2, tqdm 4.31.1, scipy 1.2.1.

Readers only need to run *main_train.py* to start training the neural network. It should be noted that the corresponding parameters need to be set in Lines 15-22 of *main_train.py* at first, as shown in Fig. S6. During training, information such as progress bar and loss function will be displayed and updated every epoch, as shown in Fig. S7.

After training, two files (*loss and others.csv* and *weights.pth*) will be saved in the folder *model_weights*, as shown in Fig. S8. The former saves the parameters in the training process, such as learning rate, loss function, time-consuming, etc. The latter saves the weights and biases of the trained neural network.

```
12  ' Definition of the needed parameters '
13  def get_args():
14      parser = OptionParser()
15      parser.add_option('-e', '--epochs', dest='epochs', default=100, type='int', help='number of epochs')
16      parser.add_option('-b', '--batch size', dest='batch_size', default=32, type='int', help='batch size')
17      parser.add_option('-l', '--learning rate', dest='lr', default=0.01, type='float', help='learning rate')
18      parser.add_option('-r', '--root', dest='root', default="", help='root directory')
19      parser.add_option('-i', '--input', dest='input', default='train_in', help='folder of input')
20      parser.add_option('-g', '--ground truth', dest='gt', default='train_gt', help='folder of ground truth')
21      parser.add_option('-s', '--model', dest='model', default='model_weights', help='folder for model/weights')
22      parser.add_option('-v', '--val percentage', dest='val_perc', default=0.05 ,type='float', help='validation percentage')
23      (options, args) = parser.parse_args()
24      return options
```

Fig. S6 Parameters for network training.

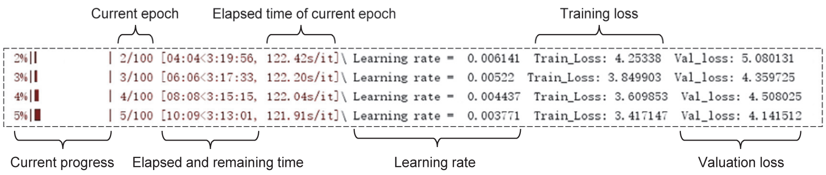

Fig. S7 Training process.

Fig. S8 Files obtained after network training.

By running *main_test.py*, the reader can use the trained neural network to do some test. It should be noted that the corresponding parameters need to be set in Lines 14-18 of *main_test.py* at first, as shown in Fig. S9. So far, the testing results will be saved in the folder *Resultes* in format of *.mat*.

```python
' Definition of the needed parameters '
def get_args():
    parser = OptionParser()
    parser.add_option('-e', '--result', dest='result', default="Results", help='folder of results')
    parser.add_option('-r', '--root', dest='root', default="", help='root directory')
    parser.add_option('-m', '--model', dest='model', default='model_weights/weights.pth', help='folder for model/weights')
    parser.add_option('-i', '--input', dest='input', default="test_in", help='folder of input')
    parser.add_option('-g', '--gt', dest='gt', default="test_gt", help='folder of ground truth')
    (options, args) = parser.parse_args()
    return options
```

Fig. S9 Parameters for network testing.

In addition, there are two files that need to be used during network training and testing, namely *train_func.py* and *dataset_read.py*, which are used for network training and dataset reading, respectively.

Finally, readers can perform error analysis on the testing results in the folder *Resultes* by running *error_evaluation.m*.

S6. A demonstration of dRG phase unwrapping method

We train other neural networks by the datasets with h in the range of [10, 80] and test the trained neural networks by the datasets with h in the range of [10, 90]. As shown in Fig. S10, the height adaptive range of the neural network to h increases from the previous 40 to nearly 80.

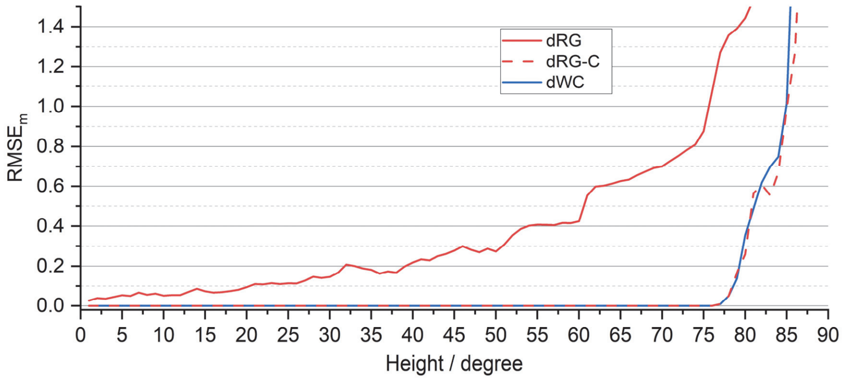

Fig. S10 $RMSE_m$ of the networks for absolute phase in different height.

S7. A demonstration of dDN with wrapped phase denoising

For dDN, we train a network to do denoise directly in wrapped phase. As shown in Fig. S11, the wrapped phase of the neural network has error with the GT at the edges of the wrap, causing severe error propagation for the line-scanning method.

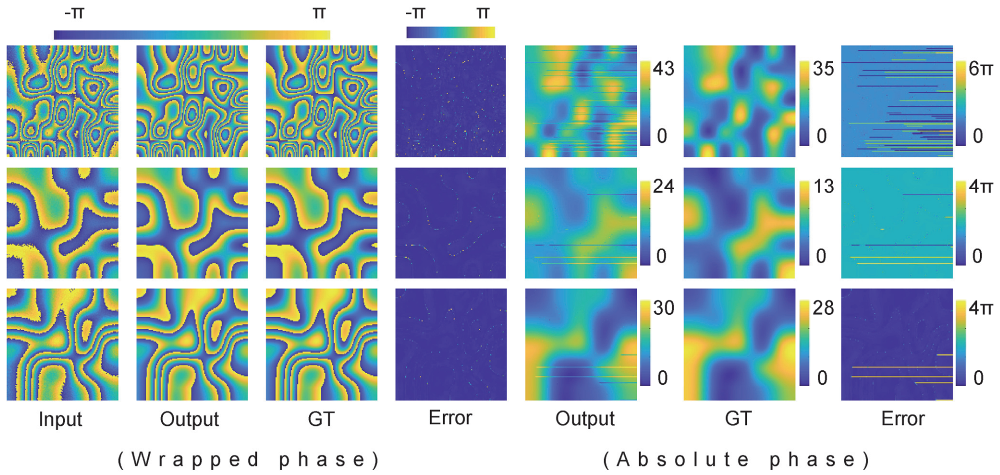

Fig. S11 Results of the dDN with wrapped phase denoising.

S8. Congruence results in different noise level

To verify the effect of congruence on dRG and dDN, we compared the RMSE of its results under different noise levels, and the results are shown in Fig. S12. After congruence, RMSE$_m$ of dRG and dDN decreases to the same level as dWC after congruence.

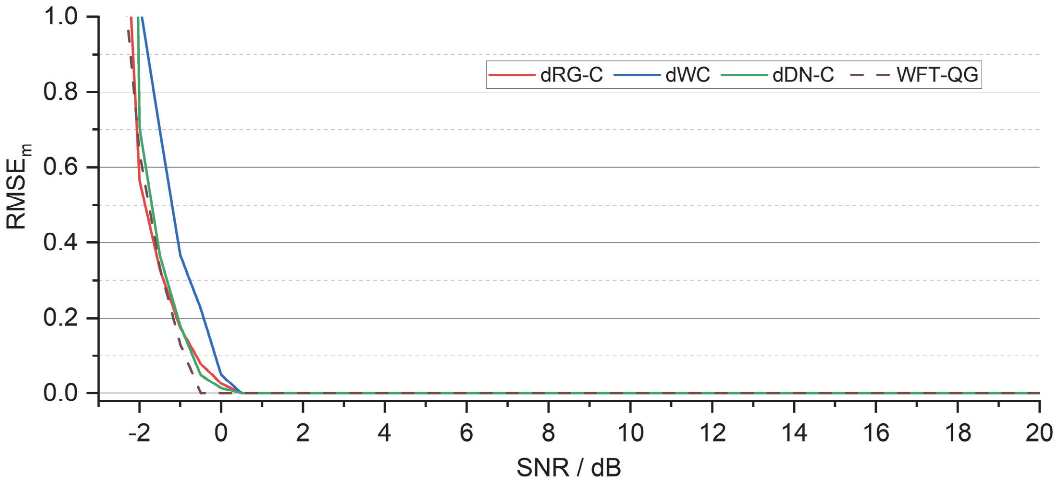

Fig. S12 RMSE$_m$ of dRG-C, dWC, dDN-C and the WFT-QG method in different noise levels. “-C” represents the congruence results.