rfablet / **PB_ANDA**

*No description, website, or topics provided.*

**36** commits | **1** branch | **0** packages | **0** releases | **3** contributors | GPL-3.0

Branch: **master** | New pull request | Create new file | Upload files | Find file | Clone or download

**rfablet** Merge pull request #1 from vietphi3892/master ... Latest commit `0681f53` on 5 Sep 2017

| | | |
|---|---|---|
| Image | Add files via upload | 2 years ago |
| .gitignore | Initial commit | 2 years ago |
| AnDA_Multiscale_Assimilation.py | Update AnDA_Multiscale_Assimilation.py | 2 years ago |
| AnDA_analog_forecasting.py | Update AnDA_analog_forecasting.py | 2 years ago |
| AnDA_data_assimilation.py | Add files via upload | 2 years ago |
| AnDA_stat_functions.py | Add files via upload | 2 years ago |
| AnDA_transform_functions.py | Update AnDA_transform_functions.py | 2 years ago |
| AnDA_variables.py | Update AnDA_variables.py | 2 years ago |
| LICENSE | Initial commit | 2 years ago |
| README.md | Update README.md | 2 years ago |
| test_AnDA_SLA.py | Update test_AnDA_SLA.py | 2 years ago |
| test_AnDA_SST.py | Update test_AnDA_SST.py | 2 years ago |

**README.md**

# DESCRIPTION

PB-MS-AnDA is a Python library for Patch-Based Multi-scale Analog Data Assimilation, applications to ocean remote sensing. We presented a novel data-driven model for the spatio-temporal interpolation of satellite-derived geophysical fields fields, an extension of analog data assimilation framework (https://github.com/ptandeo/AnDA) to high-dimensional satellite-derived geophysical fields.

This Python library is an additional material of the publication "Data-driven Models for the Spatio-Temporal Interpolation of satellite-derived SST Fields", from **R. Fablet, P. Huynh Viet, R. Lguensat**, accepted to *IEEE Transactions on Computational Imaging*

# Basic Overview

The toolbox includes 3 main modules:

1. Module **Parameters** (*AnDA_variables.py*):

- Class **PR**: to specify general parameters
  - Use multi-scale or single-scale (global-scale) assimilation ?
  - Dimension of state vector (or reduced dimensionality in PCA space)
  - Size of patch (eg. 20 × 20)
  - Size of training dataset, testing dataset (number of images)
  - Directories of datasets: sst (sla), observation, OI product (ostia)...

```
# Example of setting parameter for SST
 PR_ = PR()
 PR_.flag_scale = True  # True: multi scale AnDA, False: global scale AnDA
 PR_.n = 50 # dimension state vector
 PR_.patch_r = 20 # r_size of patch
 PR_.patch_c = 20 # c_size of patch
 PR_.training_days = 2558 # num of training images: 2008-2014
 PR_.test_days = 364 # num of test images: 2015
 PR_.lag = 1 # lag of time series: t -> t+lag
 PR_.G_PCA = 20 # N_eof for global PCA
 # Input dataset (format should be NETCDF (.nc))
 PR_.path_X = './data/AMSRE/sst.nc' # directory of sst data
 PR_.path_OI = './data/AMSRE/OI.nc' # directory of OI product (ostia sst, in this case)
 PR_.path_mask = './AMSRE/metop_mask.nc' # directory of observation mask
 # Dataset automatically created during execution
 PR_.path_X_lr = './data/AMSRE/sst_lr.nc' # directory of LR product
 PR_.path_dX_PCA = './data/AMSRE/dX_pca.nc' # directory of PCA transformation of detail fields
 PR_.path_index_patches = './data/AMSRE/list_pos.pickle' # directory to store all position of each
 PR_.path_neighbor_patches = './data/AMSRE/pair_pos.pickle' # directory to store position of each
```

- Class **VAR**: to store all necessary datasets
  - Training and testing catalog for detail fields in both original and EOF space
  - Observation
  - LR product
  - Condition dataset used in AF (if exists)
  - Indexing set that points out the position of a patch over original image

```
# Program will automatically load all data into this variable according the parameters described i
class VAR:
     X_lr = []
     dX_orig = []
     Optimal_itrp = []
     dX_train = [] # training catalogs  for dX in EOF space
     dX_eof_coeff = [] # EOF base vector
     dX_eof_mu = [] # EOF mean vector
     dX_GT_test = [] # dX GT in test year
     Obs_test = [] # Observation in test year, by applying mask to dX GT
     dX_cond = [] # condition used for AF
     gr_vl_train = [] # gradient, velocity used as physical condition
     gr_vl_test = {}
     gr_vl_coeff = {}
     index_patch = [] # store order of every image patch: 0, 1,..total_patchs
     neighbor_patchs = [] # store order of neighbors of every image patch
```

- Class **General_AF**: to specify parameters for Analog Forecasting
  - Use condition for analog forecasting ?. If using condition, specify where is the condition
  - Use clusterized version ?. If using, specify number of *k* clusters
  - Use global or local analog by specifying form of neighborhood
  - Select three forecasting strategies: locally constant, increment, local linear
  - Variance of initial error, observation error
  - Pre-trained nearest neighbor searchers ( FLANN )

```
# Example of Analog Forecasting for SST
AF_ = General_AF()
AF_.flag_reduced  = True # True: Clusterized version of Local Linear AF
AF_.flag_cond = False # True: use Obs at t+lag as condition to select successors
                  # False: no condition in analog forecasting
AF_.flag_model = False # True: Use gradient, velocity as additional regressors in AF
AF_.flag_catalog = True # True: Use each catalog for each patch position
                    # False: Use only one big catalog for all positions
```

```
AF_.cluster = 1     # number of cluster for clusterized ver.
AF_.k = 200  # number of analogs
AF_.k_initial = 200 # retrieving k_initial nearest neighbors, then using condition to retrieve k a
AF_.neighborhood = np.ones([PR_.n,PR_.n]) # global analogs
AF_.neighborhood = np.eye(PR_.n)+np.diag(np.ones(PR_.n-1),1)+ np.diag(np.ones(PR_.n-1),-1)+ \
                       np.diag(np.ones(PR_.n-2),2)+np.diag(np.ones(PR_.n-2),-2)
AF_.neighborhood[0:2,:5] = 1
AF_.neighborhood[PR_.n-2:,PR_.n-5:] = 1 # local analogs
AF_.neighborhood[PR_.n-2:,PR_.n-5:] = 1 # local analogs
AF_.regression = 'local_linear' # forecasting strategies. select among: locally_constant, incremen
AF_.sampling = 'gaussian'
AF_.B = 0.05 # variance of initial state error
AF_.R = 0.1  # variance of observation error
```

- Class **AnDA_result**: store AnDA's results, such as GT, Observation, Optimal Interpolation, AnDA Interpolations and statistical errors (rmse, correlation)

```
# All results will be computed and stored in this class.
class AnDA_result:
     itrp_AnDA = [] # AnDA interpolation
     itrp_OI = []   # OI product, for comparison
     itrp_postAnDA = []  # Post_processing AnDA interpolation (removing block artifacts)
     GT = []   # groundtruth
     Obs = []  # Observation
     LR = []   # Low resolution product
     # stats: rmse & correlation of interpolation to the groundtruth
     rmse_AnDA = []
     corr_AnDA = []
     rmse_OI = []
     corr_OI = []
     rmse_postAnDA = []
     corr_postAnDA = []
```

2. Module **Transform functions** (*AnDa_transform_functions.py*):
   - Perform Global PCA (to find LR), patch-based PCA for multi-scale assimilation
   - Post-processing to remove block artifact due to overlapping patches
   - Perform VE-DINEOF
   - Find gradient, Fourier power spectrum
   - Loading and preprocessing data according to the parameters described in **PR**
3. Module **Multi-scale Assimilation** (*Multiscale_Assimilation.py*): based on informations from PR, VAR, AF, defining a specific kind of assimilation
   - Class **Single_patch_assimilation**:
     - Processing on one single patch.
     - Input: position of patch (rows, columns) over initial image.
   - Class **Multi_patch_assimilation**:
     - Processing on a zone of image (defined by its size and coordinates of top-left point), by dividing into multiples patches, then plugging them into **Single_patch_assimilation**
     - Input: number of parallel jobs, or number of patches are executed simultaneously.

# Test

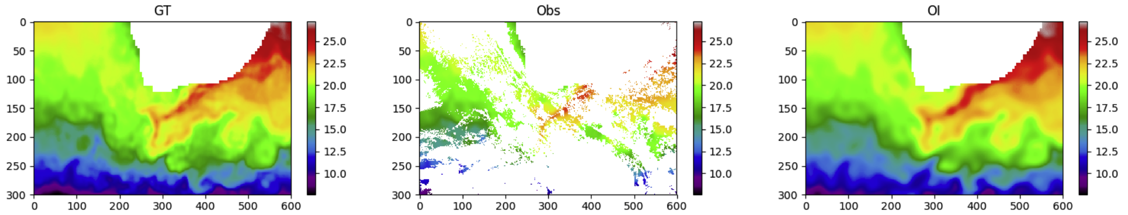
Specify all necessary parameters described in class **PR**, and **General_AF**.
Load data into class **VAR**:

```
VAR_ = VAR()
VAR_ = Load_data(PR_)
```

Visualize an example of reference Groundtruth (GT), Observation (Obs) and Low resolution Optimal Interpolation (OI) product

```
day = 50
colormap='nipy_spectral'
plt.clf()
```

```
gt = VAR_.dX_GT_test[day,:,:]
obs = VAR_.Obs_test[day,:,:]
itrp = VAR_.Optimal_itrp[day,:,:]
vmin = np.nanmin(gt)
vmax = np.nanmax(gt)
plt.subplot(1,3,1)
plt.imshow(gt,aspect='auto',cmap=colormap,vmin=vmin,vmax=vmax)
plt.colorbar()
plt.title('GT')
plt.subplot(1,3,2)
plt.imshow(obs,aspect='auto',cmap=colormap,vmin=vmin,vmax=vmax)
plt.colorbar()
plt.title('Obs')
plt.subplot(1,3,3)
plt.imshow(itrp,aspect='auto',cmap=colormap,vmin=vmin,vmax=vmax)
plt.colorbar()
plt.title('OI')
plt.draw()
```




Define test zone (top-left point and size of zone) (note: must 4 values must be divisible by 5):

```
r_start = 0
c_start = 0
r_length = 150
c_length = 300
```

Define multiprocessing level:

```
level = 22 # 22 patches executed simultaneously
```

Run Assimilation:

```
saved_path =  'path_to_save.pickle'
MS_AnDA_itrp = AnDA_result()
MS_AnDA_ = MS_AnDA(VAR_sst, PR_sst, AF_sst)
MS_AnDA_itrp = MS_AnDA_sst.multi_patches_assimilation(level, r_start, r_length, c_start, c_length)
```

Save result:

```
with open(saved_path, 'wb') as handle:
    pickle.dump(MS_AnDA_itrp, handle)
```

Reload result: Save result:

```
with open(saved_path, 'rb') as handle:
    MS_AnDA_itrp = pickle.load(handle)
```

To compare with AnDA interpolation:

- Run VE-DINEOF algorithms to compare with AnDA interpolation.

  ```
  itrp_dineof = VE_Dineof(PR_, VAR_.dX_orig+VAR_.X_lr, VAR_.Optimal_itrp+VAR_.X_lr[PR_.training_days:],
  ```

- Run G-AnDA: applying AnDA on region scale. We need to reset parameters in **PR** and **General_AF**:

  ```
  PR_.flag_scale = False  # True: multi scale AnDA, False: global scale AnDA
  ```

```
PR_.n = 200 # choose higher than the one from local scale, because we want to keep 99% variance afte
PR_.patch_r = 200 # r_size of image
PR_.patch_c = 120 # c_size of image
AF_.flag_reduced  = False or True
AF_.flag_cond = False
AF_.flag_model = False
AF_.flag_catalog = False
AF_.cluster = 1     # number of cluster for clusterized ver.
AF_.k = 500  # number of analogs, should be higher than state vector's dimension
AF_.k_initial = 500 # retrieving k_initial nearest neighbors, then using condition to retrieve k ana
AF_.neighborhood = np.ones([PR_.n,PR_.n]) # global analogs
```

Then reload data (because we now assimilate high resolution (original) fields, not detail fields):

```
VAR_ = VAR()
VAR_ = Load_data(PR_)
```

Then run single patch assimilation (this case isn't patch-based):

```
saved_path =  'path_to_save.pickle'
itrp_G_AnDA = AnDA_result()
MS_AnDA_ = MS_AnDA(VAR_, PR_, AF_)
itrp_G_AnDA = MS_AnDA_.single_patch_assimilation([np.arange(r_start,r_start+r_length),np.arange(c_star
```

Display interpolation performance & Fourier power spectrum (**note** that the input of *raPsd2dv1* should be without land pixel (avoid NaN values).

```
day =11 # 82
res_ = 0.25
f0, Pf_  = raPsd2dv1(itrp_G_AnDA[day,:,:],resSLA,True)
wf1        = 1/f1
plt.figure()
plt.loglog(wf1,Pf_AnDA,label='G-AnDA')
plt.gca().invert_xaxis()
plt.legend()
plt.xlabel('Wavelength (km)')
plt.ylabel('Fourier power spectrum')
```