# Questions

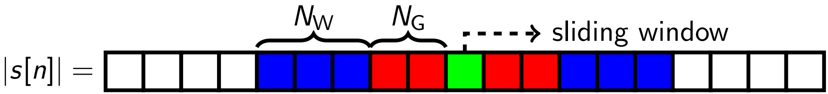
Any questions to exercises and homeworks from last time?

# Table of Content

8 CFAR

- The CA- and CAGO-CFAR Algorithms
- Numba
  - Python vs. C
  - Function Decoration

# CFAR Algorithm Review



🟩 Cell under test

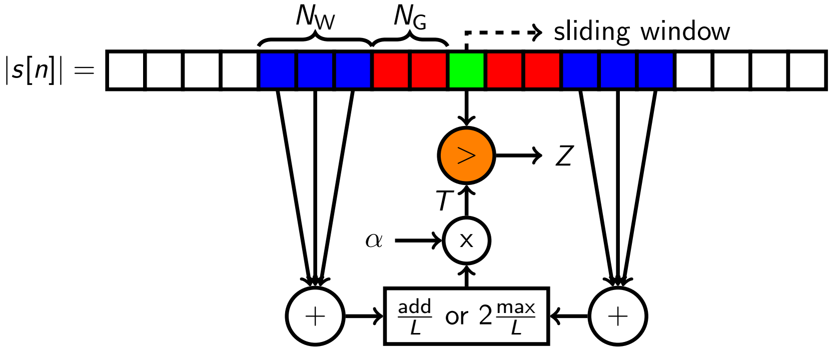
🟥 Guard cells

🟦 Calculation cells

⬜ Other cells

- Every cell becomes the cell under test once.
- Guard cells account for width of PSF.
- A statistic like CA and OSGO is derived from compute cells.
- Scaling to obtain desired probability of false alarm.
- Other cells indicate length of sequence.

# Cell Average CFAR



$$\alpha = \sqrt{\frac{4}{\pi} L \left( P_{\text{fa}}^{-\frac{1}{L}} - 1 \right) \cdot \left( 1 - \left( 1 - \frac{\pi}{4} \right) \exp(1 - L) \right)} \qquad L = 2N_W$$

# Exercise: CFAR

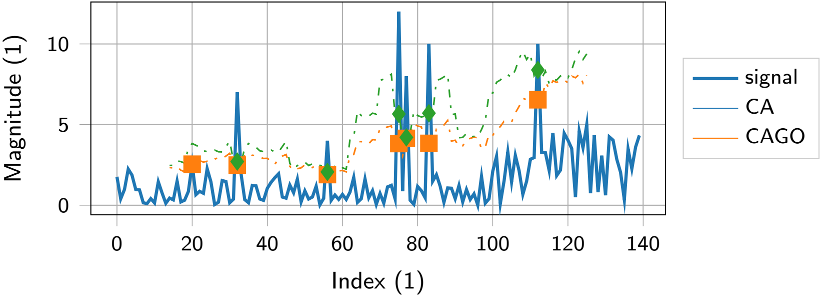
- Implement a function to calculate CA(GO)-CFAR levels.

```python
def cfar_thresh_lvl(x, n_width, Pfa=None, n_guard=None, mode=None)
    """
    Calculates the threshold level for a signal x with a CFAR
    method given by mode, by looping over all cells.
    Parameters
    ----------
    x: array
        Array of positive (absolute values) of floats
    n_width: int
        One-sided width of the window.
    Pfa: float, optional
        False alarm rate. Default: 1e-4
    n_guard: int, optional
        One sided number of guard cells. Default: no guard cells
    mode: string, optional
        'CA' or None for cell average, 'CAGO' for CA-greatest of
    Returns
    -------
    array of size of x holding threshold levels.
    """
```

# Exercise: Test CFAR

- Test `cfar_thresh_lvl` on the following sequence and parameters:

```
import numpy as np
x=np.concatenate((np.random.randn(100),5*np.random.rand(40)))
x=np.abs(n) # Rayleigh distributed
x[[32,56,75,77,83,112]]=[7,4,12,8,10,10] # setting multiple elements
Pfa=1e-2      # probability of false-alarm
n_width=12    # single-sided width of the window
n_guard=1     # single-sided number of guard cells
```



# Python vs. C

- Python is a high-level programming language.
- Versatile, general, easy to use. Example: everything is a class.
- Thus: not too fast; especially indexing and looping is very slow compared to C.
- Many methods to include C-code in python, e.g.
  - system calls to auxiliary executable using `os.system`,
  - calls to shared libraries (*.dll or *.so) using `ctypes`,
  - using Python functions/classes/modules directly C-code via `Python.h` (similar to Matlab's mex files),
  - include independently written C-code using `cffi` in Python scripts, and
  - compiling python code.

# Numba's JIT as Function Decorator

- Idea: convert python code to C and compile it.
- Function decorator allows to modify a function without changing its code.

```
from numba import jit

@jit(nopython=True, cache=True, parallel=True)
def my_slow_function(a,b,c=None):
    if c None:
        return a+b
    else:
        return a+b+c
```

- Compiled version is only valid for a certain set of parameters (types and array shapes).
- First execution is slow, since compilation takes few seconds.
- Precompilation upon `import` is possible (if types and shapes were known), but syntax is tricky.
- Using `nopython=True` may speed up coded by orders of magnitude instead of a few percent.

# Common Limitation of Numba's `nopython`

- Comparing inequal types with `==`, e.g., `None` with a string, fails (might be a bug). Thus santanize inputs!

  - Failing with `nopython=True`:

```python
def test_string(a, mode=None):

  if mode is None or mode=='CA':
    return a
  else:
    return 2*a
```

  - Working with `nopython=True`:

```python
def test_string(a, mode=None):
  if mode is None: mode='CA'
  if mode=='CA':
    return a
  else:
    return 2*a
```

- `numba` support a wide range of `numpy` features, but not all, and not all optional arguments.

  http://numba.pydata.org/numba-doc/dev/reference/numpysupported.html

# Timeit and Numba

- numba's jit can be used directly too, to have both, the wrapped/compiled as well as the Python version.
- Result should be numerically the same.

```
from numba import jit
...
# add wrapper
c=jit(cfar_thresh_lvl, nopython=True, cache=True)
# compare result
t1=c(n, n_width, Pfa)                     # call to wrapped function
t2=cfar_thresh_lvl(n, n_width, Pfa)       # call to original function
np.allclose(t1, t2)                       # True
```

- timeit can assess execution time of a function call.

```
%timeit c(n, n_width, Pfa)                # magic % from ipython
%timeit cfar_thresh_lvl(n, n_width ,Pfa) # magic % from ipython
```

- Intermediate code can be inspected after call/compilation.

```
a=c.inspect_types()                       # returns a dictionary
for b in a: print(a[b])                   # print elements of dict
```

## Homework: Numba

- Add numba's jit decorator to produce compiled code.
- Use timeit to show and document the effect of compilation on execution time.