(Hopefully) Efficient Numerical Estimation of Optical Flow

Thorbjørn Djupvik, Rudolfs Sietinsons & Ivan Nygaard

Problem Statement (part I)

$$I : \Omega \times \mathbb{R} \to \mathbb{R}$$ ← Video (Intensity)

$$\boldsymbol{w}(x, y) := [u(x, y), v(x, y)]$$ ← Optical flow-field (what we want)

$$I(x(t_0 + \Delta t), y(t_0 + \Delta t), t_0 + \Delta t) = I(x(t_0), y(t_0), t_0)$$ ← Brightness consistency

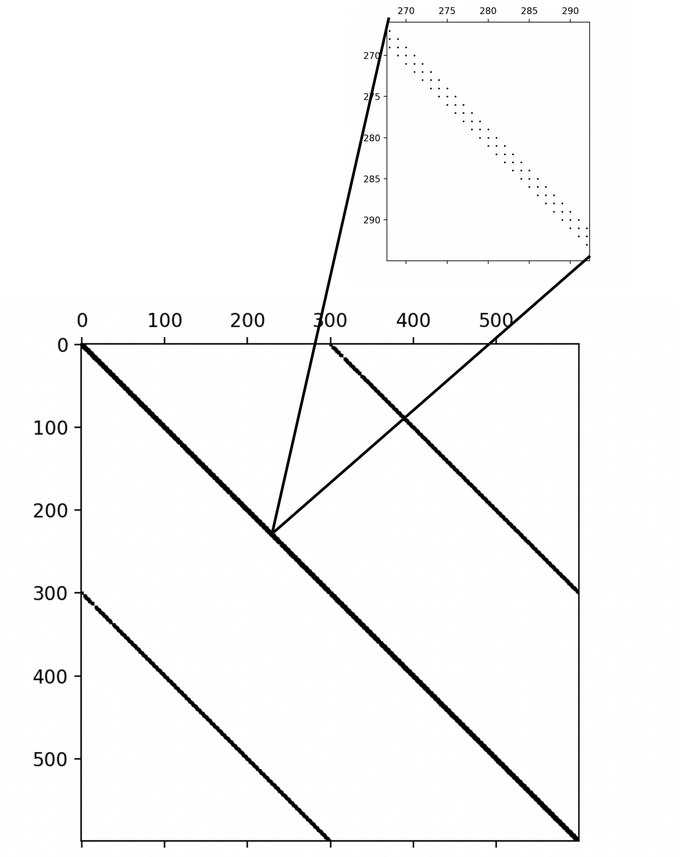
$$u\partial_x I + v\partial_y I = -\partial_t I$$ ← PDE

$$\left[(\partial_x I)_{ij}^2 + \frac{4\lambda}{h^2}\right] u_{ij} - \frac{\lambda}{h^2}\left[u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}\right] + (\partial_y I)_{ij}(\partial_x I)_{ij} v_{ij} = -(\partial_t I)_{ij}(\partial_x I)_{ij},$$

$$\left[(\partial_y I)_{ij}^2 + \frac{4\lambda}{h^2}\right] v_{ij} - \frac{\lambda}{h^2}\left[v_{i+1,j} + v_{i-1,j} + v_{i,j+1} + v_{i,j-1}\right] + (\partial_y I)_{ij}(\partial_x I)_{ij} u_{ij} = -(\partial_t I)_{ij}(\partial_y I)_{ij}.$$

← Discretized equations

Problem Statement (part II)



$(2mn) \times (2mn)$

u11		rhsu11
u12		rhsu12
u13		rhsu13
...	=	...
v11		rhsv11
v12		rhsv12
v13		rhsv13
...		...

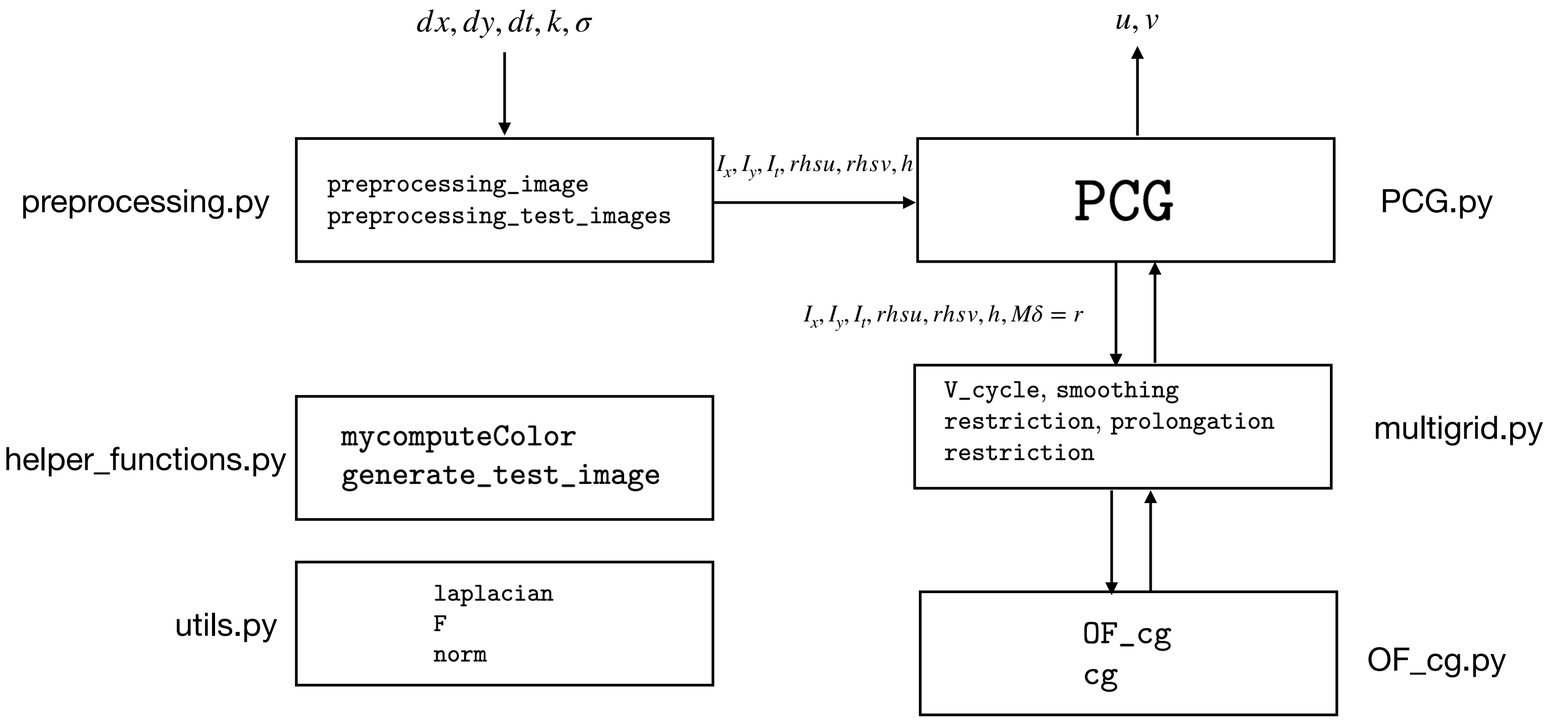
$(2mn)$ $(2mn)$

$$\underbrace{\begin{bmatrix} \boldsymbol{A}_{11} & \boldsymbol{A}_{12} \\ \boldsymbol{A}_{21} & \boldsymbol{A}_{22} \end{bmatrix}}_{\boldsymbol{A}} \underbrace{\begin{bmatrix} \boldsymbol{u} \\ \boldsymbol{v} \end{bmatrix}}_{\boldsymbol{x}} = \underbrace{\begin{bmatrix} \boldsymbol{b}_1 \\ \boldsymbol{b}_2 \end{bmatrix}}_{\boldsymbol{b}}$$

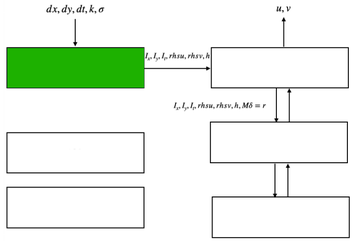
$$\left.\begin{aligned} \left[(\partial_x I)_{ij}^2 + \frac{4\lambda}{h^2}\right] u_{ij} - \frac{\lambda}{h^2}\left[u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}\right] + (\partial_y I)_{ij}(\partial_x I)_{ij} v_{ij} &= -(\partial_t I)_{ij}(\partial_x I)_{ij}, \\ \left[(\partial_y I)_{ij}^2 + \frac{4\lambda}{h^2}\right] v_{ij} - \frac{\lambda}{h^2}\left[v_{i+1,j} + v_{i-1,j} + v_{i,j+1} + v_{i,j-1}\right] + (\partial_y I)_{ij}(\partial_x I)_{ij} u_{ij} &= -(\partial_t I)_{ij}(\partial_y I)_{ij}. \end{aligned}\right\}$$

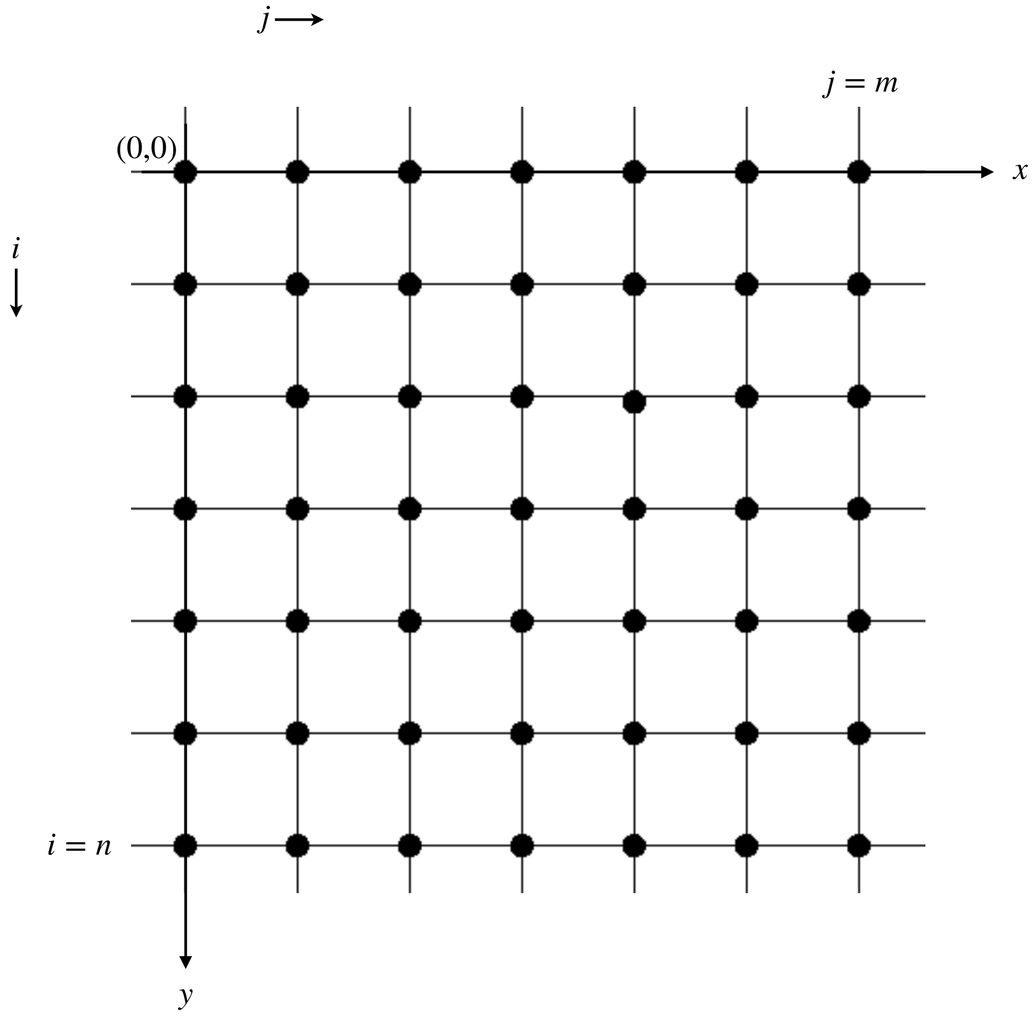
The matrix A can be proven to be SPD, this has some important consequences for the implementation.

Workflow


dx, dy, dt, k, σ
u, v
preprocessing.py
preprocessing_image
preprocessing_test_images
$I_x, I_y, I_t, rhsu, rhsv, h$
PCG
PCG.py
$I_x, I_y, I_t, rhsu, rhsv, h, M\delta = r$
V_cycle, smoothing
restriction, prolongation
restriction
multigrid.py
helper_functions.py
mycomputeColor
generate_test_image
utils.py
laplacian
F
norm
OF_cg
cg
OF_cg.py

preprocessing.py





$$\partial_x I_k(x_i, y_j) := \begin{cases} I_k(x_{i+1}, y_j) - I_k(x_i, y_j), & \text{if } i < m, \\ I_k(x_m, y_j) - I_k(x_{m-1}, y_j), & \text{if } i = m. \end{cases}$$

I0x[:, :-1] = (I0[:, 1:] - I0[:, :-1]) / dx

← Frwrd. Diff.

$$\partial_x I_k(x_i, y_j) := \begin{cases} I_k(x_i, y_{j+1}) - I_k(x_i, y_j), & \text{if } j < n, \\ I_k(x_i, y_n) - I_k(x_i, y_{n-1}), & \text{if } j = n. \end{cases}$$

I0x[:, -1] = (I0[:, -1] - I0[:, -2]) / dx

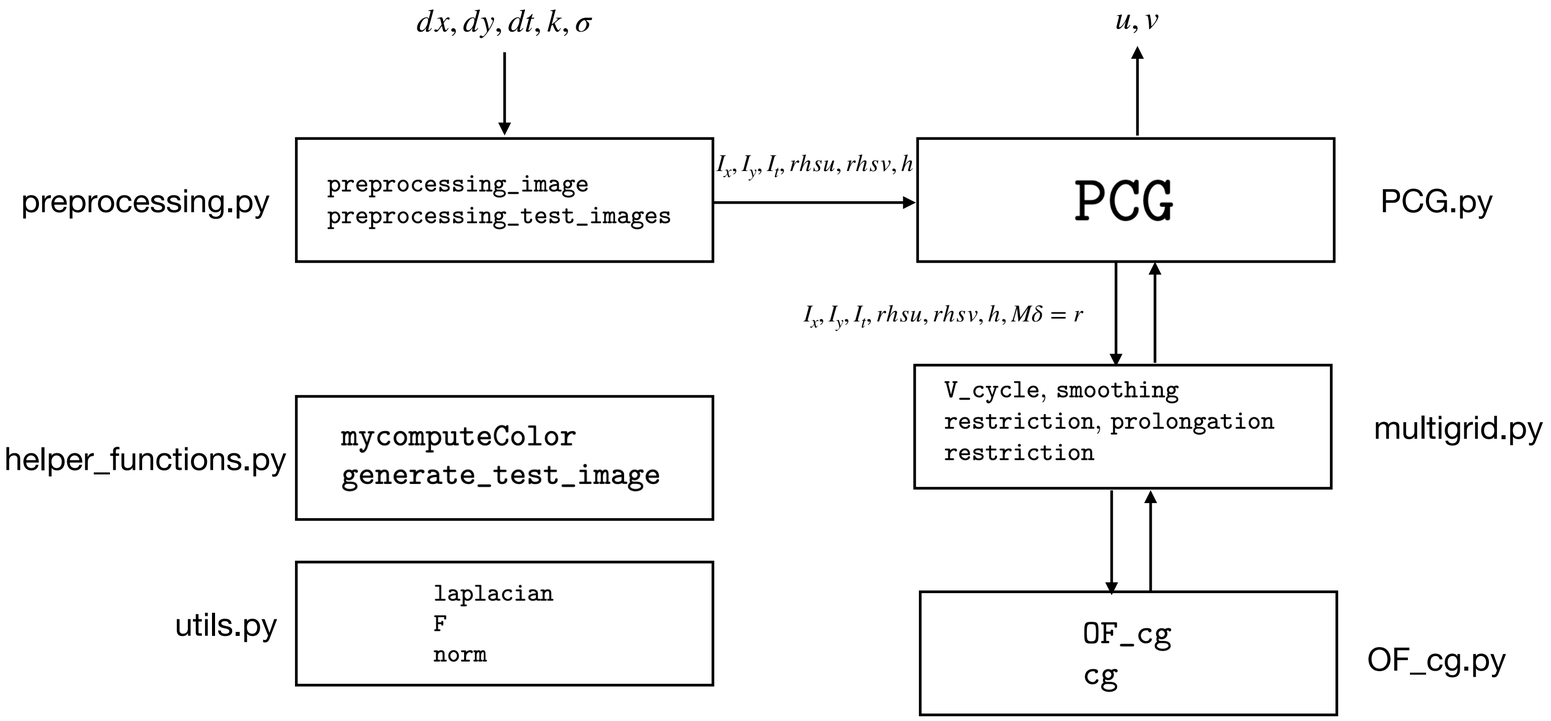
← Bcwrd. Diff.

$$\partial_t I(x_i, y_j) := I_1(x_i, y_j) - I_0(x_i, y_j)$$

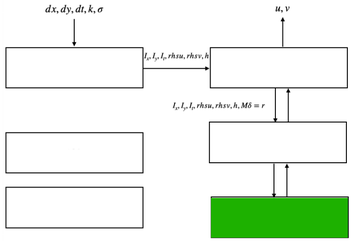
It = (I1 - I0) / dt

← Frwrd. Diff
«Frame difference»

Workflow


dx, dy, dt, k, σ
u, v
preprocessing.py
preprocessing_image
preprocessing_test_images
$I_x, I_y, I_t, rhsu, rhsv, h$
PCG
PCG.py
$I_x, I_y, I_t, rhsu, rhsv, h, M\delta = r$
V_cycle, smoothing
restriction, prolongation
restriction
multigrid.py
helper_functions.py
mycomputeColor
generate_test_image
utils.py
laplacian
F
norm
OF_cg
cg
OF_cg.py

OF_cg.py



Algorithm 1 CG

Compute $r_0 := b - Ax_0$, $p_0 := r_0$
for $j = 0, 1, \ldots,$ until convergence **do**
  $\alpha_j \leftarrow \frac{(r_j, r_j)}{(Ap_j, p_j)}$
  $x_{j+1} \leftarrow x_j + \alpha_j p_j$
  $r_{j+1} \leftarrow r_j - \alpha_j A p_j$
  $\beta_j \leftarrow \frac{(r_{j+1}, r_{j+1})}{(r_j, r_j)}$
  $p_{j+1} \leftarrow r_{j+1} + \beta_j p_j$
end for

Two implementations, one sparse and one that works directly on the grid.

```
A_11 = sp.diags(Ix.ravel() ** 2) + L
A_22 = sp.diags(Iy.ravel() ** 2) + L
A_12 = sp.diags(Ixy.ravel())
A_21 = A_12.copy()

A = sp.block_array([[A_11, A_12], [A_21, A_22]]).tocsr()
```

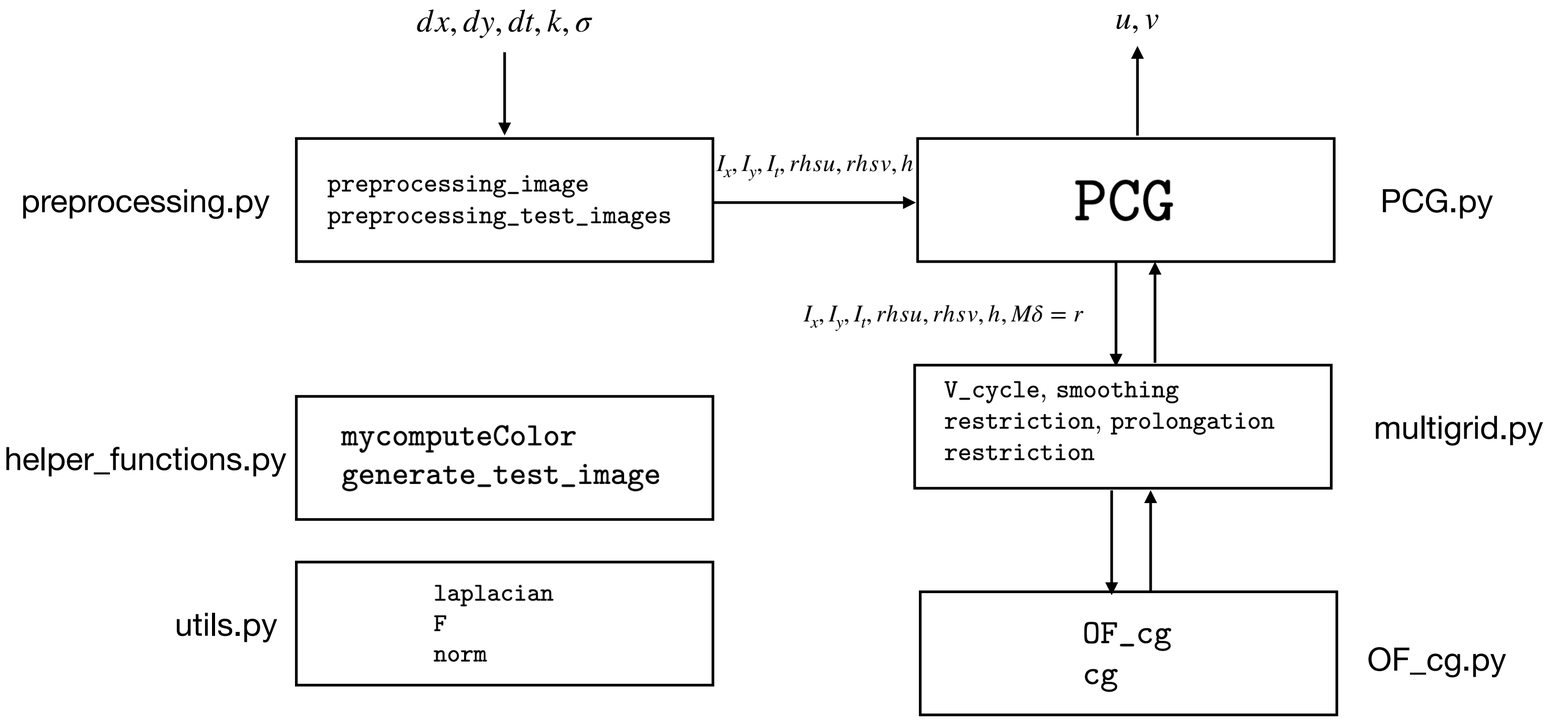
← Sparse CG

```
# Residuals (For experiments)
relative_residuals_arr = np.zeros(maxitr)

# Initialize
Fu, Fv = F(u0, v0, Ix, Iy, lam, h)

# r
ru = np.copy(rhs_u - Fu)
rv = np.copy(rhs_v - Fv)
# x
u = np.copy(u0)
v = np.copy(v0)
# p
pu = np.copy(ru)
pv = np.copy(rv)

# Calculate the norm of r
rr0 = norm(ru, rv) ** 2
rk1_rk1 = rr0
```

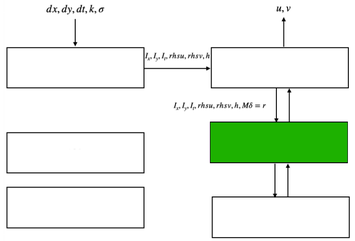
```
def F(
    u: np.ndarray,
    v: np.ndarray,
    Ix: np.ndarray,
    Iy: np.ndarray,
    lam: float,
    h: float,
) -> tuple[np.ndarray, np.ndarray]:
    Ixy = Ix*Iy
    Fu = Ix**2 * u + Ixy * v - lam * laplacian(u, h)
    Fv = Iy**2 * v + Ixy * u - lam * laplacian(v, h)
    return Fu, Fv
```

← Grid CG

Workflow


dx, dy, dt, k, σ
u, v
preprocessing.py
preprocessing_image
preprocessing_test_images
$I_x, I_y, I_t, rhsu, rhsv, h$
PCG
PCG.py
$I_x, I_y, I_t, rhsu, rhsv, h, M\delta = r$
V_cycle, smoothing
restriction, prolongation
restriction
multigrid.py
helper_functions.py
mycomputeColor
generate_test_image
utils.py
laplacian
F
norm
OF_cg
cg
OF_cg.py

Multigrid V-cycle (part I)

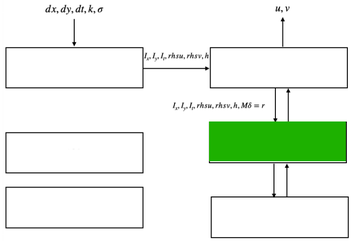


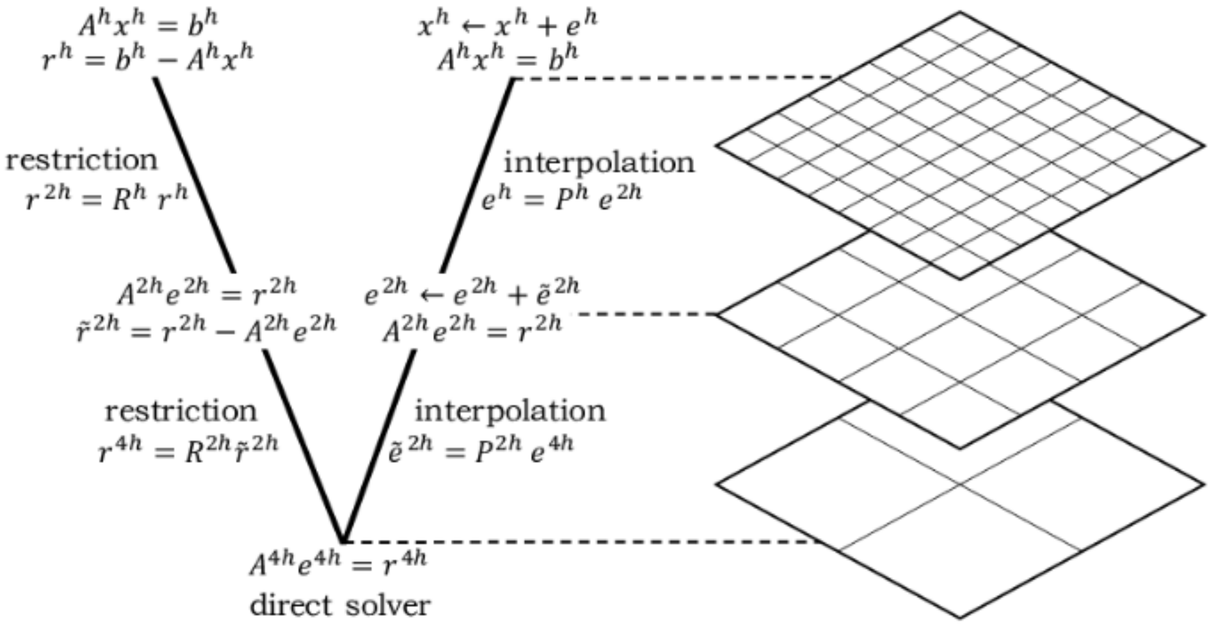

```python
def V_cycle(
    u0: np.ndarray,
    v0: np.ndarray,
    Ix: np.ndarray,
    Iy: np.ndarray,
    lam: float,
    rhs_u: np.ndarray,
    rhs_v: np.ndarray,
    s1: int,
    s2: int,
    level: int,
    max_level: int,
) -> tuple[np.ndarray, np.ndarray]:
    """
    V-cycle for the optical flow problem

    Args:
    ---
    u0 : np.ndarray
        Initial guess for u
    v0 : np.ndarray
        Initial guess for v
    Ix : np.ndarray
        x-derivative of the first frame
    Iy : np.ndarray
        y-derivative of the first frame
    lam : float
        Penalty term
    rhs_u : np.ndarray
        RHS in eq for u
    rhs_v : np.ndarray
        RHS in eq for v
    s1 : int
        Number of pre-smoothings
    s2 : int
        Number of post-smoothings
    level : int
        Current level
    max_level : int
        Total number of levels

    Returns:
    ---
    tuple[np.ndarray, np.ndarray, list[float]]
        Numerical solution for u, v
    """
```

← V-cycle routine

The method is implemented recursively and calls the sub-routines red_update, black_update, smoothing, prolongation, restriction and interpolation.

Multigrid V-cycle (part II)





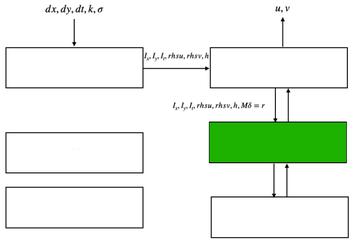

```python
assert ru_h.shape == rv_h.shape == Ix.shape == Iy.shape

n, m = ru_h.shape

ru_2h = (
    ru_h[0 : n - 1 : 2, 0 : m - 1 : 2]  # upper left
    + ru_h[1:n:2, 0 : m - 1 : 2]  # lower left
    + ru_h[0 : n - 1 : 2, 1:m:2]  # upper right
    + ru_h[1:n:2, 1:m:2]  # lower right
) / 4
rv_2h = (
    rv_h[0 : n - 1 : 2, 0 : m - 1 : 2]  # upper left
    + rv_h[1:n:2, 0 : m - 1 : 2]  # lower left
    + rv_h[0 : n - 1 : 2, 1:m:2]  # upper right
    + rv_h[1:n:2, 1:m:2]  # lower right
) / 4

Ix2h = (
    Ix[0 : n - 1 : 2, 0 : m - 1 : 2]  # upper left
    + Ix[1:n:2, 0 : m - 1 : 2]  # lower left
    + Ix[0 : n - 1 : 2, 1:m:2]  # upper right
    + Ix[1:n:2, 1:m:2]  # lower right
) / 4

Iy2h = (
    Iy[0 : n - 1 : 2, 0 : m - 1 : 2]  # upper left
    + Iy[1:n:2, 0 : m - 1 : 2]  # lower left
    + Iy[0 : n - 1 : 2, 1:m:2]  # upper right
    + Iy[1:n:2, 1:m:2]  # lower right
) / 4
```

← Restriction

```python
n, m = e_2h.shape

e_h = np.zeros((2 * n, 2 * m))
k, d = e_h.shape

# Upper left
e_h[0 : k - 1 : 2, 0 : d - 1 : 2] = e_2h
# Lower left
e_h[1:k:2, 0 : d - 1 : 2] = e_2h
# Upper right
e_h[0 : k - 1 : 2, 1:d:2] = e_2h
# Lower right
e_h[1:k:2, 1:d:2] = e_2h

return e_h
```

← Prolongation

Multigrid V-cycle (part III)



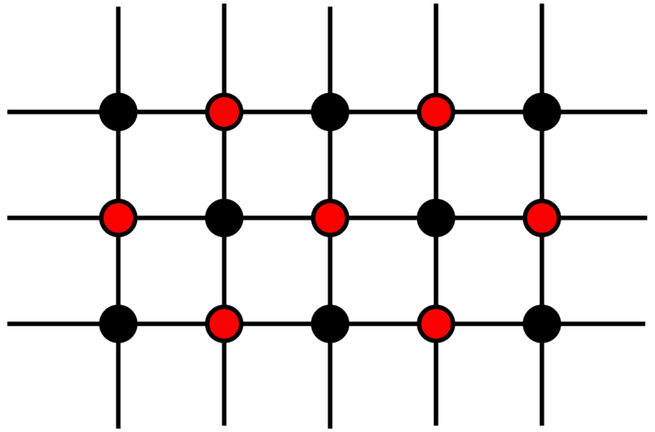
Algorithm 3 Red-Black

1: Red-Update(u)
2: Black-Update(v)
3: Red-Update(v)
4: Black-Update(u)

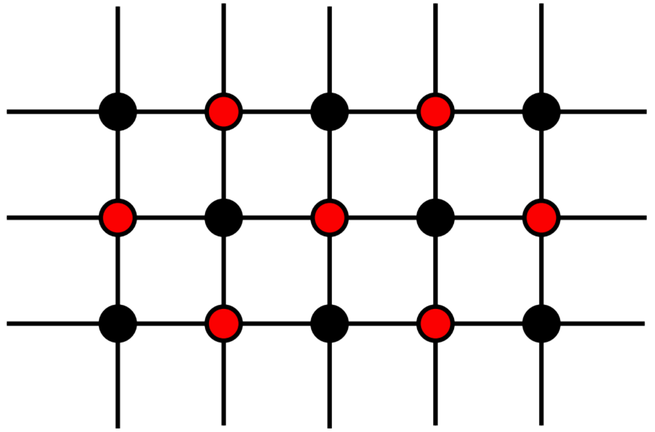
```python
def red_update(
    w: np.ndarray,
    p: np.ndarray,
    Iw: np.ndarray,
    Ip: np.ndarray,
    lam: float,
    rhs: np.ndarray,
    h: float,
) -> None:
    """
    Red update using Red-Black Gauss-Seidel
    Note: This function mutates w

    Args:
    ---
    w : np.ndarray
        The padded array to update
    p : np.ndarray
        The padded other array
    Iw : np.ndarray
        If w=u -> Iw=Ix
        If w=v -> Iw=Iy
    Ip : np.ndarray
        If w=u -> Ip=Iy
        If w=v -> Ip=Ix
    lam : float
        Penalty term
    rhs : np.ndarray
        RHS in eq
    h : int
        Stepsize corresponding to the level

    Returns:
    ---
    None
    """
    n, m = w.shape
    k, d = Iw.shape

    # Corner update
    w[1 : n - 1 : 2, 1 : m - 1 : 2] = (
        rhs[0:k:2, 0:d:2]
        - Ip[0:k:2, 0:d:2] * Iw[0:k:2, 0:d:2] * p[1 : n - 1 : 2, 1 : m - 1 : 2]
        + lam
        / h**2
        * (
            # Left
            w[1 : n - 1 : 2, 0 : m - 2 : 2]
            # Right
            + w[1 : n - 1 : 2, 2:m:2]
            # Up
            + w[0 : n - 2 : 2, 1 : m - 1 : 2]
            # Down
            + w[2:n:2, 1 : m - 1 : 2]
        )
    ) / (Iw[0:k:2, 0:d:2] ** 2 + 4 * lam / h**2)

    # Interior corner update
    w[2 : n - 1 : 2, 2 : m - 1 : 2] = (
        rhs[1:k:2, 1:d:2]
        - Ip[1:k:2, 1:d:2] * Iw[1:k:2, 1:d:2] * p[2 : n - 1 : 2, 2 : m - 1 : 2]
        + lam
        / h**2
        * (
            # Left
            w[2 : n - 1 : 2, 1 : m - 2 : 2]
            # Right
            + w[2 : n - 1 : 2, 3:m:2]
            # Up
            + w[1 : n - 2 : 2, 2 : m - 1 : 2]
            # Down
            + w[3:n:2, 2 : m - 1 : 2]
        )
    ) / (Iw[1:k:2, 1:d:2] ** 2 + 4 * lam / h**2)

```

← Smoother (red)

$u(x, y)$



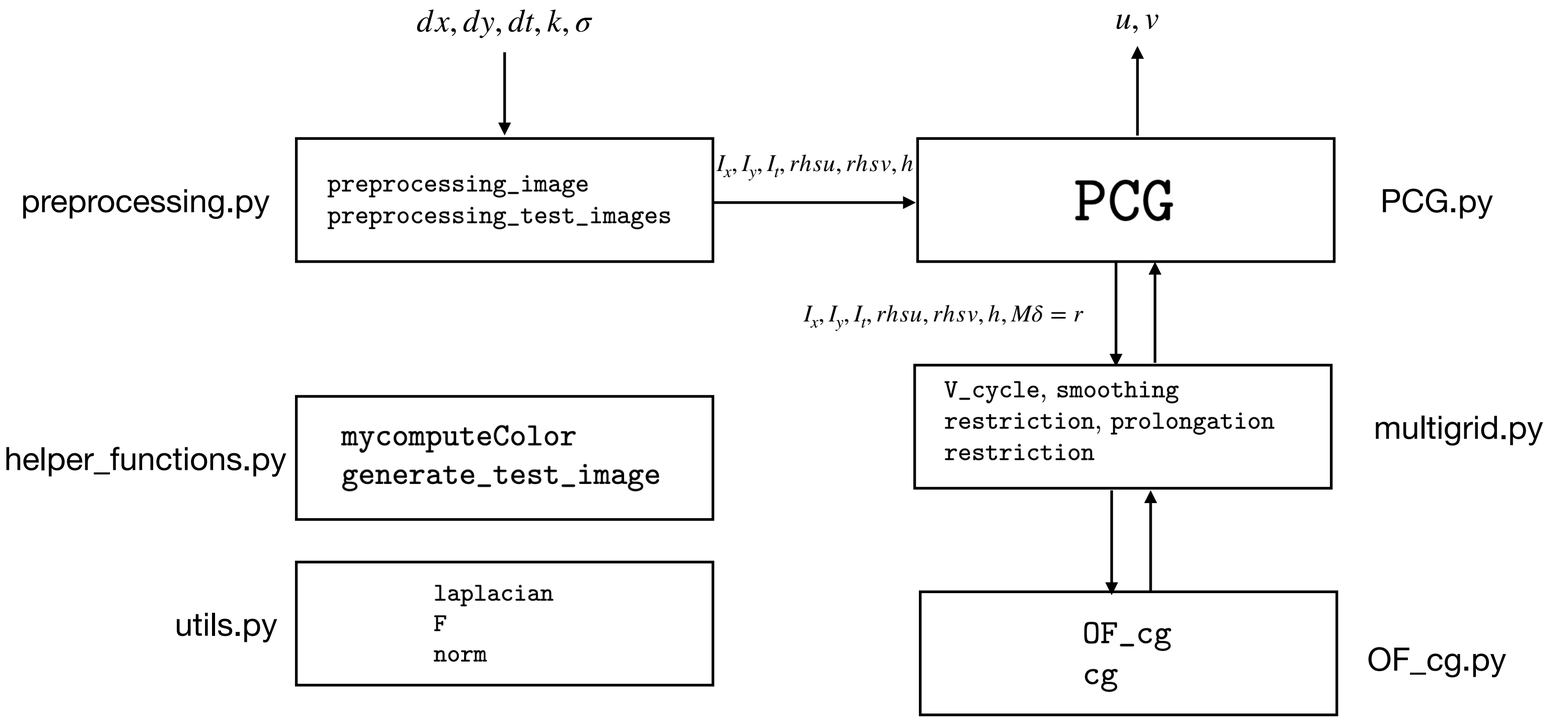
$v(x, y)$



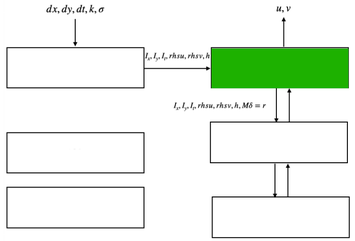
Red
Black

Red
Black

Workflow


dx, dy, dt, k, σ
u, v
preprocessing.py
preprocessing_image
preprocessing_test_images
$I_x, I_y, I_t, rhsu, rhsv, h$
PCG
PCG.py
$I_x, I_y, I_t, rhsu, rhsv, h, M\delta = r$
V_cycle, smoothing
restriction, prolongation
restriction
multigrid.py
helper_functions.py
mycomputeColor
generate_test_image
utils.py
laplacian
F
norm
OF_cg
cg
OF_cg.py

PCG



Algorithm 1 CG

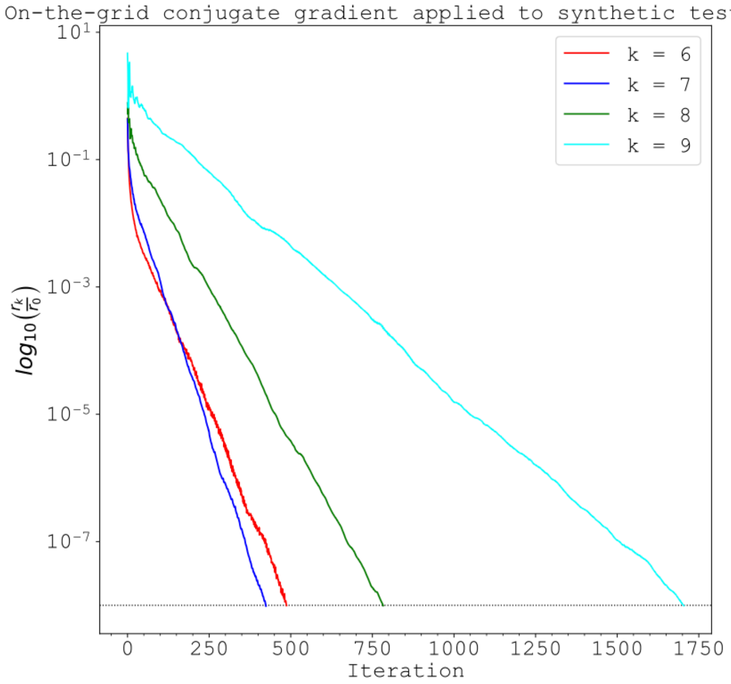
Compute $r_0 := b - Ax_0$, $p_0 := r_0$
for $j = 0, 1, \ldots,$ until convergence **do**
  $\alpha_j \leftarrow \frac{(r_j, r_j)}{(Ap_j, p_j)}$
  $x_{j+1} \leftarrow x_j + \alpha_j p_j$
  $r_{j+1} \leftarrow r_j - \alpha_j A p_j$
  $\beta_j \leftarrow \frac{(r_{j+1}, r_{j+1})}{(r_j, r_j)}$
  $p_{j+1} \leftarrow r_{j+1} + \beta_j p_j$
end for

Algorithm 2 PCG

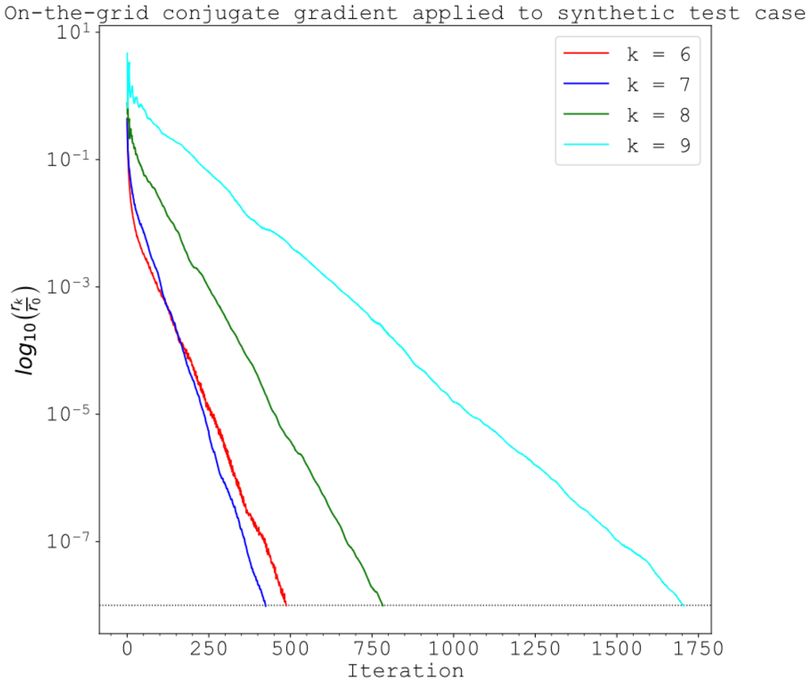
Compute $r_0 := b - Ax_0$, $z_0 := M^{-1} r_0$, $p_0 := z_0$
for $j = 0, 1, \ldots,$ until convergence **do**
  $\alpha_j \leftarrow \frac{(r_j, z_j)}{(Ap_j, p_j)}$
  $x_{j+1} \leftarrow x_j + \alpha_j p_j$
  $r_{j+1} \leftarrow r_j - \alpha_j A p_j$
  $z_{j+1} \leftarrow M^{-1} r_{j+1}$
  $\beta_j \leftarrow \frac{(r_{j+1}, z_{j+1})}{(r_j, z_j)}$
  $p_{j+1} \leftarrow z_{j+1} + \beta_j p_j$
end for

M needs to be symmetric positive definite!

Results (part I)



(a) Convergence plot of sparse implementation.



(b) Convergence plot of on-the-grid implementation.

Figure 2: Convergence plots of sparse vs on-the-grid implementations of the Conjugate Gradient algorithm, without multigrid, tested on two synthetic image frames depicting two Gaussian's circling each other. The image sizes were $2^k \times 2^k$, for $k = 6, 7, 8, 9$.

k	t_{SCG} [s]	t_{GCG} [s]	t_{SCG}/t_{GCG}
6	0.047	0.086	0.546
7	0.131	0.118	1.110
8	0.988	0.815	1.212
9	9.022	9.281	0.972

Table 1: Timing comparison for sparse (t_{SCG}) vs. on-the-grid (t_{GCG}) implementations of the Conjugate Gradient algorithm without multigrid, tested on two synthetic image frames depicting two Gaussians circling each other. Image sizes are $2^k \times 2^k$ for $k = 6, 7, 8, 9$.

Results (part II)

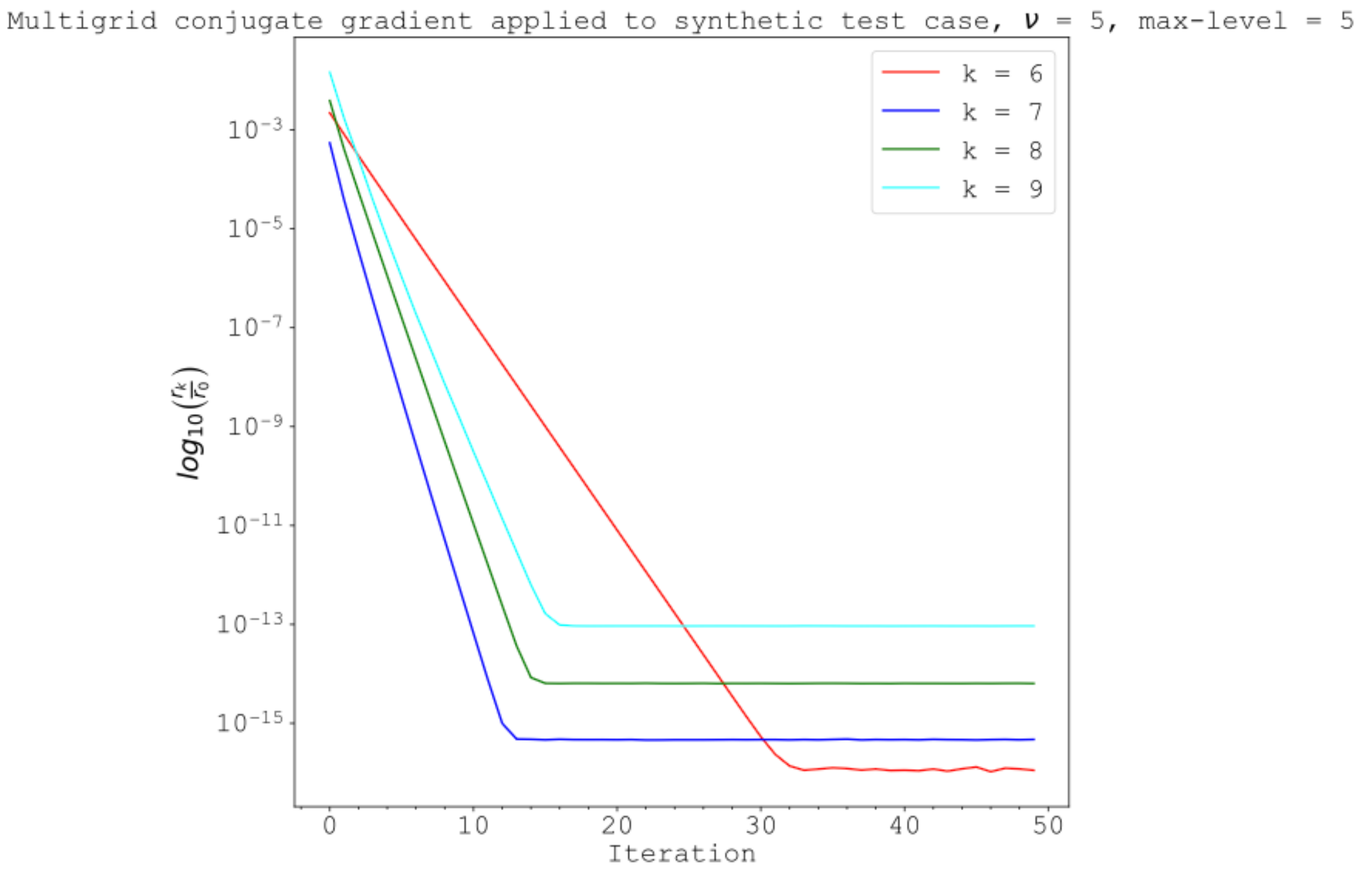
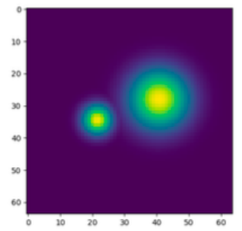
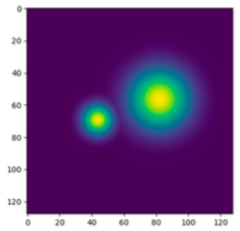


Figure 3: Convergence plot for multigrid conjugate gradient applied to the synthetic test case of two Guassian's circling each other for image sizes of $2^k \times 2^k$ for $k = 6, 7, 8, 9$.

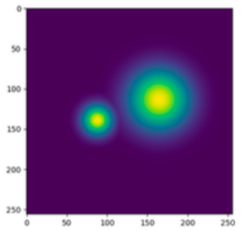
Results (part III)



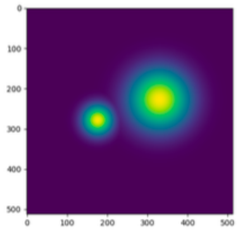
(a) k = 6



(b) k = 7

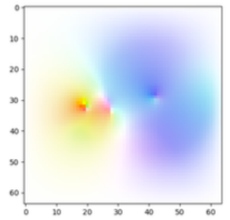


(c) k = 8

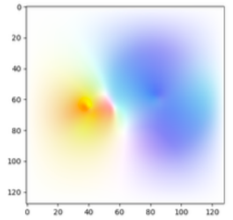


(d) k = 9

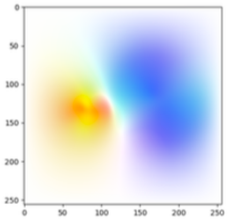
Figure 6: Second frame from synthetic test images of Gaussian's circling each other for image sizes of $2^k \times 2^k$ for $k = 6, 7, 8, 9$.



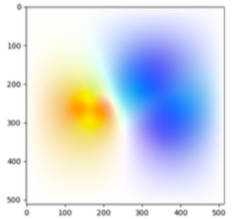
(a) k = 6



(b) k = 7



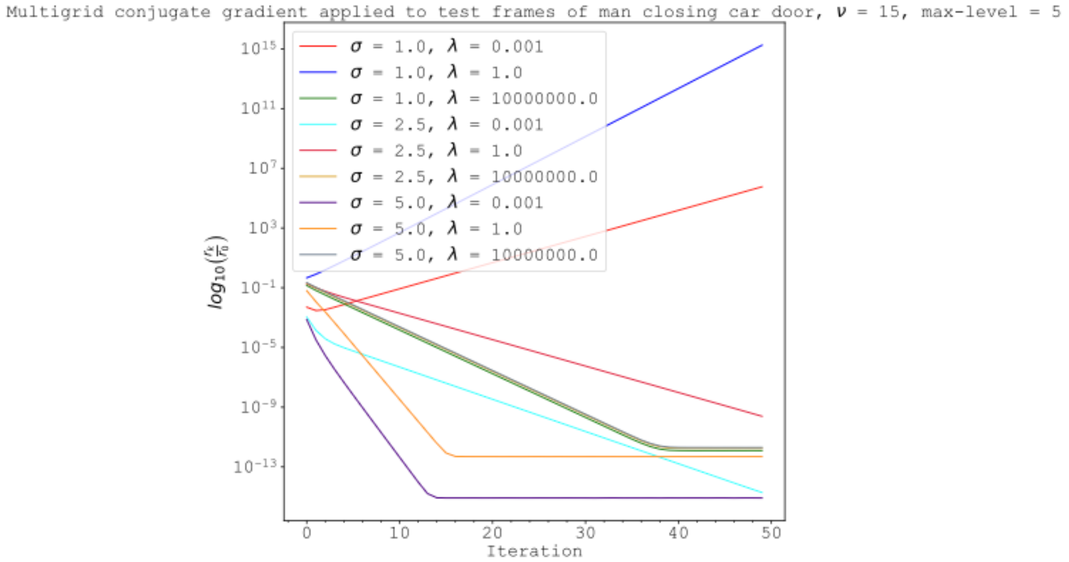
(c) k = 8



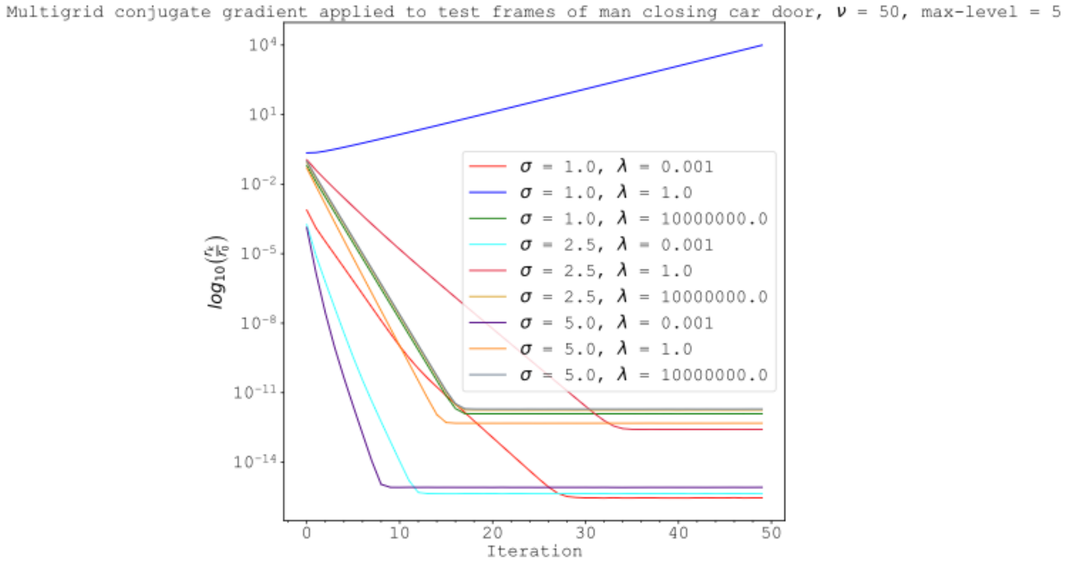
(d) k = 9

Figure 7: Resulting optical flow fields for the test case of two Gassians circling each other for image sizes of $2^k \times 2^k$ for $k = 6, 7, 8, 9$ using the sparse implementation of the conjugate gradient algorithm (on-the-grid implementation yields identical results).

Results (part IV)



(a) Convergence plot for multigrid V-cycle for different values of σ and λ, $\nu = 15$, max-lvl $= 5$.



(b) Convergence plot for multigrid V-cycle for different values of σ and λ, $\nu = 50$, max-lvl $= 5$.

Figure 4: Convergence plots for for multigrid conjugate gradient for various σ and λ values.

Results (part V)

σ	λ	t	$t_{\nu=15,\,\mathrm{lvl}=5}$	$t_{\nu=50,\,\mathrm{lvl}=5}$	$t_{\nu=15,\,\mathrm{lvl}=2}$
1.0	10^{-3}	0.0974	0.4135	1.2325	0.5367
1.0	10^{0}	0.0966	0.4008	1.2472	0.5177
1.0	10^{7}	0.0959	0.3988	1.3043	0.5087
2.5	10^{-3}	0.0961	0.3934	1.2449	0.4848
2.5	10^{0}	0.0969	0.3951	1.2290	0.5195
2.5	10^{7}	0.0977	0.3991	1.2549	0.5073
5.0	10^{-3}	0.0937	0.3891	1.2478	0.4470
5.0	10^{0}	0.0932	0.3826	1.3599	0.5340
5.0	10^{7}	0.0967	0.3982	1.2861	0.5144

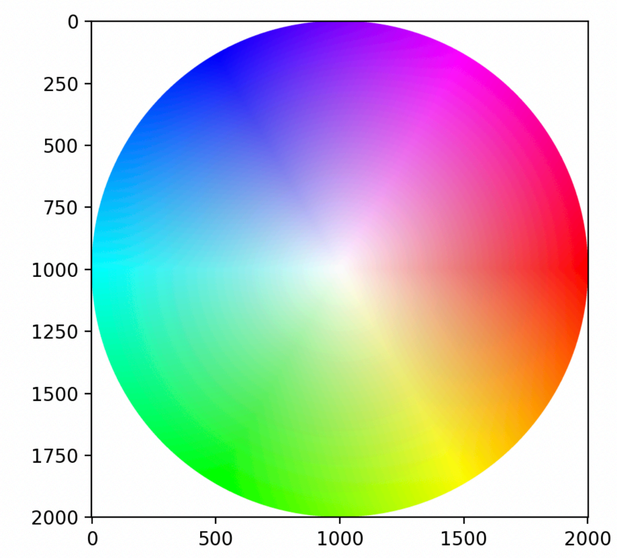
Table 2: Runtimes for multigrid V-cycle for various (σ, λ) and smoothing settings.

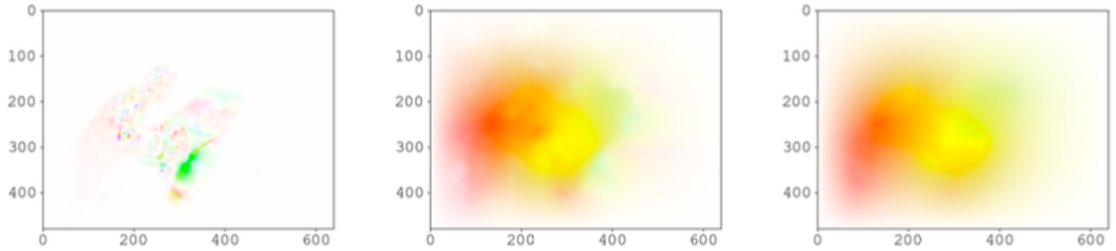
Results (part VI)

(a) First frame used for testing.

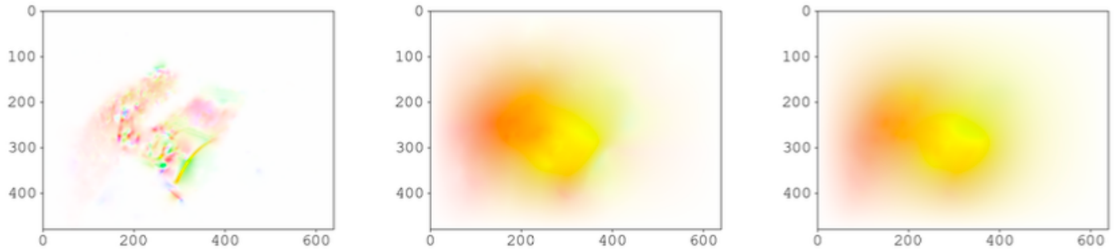
(b) Second frame used for testing.

Figure 8: The two frames of a man closing his car door which were used to test the implementation.

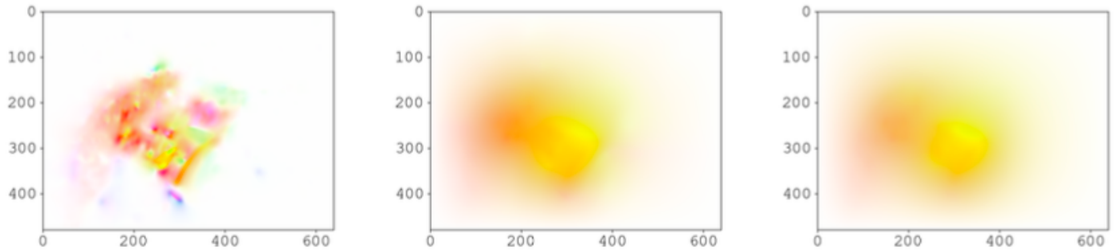




(a) $\sigma = 1.0$, $\lambda = 0.001$ (b) $\sigma = 1.0$, $\lambda = 1.0$ (c) $\sigma = 1.0$, $\lambda = 10^7$



(d) $\sigma = 2.5$, $\lambda = 0.001$ (e) $\sigma = 2.5$, $\lambda = 1.0$ (f) $\sigma = 2.5$, $\lambda = 10^7$



(g) $\sigma = 5.0$, $\lambda = 0.001$ (h) $\sigma = 5.0$, $\lambda = 1.0$ (i) $\sigma = 5.0$, $\lambda = 10^7$

Figure 9: Resulting optical flow field for the provided test frames of a man closing his car door using the multigrid conjugate gradient (on-the-grid, not sparse) algorithm. The number of iterations for the smoother was set to $\nu = 15$ and the max-level for the V-cycle was set to 5.

Results (part VII)

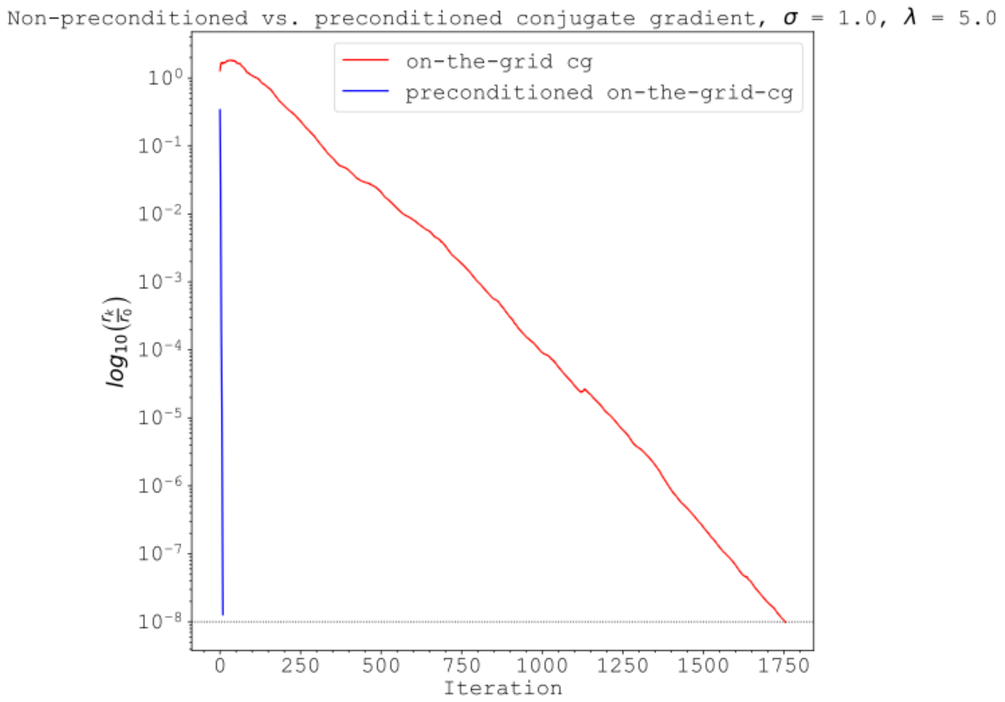


Figure 5: Convergence plot comparing the non-preconditioned on-the-grid implementation of the Conjugate Gradient algrotihm vs. the preconditioned version for the test frames of a man closing a car door.

12.210s to 4.325s.

Problems encountered

- Enforce Dirichlet B.C. ? (debugging CG padding vs work on interior)

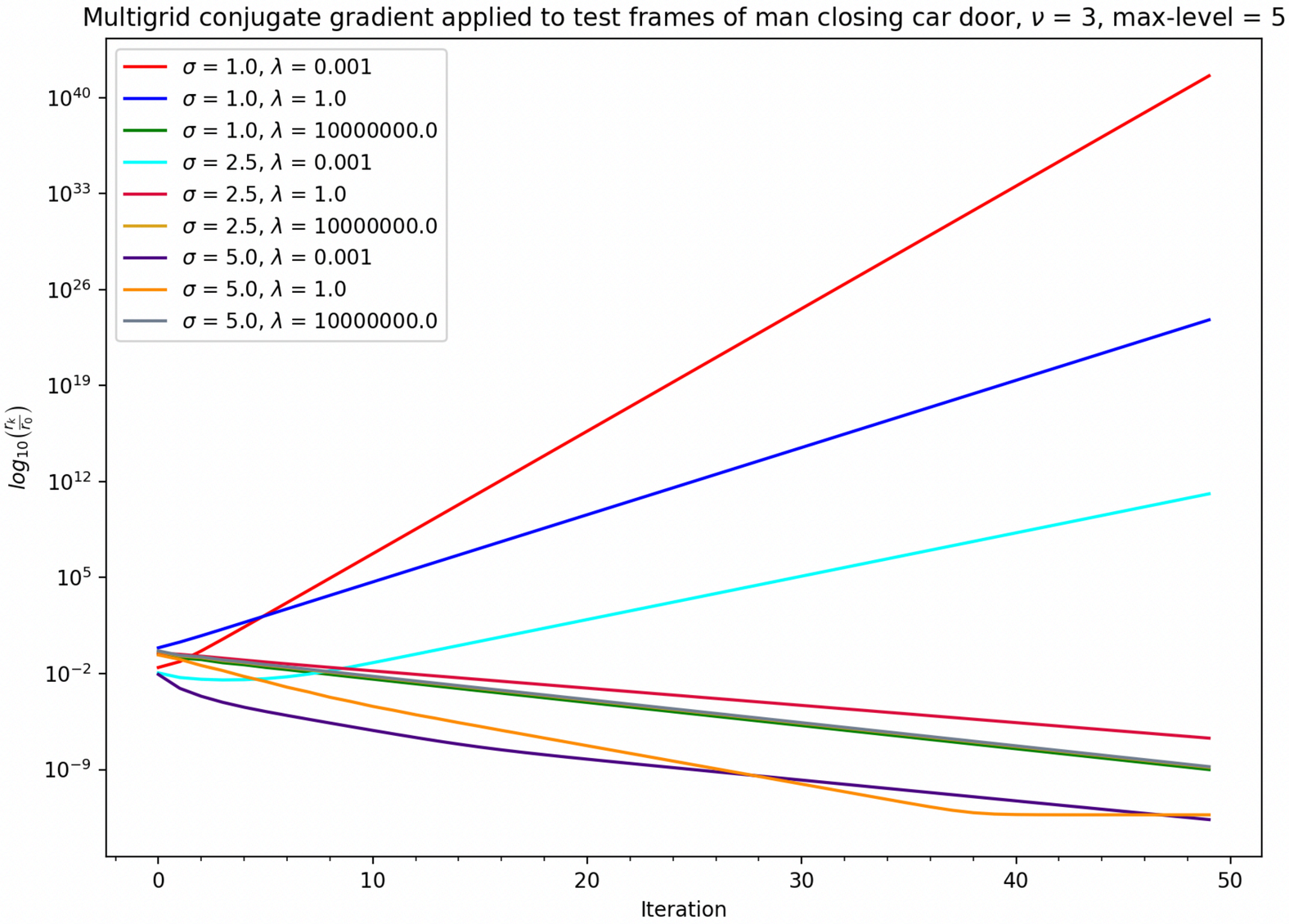
- Update order red-black? no hahahah IxIy IyIx IxIt IyIt messup, ANNOYING BUG!!!

- Multigrid V-cycle diverges for small values of lambda (if sigma = 1, lambda < 5.0 behaves badly)

- Why h and not 2h when calling conjugate gradient on the coarse grid?

Thank you, questions?

A note on smoothing



time = 0.14... [s]