Assigment III

Pablo Agustín Ortega-Kral

November 1, 2024

Problem 1. Consider the following relationship between a function $y(x)$ and its derivative $y'(x)$ over the interval $[0, 1]$.

$$3y^2y' = 1 \text{ with } y(1) = 1$$

Subproblem 1.a. *Obtain an exact analytic solution y(x) to this differential equation by hand.*

$$3y^2\frac{dy}{dx} = 1$$
$$3y^2dy = dx$$
$$\int 3y^2dy = \int dx$$
$$y^3 = x + c$$

We can substitute initial conditions $y(1) = 1$ to find the constant term

$$1^3 = 1 + c$$
$$1 = 1 + c$$
$$c = 0$$

Thus,

$$y = \sqrt[3]{x} \text{ , with roots } x = 0$$

Subproblem 1.b - 1.d. *Implement Euler, Runge-Kutta, and Adams-Bashforth numerical methods for solving differential equations.*

```python
def euler_solver(f:callable, x0: float, y0: float, step: float, n_iter: int) -> tuple:
    x = [x0]
    y = [y0]
    for i in range(1, n_iter+1):
        x.append(x0 + i*step)
        y.append(y[i-1] + step*f(x[i-1], y[i-1]))
    return x, y

def runge_kutta4_solver(f: callable, x0: float, y0: float, step: float, n_iter: int) ->
    tuple:
    x = [x0]
    y = [y0]
    for i in range(1, n_iter+1):
        x.append(x0 + i*step)
        # Calculate runge kutta constants
        k1 = step*f(x[i-1], y[i-1])
        k2 = step*f(x[i-1] + step/2, y[i-1] + k1/2)
        k3 = step*f(x[i-1] + step/2, y[i-1] + k2/2)
        k4 = step*f(x[i-1] + step, y[i-1] + k3)
```

```
        y_hat = y[i-1] + (k1 + 2*k2 + 2*k3 + k4)/6
        y.append(y_hat)
    return x, y

def adams_bashforth_solver(f: callable, x0: list, y0: list, step: float, n_iter: int) ->
    tuple:
    x = x0
    y = y0
    for i in range(4, n_iter+4):
        x.append(x0[0] + i*step)
        y_hat = y[i-1] + step/24*(55*f(x[i-1], y[i-1]) - 59*f(x[i-2], y[i-2]) + 37*f(x[i-3],
     y[i-3]) - 9*f(x[i-4], y[i-4]))
        y.append(y_hat)
    return x[3:], y[3:]
```

Listing 1: Implementation of solvers

Euler			**Runge Kutta**			**Adams Bashforth**			**Exact**
y	f(x,y)	Error	y	f(x,y)	Error	y	f(x,y)	Error	Solution
1.00	0.33	0.00	1.00	0.33	0.00	1.00	0.33	0.00	1.00
0.98	0.34	-0.00	0.98	0.34	2.32e-10	0.98	0.34	-3.13e-07	0.98
0.97	0.36	-0.00	0.97	0.36	5.35e-10	0.97	0.36	-7.32e-07	0.97
0.95	0.37	-0.00	0.95	0.37	9.38e-10	0.95	0.37	-1.27e-06	0.95
0.93	0.39	-0.00	0.93	0.39	1.48e-09	0.93	0.39	-1.98e-06	0.93
0.91	0.40	-0.00	0.91	0.40	2.21e-09	0.91	0.40	-2.92e-06	0.91
0.89	0.42	-0.00	0.89	0.42	3.23e-09	0.89	0.42	-4.20e-06	0.89
0.87	0.44	-0.00	0.87	0.44	4.65e-09	0.87	0.44	-5.96e-06	0.87
0.85	0.46	-0.00	0.84	0.47	6.71e-09	0.84	0.47	-8.43e-06	0.84
0.82	0.49	-0.00	0.82	0.50	9.76e-09	0.82	0.50	-0.00	0.82
0.80	0.52	-0.01	0.79	0.53	1.44e-08	0.79	0.53	-0.00	0.79
0.77	0.56	-0.01	0.77	0.57	2.18e-08	0.77	0.57	-0.00	0.77
0.75	0.60	-0.01	0.74	0.61	3.42e-08	0.74	0.61	-0.00	0.74
0.72	0.65	-0.01	0.70	0.67	5.62e-08	0.70	0.67	-0.00	0.70
0.68	0.71	-0.01	0.67	0.74	9.83e-08	0.67	0.74	-0.00	0.67
0.65	0.80	-0.02	0.63	0.84	1.88e-07	0.63	0.84	-0.00	0.63
0.61	0.90	-0.02	0.58	0.97	4.08e-07	0.59	0.97	-0.00	0.58
0.56	1.05	-0.03	0.53	1.18	1.08e-06	0.53	1.18	-0.00	0.53
0.51	1.28	-0.05	0.46	1.55	3.99e-06	0.47	1.54	-0.00	0.46
0.45	1.68	-0.08	0.37	2.46	0.00	0.38	2.36	-0.01	0.37
0.36	2.55	-0.36	0.10	31.14	-0.10	0.22	6.84	-0.22	0.00

Table 1: Comparisson between numerical solvers

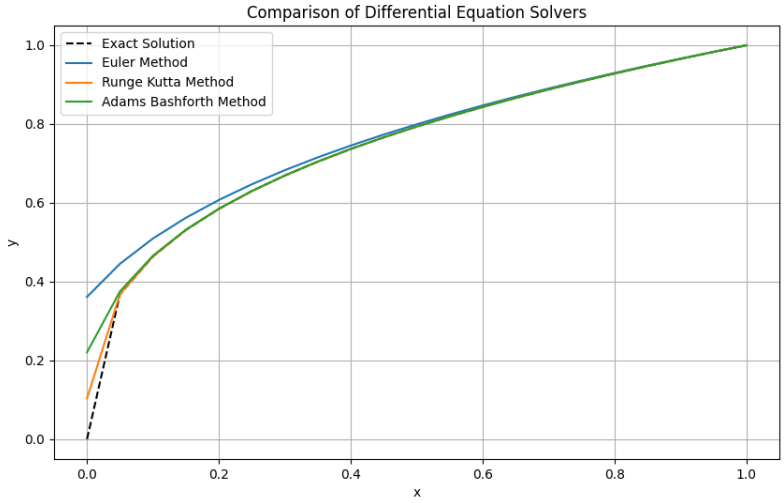


Figure 1: Comparisson between produced Y values

Problem 2. Consider the function $f(x, y) = x^3 + y^3 - 2x^2 + 3y^2 - 8$

Subproblem 2.a. *Find all critical points and classify the critical points into local minima, local maxima, and saddle points by considering either nearby gradient directions or the Hessian of f.*

For determining critical points for f, we solve the equations formed $\frac{\partial f}{\partial x} = 0$ and $\frac{\partial f}{\partial y}$

$$\frac{\partial f}{\partial x} = 3x^2 - 4x$$
$$\frac{\partial f}{\partial y} = 3y^2 + 6y$$

We solve each equality to find the coordinates of the critical points.

$$3x^2 - 4x = 0$$
$$x(3x - 4) = 0$$

Thus for x we have two possible equations, the trivial case where $x = 0$ and,

$$3x - 4 = 0$$
$$x = \frac{4}{3}$$

For y we also see that

$$y(3x + 6) = 0$$

Therefore we also consider the trivial case $y = 0$ as well as

$$3y + 6 = 0$$
$$y = \frac{-6}{3} = -2$$

The critical points are thus $\{(0, 0), (\frac{4}{3}, 0), (0, -2), (\frac{4}{3}, -2)\}$. We can classify them by calculating the hessian, $H = \begin{bmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial y^2} \end{bmatrix}$

$$\frac{\partial^2 f}{\partial x^2} = 6x - 4$$
$$\frac{\partial^2 f}{\partial y^2} = 6y + 6$$

Given that there are no terms with both x and y, we see that the hessian is

$$H = \begin{bmatrix} 6x - 4 & 0 \\ 0 & 6y + 6 \end{bmatrix}$$

We can classify the critical points by looking into the value of the Hessian's determinant evaluated for the given coordinates

$$\Delta = (6x - 4)(6y + 6)$$

Subproblem 4.b. *By hand, show how the steepest descent would behave starting from the point $(x, y) = (1, -1)$. Use the version of the steepest descent that moves to the nearest local minimum on the negative gradient line, as discussed in class. How many such steepest descent steps are needed to converge to an overall local minimum of f?*

Coordinates (x, y)	**Determinant of Hessian** Δ	**Classification**
$(0, 0)$	-24	Saddle point, given determinant is negative
$(0, -2)$	24	Local maximum, given determinant is positive and $\frac{\partial^2 f}{\partial x^2} < 0$
$(\frac{4}{3}, 0)$	24	Local minimum, given determinant is positive and $\frac{\partial^2 f}{\partial x^2} > 0$
$(\frac{4}{3}, -2)$	-24	Saddle point, given determinant is negative

Table 2: Classification of Critical Points

Given that the gradient points to the direction that maximizes a function, we follow the gradient in the opposite direction to find a local minimum. We can do this iteratively by defining an update function $g(t)$, such that

$$g(t) = f(x^{(m)} - t\Delta f)$$

By minimizing $g(t)$ we can always take the step in the steepest direction of the gradient, by finding t^*. Applying this to the giving problem, and considering that

$$\Delta f(x, y) = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix} = \begin{bmatrix} 3x^2 - 4x \\ 3y^2 + 6y \end{bmatrix}$$

For the first iteration $m = 0$

$$x^{(0)} = (1, -1)$$

We have the gradient,

$$\Delta f(1, -1) = \begin{bmatrix} 3(1)^2 - 4(1) \\ 3(-1)^2 + 6(-1) \end{bmatrix} = \begin{bmatrix} -1 \\ -3 \end{bmatrix}$$

this gives $g(t) = f(\begin{bmatrix} 1 \\ -1 \end{bmatrix} - t \begin{bmatrix} -1 \\ -3 \end{bmatrix})$,

$$g(t) = f(1 + t, -1 + 3t)$$

$$g(t) = (1 + t)^3 + (-1 + 3t)^3 - 2(1 + t)^2 + 3(-1 + 3t)^2 - 8$$

$$g(t) = 28t^3 + t^2 - 10t - 7$$

We can find the optimal step by optimizing $g(t)$ by finding its critical points

$$g'(t) = 84t^2 + 2t - 10$$

$$2(3t - 1)(14t + 5) = 0$$

$$3t - 1 = 0$$

$$t_0 = \frac{1}{3}$$

$$14t + 5 = 0$$

$$t_1 = \frac{-5}{14}$$

We determine which is minima by evaluating the second derivative

$$g''(t) = 168t + 2$$

$$g''(t_0) = 168(\frac{1}{3}) + 2 = 58$$

$$g''(t_1) = 168(\frac{-5}{14}) + 2 = -58$$

This is the local minima given that $g''(t_0) > 0$. Therefore, $t^* = \frac{1}{3}$. We compute the next update as,

$$x^{m+1} = \begin{bmatrix} 1 \\ -1 \end{bmatrix} - \frac{1}{3}\begin{bmatrix} -1 \\ -3 \end{bmatrix}$$

$$x^{m+1} = \begin{bmatrix} 1 + \frac{1}{3} \\ -1 + 1 \end{bmatrix} = \begin{bmatrix} \frac{4}{3} \\ 0 \end{bmatrix}$$

Looking at the results from the previous exercise we know that this is a local minima, we can verify by evaluating the gradient for the next iteration, which should yield 0.

$$\Delta f(\frac{4}{3}, 0) = \begin{bmatrix} 3(\frac{4}{3})^2 - 4(\frac{4}{3}) \\ 3(0)^2 + 6(0) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

The gradient is 0 meaning $(\frac{4}{3}, 0)$ is indeed a local minima. This means we have **converged to a local minima in 1 step**.

Problem 3. Let Q be a real symmetric positive definite $n \times n$ matrix. Show that any two eigenvectors of Q corresponding to distinct eigenvalues of Q are Q-orthogonal. Show this directly from the definition of eigenvector, without assuming eigenvectors are orthogonal in the usual sense.

By definition, we know that for two eigenvalues λ_1, λ_2, eigenvectors $\vec{X}$ follow

$$Q\vec{X_1} = \lambda_1\vec{X_1}$$
$$Q\vec{X_2} = \lambda_2\vec{X_2}$$

We aim to prove that $\vec{X_1}$ and $\vec{X_2}$ are orthogonal with respect to the matrix Q, meaning their inner produced, weighed by Q, is equal to 0.

$$\vec{X_1}^T Q\vec{X_2} = 0$$

We can begin by writing $Q\vec{X_2}$ as $\lambda_2\vec{X_2}$ following the eigenvector definition,

$$\vec{X_1}^T \lambda_2\vec{X_2} = 0$$
$$\lambda_2(\vec{X_1}^T\vec{X_2}) = 0$$

Knowing that Q is symmetric, implying $Q = Q^T$ we can express the inner product

$$\vec{X_1}^T Q^T\vec{X_2} = 0$$
$$(Q\vec{X_1})^T\vec{X_2} = 0$$

We can now write the inner product in terms of λ_1

$$(\lambda_1\vec{X_1})^T\vec{X_2} = 0$$

Knowing that eigenvalues are scalars,

$$\lambda_1(\vec{X_1}^T\vec{X_2}) = 0$$

We have two expressions describing the inner product in terms of two eigenvalues, this should be equivalent. We can prove orthogonality by the following equality

$$\lambda_1(\vec{X_1}^T\vec{X_2}) = \lambda_2(\vec{X_1}^T\vec{X_2})$$
$$\lambda_1(\vec{X_1}^T\vec{X_2}) - \lambda_2(\vec{X_1}^T\vec{X_2}) = 0$$
$$(\lambda_1 - \lambda_2)(\vec{X_1}^T\vec{X_2}) = 0$$

Given that eigenvalues have proper values and cannot be the same, this means that $\vec{X_1}^T\vec{X_2} = 0$, and therefore are orthogonal respect to their inner product with Q, ie. they are Q-orthogonal.

Problem 4. Analysis of purely quadratic form of the conjugate gradient method.

Subproblem 4.a. *Show that in the purely quadratic form of the conjugate gradient method,* $d_k^T Q d_k = -d_k^T Q g_k$. *Consequently, show that to obtain* x_{k+1} *from* x_k *one does not need to use* Q *explicitly, assuming one already has available the vectors* g_k, Qg_k, *and* d_k.

We can show this by using the update rule in the conjugate gradient method to express the search direction in terms of the gradient,

$$d_k = -g_k + \beta d_{k-1}$$

We substitute d_k into $d_k^T Q d_k$

$$d_k^T Q d_k = d_k^T Q(-g_k + \beta_k d_{k-1})$$
$$d_k^T Q d_k = -d_k^T Q g_k + \beta_k - d_k^T Q d_{k-1}$$

Note that by definition, the search directions in the conjugate gradient method are Q-orthogonal, meaning that for $i \neq j$, $d_i^T Q d_j$. This allows us to rewrite the previous expression as

$$d_k^T Q d_k = -d_k^T Q g_k + \beta_k(0)$$
$$d_k^T Q d_k = -d_k^T Q g_k$$

Subproblem 4.b. *Show that in the purely quadratic form,* Qg_k *can be computed by taking a unit step from* x_k *in the direction of the negative gradient and then evaluating the gradient there. Specifically, if* $y_k = x_k - g_k$ *and* $p_k = \Delta f(y_k)$, *then* $Qg_k = g_k - p_k$

We aim to show that for the purely quadratic form $\frac{1}{2}x^T Q x + b^T x$, $Qg_k = g_k - p_k$. Where $p_k = \Delta f(y_k)$

$$\Delta f(x_k) = g_k = Qx_k + b^T$$
$$p_k = Q(x_k - g_k) + b^T$$
$$p_k = Q(x_k - g_k) + b^T$$
$$p_k = Qx_k - Qg_k + b^T$$

Here we can solve for Qg_k

$$p_k + Qg_k - Qx_t - b^T = 0$$
$$Qg_k = Qx_t + b^T - p_k$$

Given that $g_k = Qx_t + b^T$,

$$Qg_k = g_k - p_k$$

Subproblem 4.c. *Combine the results of parts (a) and (b) to derive a conjugate gradient method for general functions* f *in the spirit of the algorithm presented in class, but which does not require knowledge of the Hessian of* f *or a line search.*

We can achieve this by parametrizing the update of x_k with $p_k = Q(x_k - g_k) + b^T$. By substituting the equality found in a into α_k and β_k.

$$\beta_k = \frac{g_{k+1}^T Q d_k}{dk^T Q d_k}$$

Changing $d_k^T Q d_k = -d_k^T Q g_k$

$$\beta_k = \frac{g_{k+1}^T Q d_k}{-d_k^T Q g_k}$$

We do the same for alpha,

$$\alpha_k = \frac{-g_k^T d_k}{d_k^T Q d_k}$$

$$\alpha_k = \frac{-g_k^T d_k}{-d_k^T Q g_k}$$

Now, we substitute the equality found in b $Qg_k = g_k - p_k$,

$$\beta_k = \frac{g_{k+1}^T Q d_k}{-d_k^T Q g_k} = \frac{(g_{k+1} - p_{k+1})^T d_k}{-d_k^T (g_k - p_k)}$$

$$\alpha_k = \frac{-g_k^T d_k}{-d_k^T (g_k - p_k)}$$

Remembering the statement of b), this means that both updates can be computed by taking a unit step from x_k in the direction of the negative gradient and then evaluating the gradient there.

Algorithm 1 Proposed Algorithm

Initialize: $\mathbf{d}_0 \leftarrow -\mathbf{g}_0$
for $k = 0, \ldots, n-1$ **do**
    $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k \mathbf{d}_k$
    $\alpha_k \leftarrow \frac{-g_k^T d_k}{-d_k^T (g_k - p_k)}$
    $\mathbf{d}_{k+1} \leftarrow -\mathbf{g}_{k+1} + \beta_k \mathbf{d}_k$
    $\beta_k \leftarrow \frac{(g_{k+1} - p_{k+1})^T d_k}{-d_k^T (g_k - p_k)}$
end for
Return $\mathbf{x}_n$

Problem 5. Find the rectangle with the greatest area of a given perimeter by solving the Lagrange multiplier's first-order necessary conditions. Verify the second-order sufficiency conditions.

We want the optimize the area function of a rectangle constrained on its perimeter

$$A(x, y) = xy$$
$$P(x, y) = 2(x + y)$$

Given that the constraint on the perimeter function we say $P(x, y) = 2(x + y) = C$. We can define the lagrangian as,

$$\mathcal{L}(x, y, \lambda) = A(x, y) + \lambda P(x, y)$$
$$\mathcal{L}(x, y, \lambda) = xy + \lambda(C - 2(x + y))$$

To find the critical points, we solve the system of equations formed by $\frac{\partial \mathcal{L}}{\partial x} = 0$, $\frac{\partial \mathcal{L}}{\partial y} = 0$, $\frac{\partial \mathcal{L}}{\partial \lambda} = 0$

$$\frac{\partial \mathcal{L}}{\partial x} = y - 2\lambda = 0$$
$$\frac{\partial \mathcal{L}}{\partial y} = x - 2\lambda = 0$$
$$\frac{\partial \mathcal{L}}{\partial \lambda} = C - 2(x + y) = 0$$

We can solve for x, y in terms of λ directly

$$x = 2\lambda$$
$$y = 2\lambda$$

Substituting into $\frac{\partial \mathcal{L}}{\partial \lambda}$,

$$C - 8\lambda = 0$$
$$\lambda = \frac{C}{8}$$

Therefore, the dimensions that maximize the area given a perimeter C are

$$x = y = \frac{C}{4}$$

To verify the second-order sufficiency conditions, we calculate the Hessian

$$H = \begin{bmatrix} \frac{\partial^2 \mathcal{L}}{\partial x^2} & \frac{\partial^2 \mathcal{L}}{\partial x \partial y} & \frac{\partial^2 \mathcal{L}}{\partial x \partial \lambda} \\ \frac{\partial^2 \mathcal{L}}{\partial y \partial x} & \frac{\partial^2 \mathcal{L}}{\partial y^2} & \frac{\partial^2 \mathcal{L}}{\partial y \partial \lambda} \\ \frac{\partial^2 \mathcal{L}}{\partial \lambda \partial x} & \frac{\partial^2 \mathcal{L}}{\partial \lambda \partial y} & \frac{\partial^2 \mathcal{L}}{\partial \lambda^2} \end{bmatrix}$$
$$H = \begin{bmatrix} 0 & 1 & -2 \\ 1 & 0 & -2 \\ -2 & -2 & 0 \end{bmatrix}$$

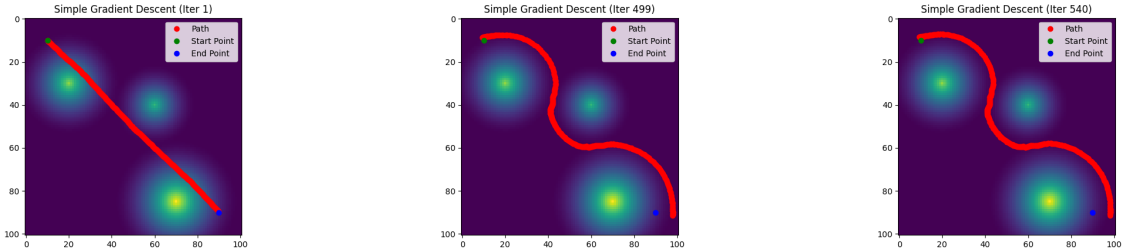
We evaluate the determinant of the Hessian

$$\texttt{det}(H) = 8$$

Given the determinate is definite positive, we satisfy the second-order sufficiency conditions.

Problem 6. In this problem, you will use trajectory optimization to find obstacle-free paths for a robot. You will start with an obstacle-cost world and a straight-line path that blasts through the obstacles, as in Figures 1 and 2 on page 4. We are providing some starter code to make it easier – see the file trajectory optimization. m—py. For your code, you mainly just need to add the actual optimization code to one of these files.

Subproblem 6.a. *Perform gradient descent to find a path with locally optimal cost. You should represent the path as a sequence of 2D points, view that sequence as a higher-dimensional point in some finite-dimensional space, and take gradient steps in that space. Rather than optimize along the entire gradient line as we did in lecture, simply scale the negative gradient by 0.1 and take a corresponding step. Remember, this is a path for your robot, so make sure you don't accidentally optimize its start and goal as well; those should remain fixed. Plot the path after one iteration — notice that it is moving away from the obstacles. Pick convergence criteria and plot the path after convergence. What happened to it?*

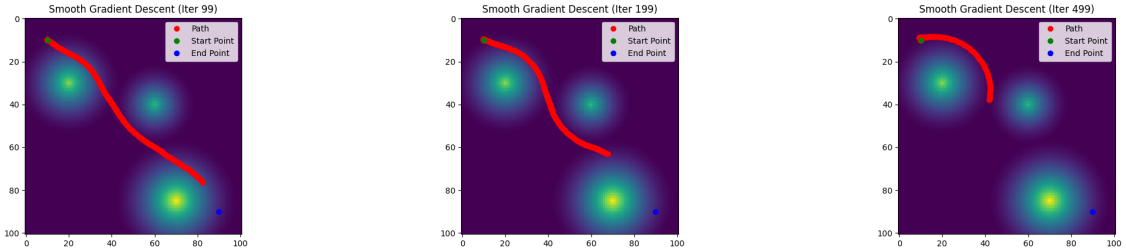


(a) 1 iteration (b) 500 iterations (c) 540 iterations

Figure 2: Path optimization with simple gradient descent

This approach does not yield good results because there is no notion of a connection between the points in the paths; therefore, the optimization is not constrained to group consecutive points and the path begins to separate.

Subproblem 6.b. *To avoid the issue from part (a), you need to tell the optimizer that this is a path instead of some independent 2D points. Implement it. Why does it not work?*

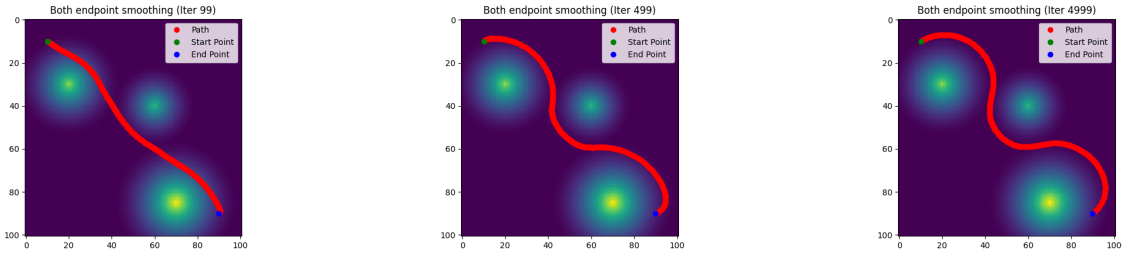


(a) 100 iterations (b) 200 iterations (c) 500 iterations

Figure 3: Path optimization with smoothed gradient descent

We can see that as the optimization runs, the path appears to be receding to one of the endpoints. This is because the smoothness term introduced seeks to minimize the distance between a given point and its predecessors, therefore the best solution will tend to group all points on the left side.

Subproblem 6.c. *Finally, try augmenting the obstacle gradient with a different smoothness cost, telling each 2D point to be close to both its previous and next points on the path:*



(a) 100 iterations (b) 500 iterations (c) 5000 iterations

Figure 4: Path optimization with bilaterally-smoothed gradient descent

We can solve the previous issues by introducing a term that accounts for the gradient of distance of points both before and after. We can see that this yields smoother paths that are able to reach both endpoints.

Subproblem 6.d. *What would happen if you were to start the procedure from different initial paths? Will the final answer be the same every time? Why?*

Given that gradient descent optimization finds local minima, there is no guarantee that different starting conditions yield the same result. It is likely, that different starting conditions will result in different final paths, as they where converge to the closest local minima.

Subproblem 6.e. *Suppose that the final solution path found by gradient descent, given some initial path and given our original cost function is not feasible for the robot to execute (it still passes through some high-cost regions that you would rather not have the robot pass through)? What simple common-sense strategy can you come up with to try to mitigate this problem?*

Taking advantage of the above statement, multiple different random starting paths can be selected to produce multiple rollouts of possible paths, the simplest solution would be to pick the paths with the best performance. If this does not work, performing a hyperparameter sweep on learning rate and smoothness constants could help shape the paths to the desired outcome.

Subproblem Code.

```
def zero_gradient(path, grad_values, tol = 1e-5):
    np.linalg.norm(grad_values, axis=1)
    return  np.all(grad_values < tol)

def get_path_grad(path):
    path_grad = np.empty(path.shape)
    for i in range(1, path.shape[0]):
        path_grad[i] = path[i] - path[i-1]
    return path_grad

def get_biliteral_path_grad(path):
    path_grad = np.empty(path.shape)
    for i in range(1, path.shape[0]-1):
        path_grad[i] = (path[i] - path[i-1])  + (path[i] - path[i+1])
    return path_grad

def optmize_path_gd(path: np.array, grad : tuple, convergence_criteria : callable,
                    obstacle_cost: np.array, max_iter: int = 10000, learning_rate: float =
    0.1):
    gx, gy = grad
    for i in range(max_iter):
        # Read gradient from precomputed values
        x, y = path.astype(int).T
        grad_values = np.array([gx[x, y], gy[x, y]]).T
```

```
        # Update path
        path[1:-1] = path[1:-1] - learning_rate * grad_values[1:-1]
        path = np.clip(path, BOUNDS[0], BOUNDS[1])
        if i == 1:
            plot_scene(path, obstacle_cost, tag="first_iter_simple_gradient", title=f"Simple Gradient Descent (Iter {i})")
        if convergence_criteria(path, learning_rate* grad_values, tol = 1e-2):
            print("Converged!")
            break
    print(f"Optimization finished after {i} iterations")
    return path, i

def optmize_path_smooth(path: np.array, grad : tuple, convergence_criteria : callable,
                    obstacle_cost: np.array, max_iter: int = 10000, learning_rate: float = 0.1, smooth_factor: tuple = (0.8, 4)):
    gx, gy = grad
    for i in range(max_iter):
        # Read gradient from precomputed values
        x, y = path.astype(int).T
        grad_values = np.array([gx[x, y], gy[x, y]]).T
        path_grad = get_path_grad(path)
        # Update path
        path[1:-1] = path[1:-1] - learning_rate * (smooth_factor[0]*grad_values + smooth_factor[1]* path_grad)[1:-1]
        path = np.clip(path, BOUNDS[0], BOUNDS[1])
        if convergence_criteria(path, grad_values, tol = 1e-5):
            break

    print(f"Optimization finished after {i} iterations")
    return path, i

def optmize_path_smooth_v2(path: np.array, grad : tuple, convergence_criteria : callable,
                    obstacle_cost: np.array, max_iter: int = 10000, learning_rate: float = 0.1, smooth_factor: tuple = (0.8, 4)):
    gx, gy = grad
    for i in range(max_iter):
        # Read gradient from precomputed values
        x, y = path.astype(int).T
        grad_values = np.array([gx[x, y], gy[x, y]]).T
        path_grad = get_biliteral_path_grad(path)
        # Update path
        path[1:-1] = path[1:-1] - learning_rate * (smooth_factor[0]*grad_values + smooth_factor[1]* path_grad)[1:-1]
        path = np.clip(path, BOUNDS[0], BOUNDS[1])
        if convergence_criteria(path, grad_values, tol = 1e-5):
            break

    print(f"Optimization finished after {i} iterations")
    return path, i
```

Listing 2: Path Optimization