AEROSP 588 Homework 2

Andi Zhou

September 27, 2023

1 Kepler's Equation

We are given the Kepler's equation:

$$E - e\sin(E) = M \tag{1}$$

where M is the mean anomaly (a parameterization of time), E is the eccentric anomaly (a parameterization of polar angle), and e is the eccentricity of the elliptical orbit.

1.1 Newton's Method

We implement Newton's method with $e = 0.7$ and $M = \pi/2$. Newton's method are structured as following:

$$u_{k+1} = u_k - \frac{r(u_k)}{r'(u_k)} \tag{2}$$

Kepler's equation is given in Equation 1, which we rewrite as:

$$\begin{aligned} E - e\sin(E) &= M \\ E - e\sin(E) - M &= 0 \end{aligned} \tag{3}$$

We take the derivative with respect to E:

$$\frac{\partial}{\partial E}(E - e \cdot \sin(E) - M) = 1 - e \cdot \cos(E) \tag{4}$$

The Newton's method is then expressed as:

$$E_{k+1} = E_k - \frac{E_k - e\sin(E_k) - M}{1 - e \cdot \cos(E_k)} \tag{5}$$

From Newton's method, the answer, converged to machine accuracy $< 1E - 16$, is $\boxed{E = 2.15478}$. For this problem, we are able to converge the tolerance down to machine accuracy, which is typically situated around 1×10^{-16}. Any more convergence will result in the computer spitting out $\text{tol} = 0$ directly as the output numbers are not longer differentiable.

1.2 Fixed-Point Iteration

For Fixed-Point Iteration, we re-arrange the equations such that:

$$\begin{aligned} E - e\sin(E) &= M \\ E &= M + e \cdot \sin(E) \end{aligned} \tag{6}$$

We re-assign the notation and the equation then becomes:

$$E_{k+1} = M + e \cdot \sin(E_k) \tag{7}$$

When we code up the process in Python, we obtain the same result $\boxed{E = 2.15478}$. However, this iteration took significantly longer than Newton's method, due to its linear rate of convergence.

1.3 Iteration and Rate of Convergence

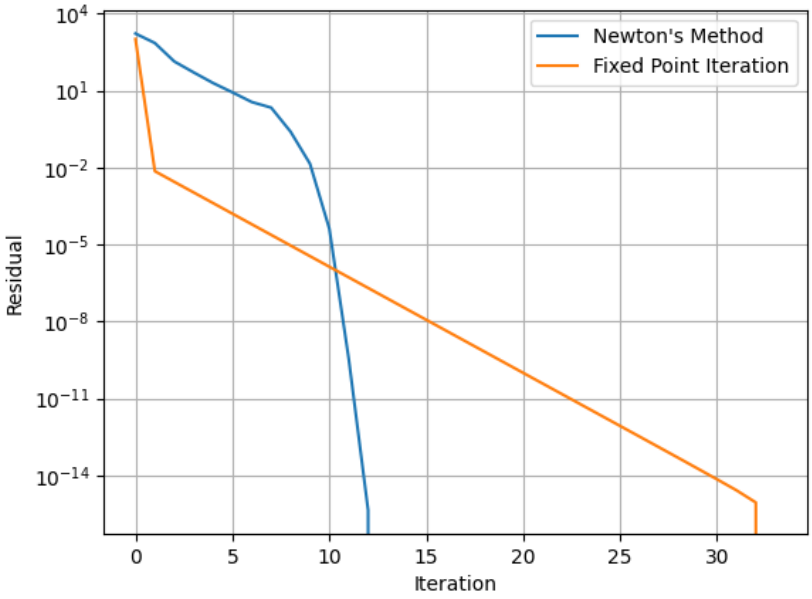

Figure 1: Iteartion vs. Residual

Figure 1 illustrates the the iteration vs. residual plot for both Newton's Method and Fixed Point Iteration. We could observe clearly from this picture that Newton's method carries a quadratic rate of convergence while Fixed-Point Iteration carries a linear rate of convergence. As a result, Newton's method converged in just half of the iterations taken by Fixed Point Method.

1.4 Numerical Stability

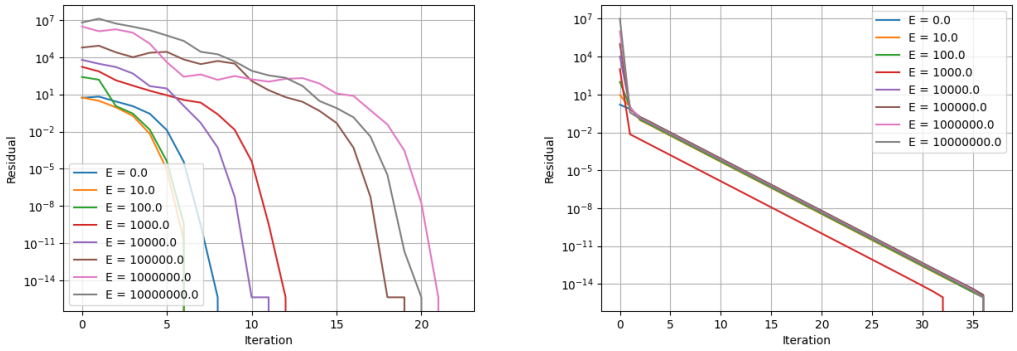

Figure 2: Convergence plot for both Newton's Method and Fixed Point iteration, as a function of E_0

Figure 2 illustrates the convergence plot for both Newton's method and Fixed point iteration with respect to different E_0. For Newton's method, we observe as we place the initial starting point further, the number of iteration it requires to converge is larger, which makes intuitive sense. For fixed-point iteration, the number of iterations it took to converge is almost the same regardless of starting position. This could be attributed to the linear rate of convergence for the fixed point iteration. Note that at the furthest point, Newton's method still converged sizeably faster

than fixed-point iteration. Both method did not exhibit numerical instabilities through out the calculation.

1.5 E vs. M

We use Newton's method for this investigation.

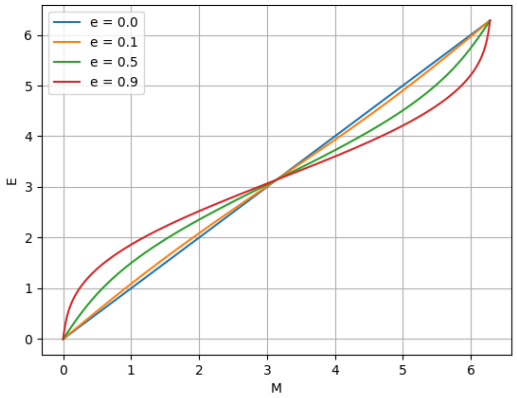

Figure 3: E vs. M for different e

With increasing eccentricity e, we observe a more curvature relationship between the eccentricity anomaly E and mean anomaly M. This is due to the sin term within the Kepler's equation. As e increases, the sin term adds more of a non-linear effect on the relation. M (mean anomaly) represents an average or uniform motion as if the planet moved around its orbit at a constant speed. It's the position the planet would be in if its orbit were perfectly circular. E (eccentric anomaly) represents the actual, non-uniform motion of the planet around its elliptical orbit. As eccentricity increases, the difference between this idealized uniform motion (mean anomaly) and the actual position (eccentric anomaly) becomes more pronounced, causing the curve.

1.6 Numerical Noise

Figure 4 illustrates the numerical noise of the model. By perturbing our solver at an extremely small region $M = \pi/2 \pm 10^{-6}$, and adding random noises to our starting location, we find that our numerical solver (Newton's Method), carries an inherent noise that is around $0.2E - 6$ in magnitude.

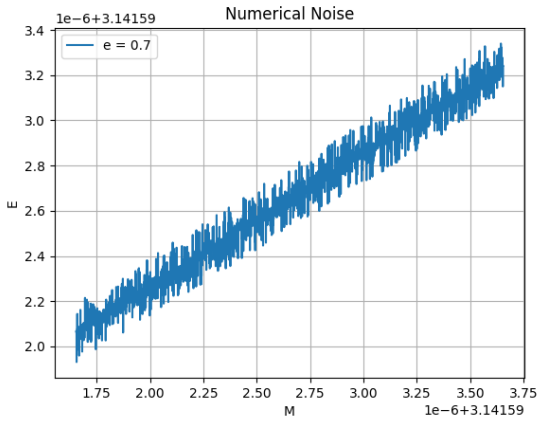

Figure 4: Numerical Noise

2 Analytic Critical Points

We are given the following equation:

$$f(x_1, x_2) = x_1^4 + 3x_1^3 + 3x_2^2 - 6x_1x_2 - 2x_2 \tag{8}$$

To find the critical points analytically, we take the partial derivative with respect to both x_1 and x_2.

$$\begin{aligned} \frac{\partial f(x_1, x_2)}{\partial x_1} &= 4x_1^3 + 9x_1^2 - 6x_2 = 0 \\ \frac{\partial f(x_1, x_2)}{\partial x_2} &= 6x_2 - 6x_1 - 2 = 0 \end{aligned} \tag{9}$$

Solving this system of equation of MATLAB, we obtain the following set of critical points:

x_1	x_2
-0.25	0.0833
-2.7321	-2.3987
0.7321	1.0654

Table 1: List of critical points

To evaluate them, we substitute the pairs into $f(x_1, x_2)$, we obtain the following:

$$\begin{aligned} f(-0.25, 0.0833) &= -0.0638 \\ f(-2.7321, -2.3987) &= -22.7256 \\ f(0.7321, 1.0654) &= -1.9410 \end{aligned} \tag{10}$$

We this, we then make the following classification:

x_1	x_2	Type
-0.25	0.0833	Local Minimum
-2.7321	-2.3987	Global Minimum
0.7321	1.0654	Local Minimum

Table 2: List of minimums and their classification

3 Line Search

3.1 Two Line Search Algorithm

We implement two line search algorithms. First, to simplify notation for line search, we use the following:

$$\phi(a) = f(x_k + \alpha p_k) \tag{11}$$

Via various substitution, we also arrive at the following derivative:

$$\phi'(a) = \nabla f(x_k + \alpha p_k)^T p_k \tag{12}$$

where p_k is the descent direction. Note that p_k ALWAYS have to be in descent, therefore $\sigma(0)$ is ALWAYS negative.

3.1.1 Backtracking

The simplest line search algorithm relies on the *sufficient decrease condition* combined with a backtracking algorithm. The sufficient decrease condition, or sometimes known as the *Armijo condition*, is given by the inequality:

$$\phi(\alpha) \leq \phi(0) + \mu_1 \alpha \phi'(0) \tag{13}$$

where μ_1 is a constant between $0 < \mu_1 < 1$. The quality $\alpha\phi'$ represents the expected decrease of the function, *assuming* the function continues the current initial slope. As long as we achieve a small fraction of the expected decrease, tuned by μ_1, we then satisfy the condition and accept the current step size α.

We attempt to code this algorithm in `Python` and try it on the following function:

$$f(x_1, x_2) = 0.1x_1^6 - 1.5x_1^4 + 5x_1^2 + 0.1x_2^4 + 3x_2^2 - 9x_2 + 0.5x_1x_2 \tag{14}$$

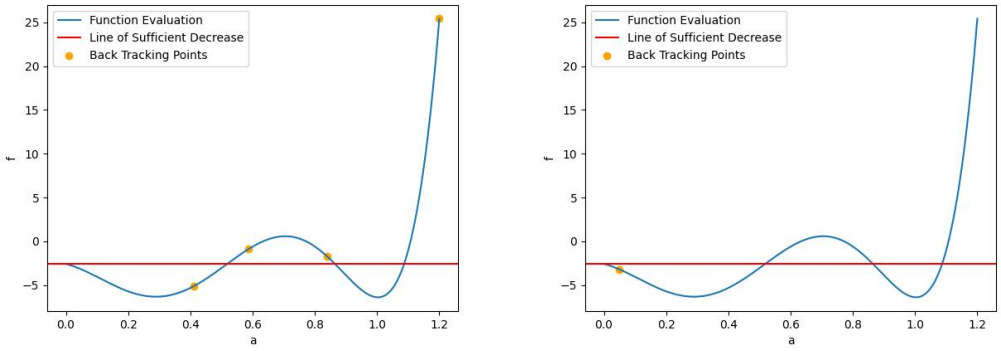

Figure 5: Backtrack Line Search with $a_0 = 1.2$ (Left) and $a_0 = 0.05$ (Right)

The above figure reproduces the result from Example 4.8 from the MDO textbook. Note that we start at $a_0 = 1.2$ and move down until our point reaches below the line of sufficient decrease. Note that smaller values of a_0 converged the line search instantly, but did not find the optimal point within the line. This result matches exactly with Example 4.8.

3.1.2 Bracketing and Pinpoint

We now code up a second algorithm that finds the minimum differently. A major weakness of the sufficient decrease condition is that it accepts small steps that marginally decrease the objective function because μ_1 is typically quite small. We now introduce a different condition, the *sufficient curvature condition*:

$$|\phi'(a)| \leq \mu_2 |\phi'(0)| \tag{15}$$

That is, the slope at the candidate point should be lower in magnitude than the slope at the start of line search by a factor of μ_2. Combining these two conditions, we obtain the *Strong Wolfe Condition*. For this question, we implement the line search algorithm introduced by More and Tuente, which contains two phases:

- The *bracketing* finds an interval within which we are certain to find a point that satisfies the strong Wolfe conditions
- The *pinpointing* phase finds a point that satisfies the strong Wolfe conditions within the interval provided by the bracketing phase

The detailed algorithms are introduced in the MDO textbook. We attempt to code this algorithm in `Python`.

We choose quadratic interpolation as the point selection algorithm, which we could analyitcally obtain the step size via:

$$\alpha^* = \frac{2\alpha_1 \left[\phi(\alpha_2) - \phi(\alpha_1)\right] + \phi'(\alpha_1)\left(\alpha_1^2 - \alpha_2^2\right)}{2\left[\phi(\alpha_2) - \phi(\alpha_1) + \phi'(\alpha_1)(\alpha_1 - \alpha_2)\right]} \tag{16}$$

Figure 6 illustrates the result for Example 4.8 using the bracketing and pinpointing algorithm. Note that the bracketing algorithm gave a way better minimum than the original backtracking algorithm thanks to the *Strong Wolfe Condition*. This result also matches Example 4.9 exactly.

By tuning the slope constraint μ_2, we could force the solver to pick a point with flatter slope, and hence a better optimum. For example, if we change from $\mu_2 = 0.9$ to $\mu_2 = 0.2$, we have a better placed optimum:

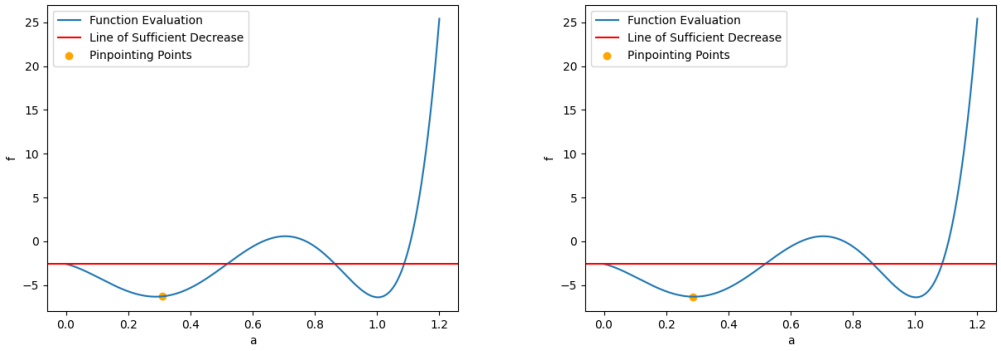

Figure 6: Line Search Bracketing and Pinpoint. $\mu = 0.9$(Left) and $\mu = 0.2$(Right)

Notice the slight change in position due to the slope constraint. Through numerical experiment, we are able to constrain the slope all the way down to $\mu_2 = 0.01$.

3.2 Bean Function

We apply the backtracking and bracketing/pinpointing algorithm to the following bean function:

$$f(x_1, x_2) = (1 - x_1)^2 + (1 - x_2)^2 + \frac{1}{2}\left(2x_2 - x_1^2\right)^2 \tag{17}$$

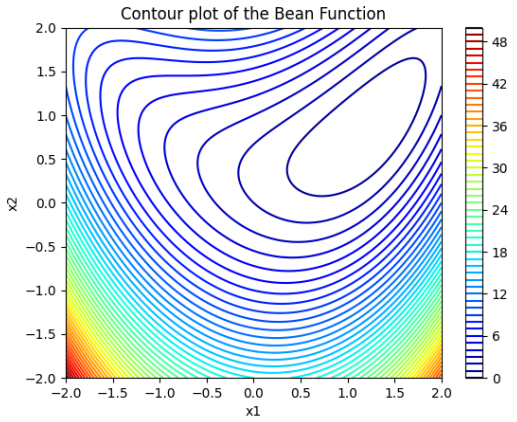

Figure 7: Example 4.11 Bean Function

Figure 7 illustrates the bean function in contour plot. We are going to pick some random starting point and descent directions to test out our Line Search algorithm. For ease of illustrations, we will take 1D slices of this 2D contour plot to visualize if our algorithms have picked the optimal minimum within the line.

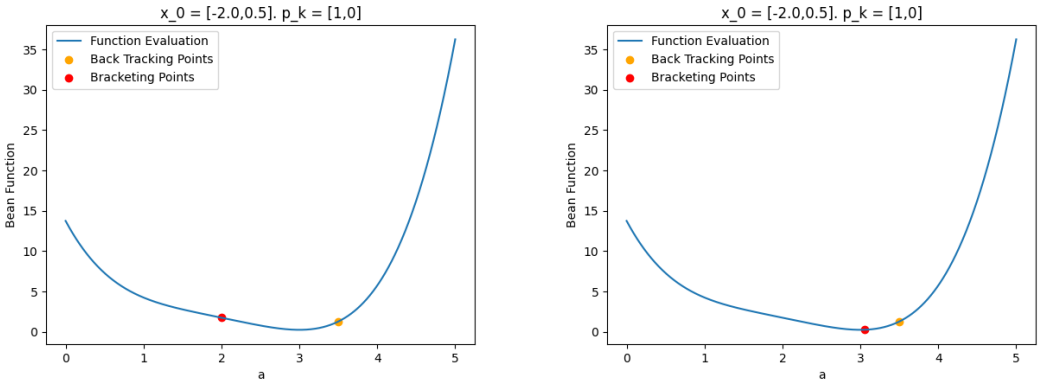

Figure 8: Bean function with $\mu_2 = 0.9$ (Left) and $\mu = 0.1$(Right)

Figure 8 illustrates us starting from $x_0 = [-2.0, 0.5]$ and looking at descent direction $p_k = [1, 0]$, which is complete east. We notice that with $\mu_2 = 0.9$, the slope tolerance was so loose that the optimizer picked a point that is far from the minimum. By tightening the constraint μ_2 to 0.1, we are able to pick a point that is readily at the minimum of the descent direction. However, over-tightening the constraint may cause the optimizer to stall as it could not find a point that satisfies the constraint, especially in this case where the minimum is rather *sharp*, where the region that satisfies the tolerance is relatively small. Therefore, it may take the optimizer a while to converge the optimal point. In reality, the point given by $\mu = 0.1$ is already extremely close to the minimum that we could readily deemed as "converged".

We now start from a different point with different descent direction where $x_0 = [0, -2.0]$ and $p_k = [0, 1]$, and we start to vary ρ and observe the effect on the backtracking algorithm.

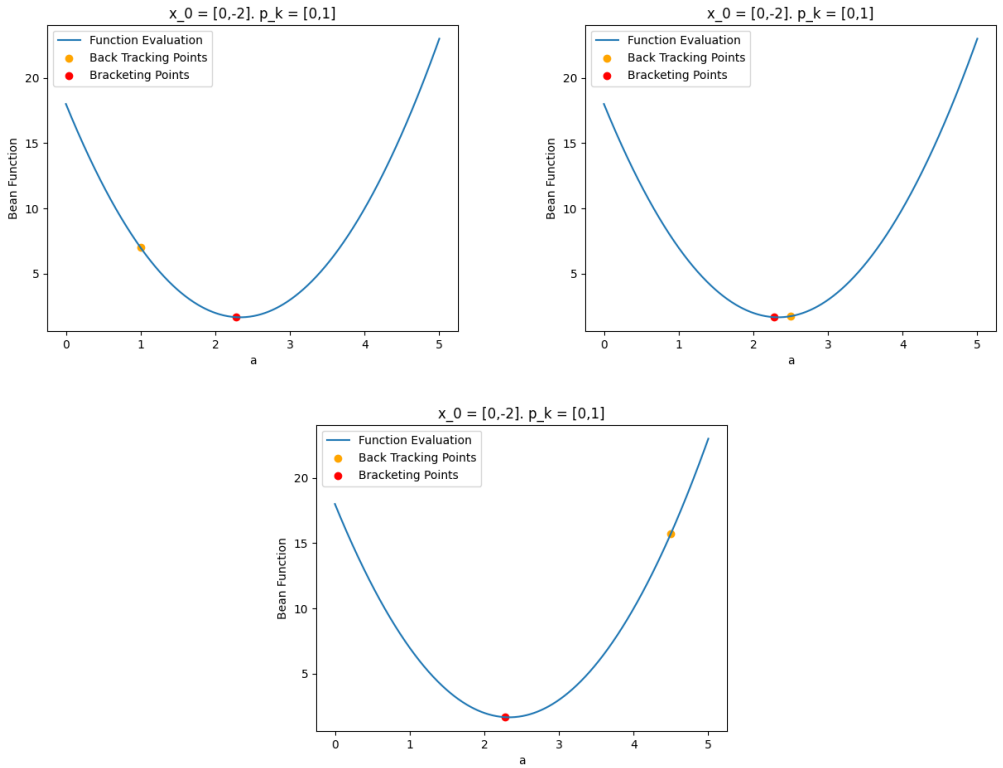

Figure 9: (Top Left) $\rho = 0.2$, (Top Right) $\rho = 0.5$, (Bottom) $\rho = 0.9$

We now see the disadvantage of the backtracking algorithm. As long as the slope of sufficient descent is satisfied, the resulting output minimum depends strictly on ρ. In this case, both $\rho = 0.2$

and $\rho = 0.9$ undershoot and overshoots the minimum. $\rho = 0.5$ arrives near the minimum and that is via numerical experimentation. In a system with a clear optimality like this one, we could increase μ_1 manually. However, this still requires manual tuning and could have all sorts of issues mentioned in the previous sections such as taking too small of a step etc. Looking at the Line search involving bracketing and pinpointing, the algorithm always located the optimal point along the line regardless. We finish off this section by starting at the same point but along a diagonal direction $p_k = [1, 1]$, with default $\rho = 0.7$.

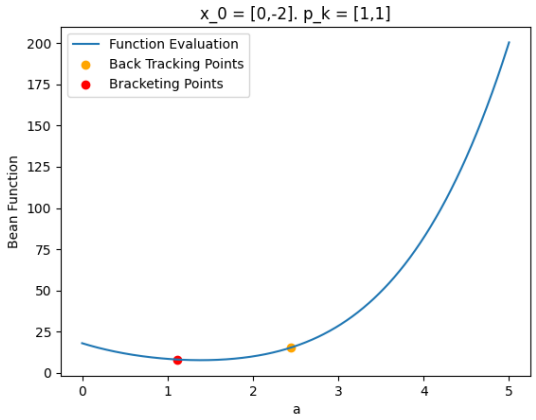

Figure 10: Line search with $x_0 = [0, -2.0]$ and $p_k = [1, 1]$

Figure 10 illustrates the new line search. We observe that the bracketing algorithm is robust that it almost always identified a more optimal minimum thanks to its stronger constraint.

3.3 Rosenbrock Function

3.3.1 2D Rosenbrock Function

The 2D Rosenbrock function is given as the following:

$$f(x, y) = (1 - x)^2 + 100(y - x^2)^2 \tag{18}$$

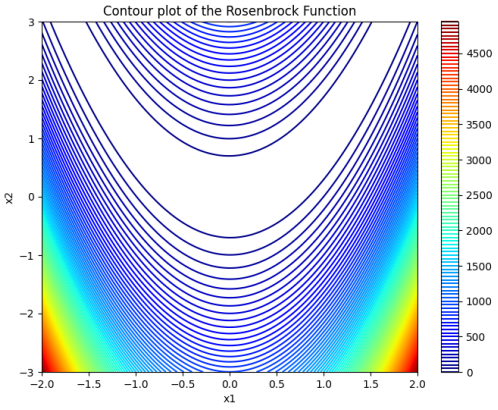

Figure 11: Contour of Rosenbrock Function

Assuming we start at $x_0 = [0, -3]$ with descent direction $p_k = [0, 1]$. We obtain the following optimal point via line search (Figure 12):

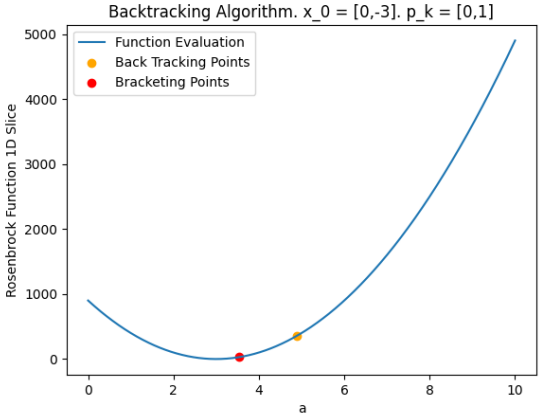

Figure 12: $\mu = 0.5$ and $\rho = 0.7$

We again observe the robustness of the bracketing algorithm. Comparing to the backtracking line search, it almost always outputs a more optimal minimum. We now try different starting point and directions (Figure 13)

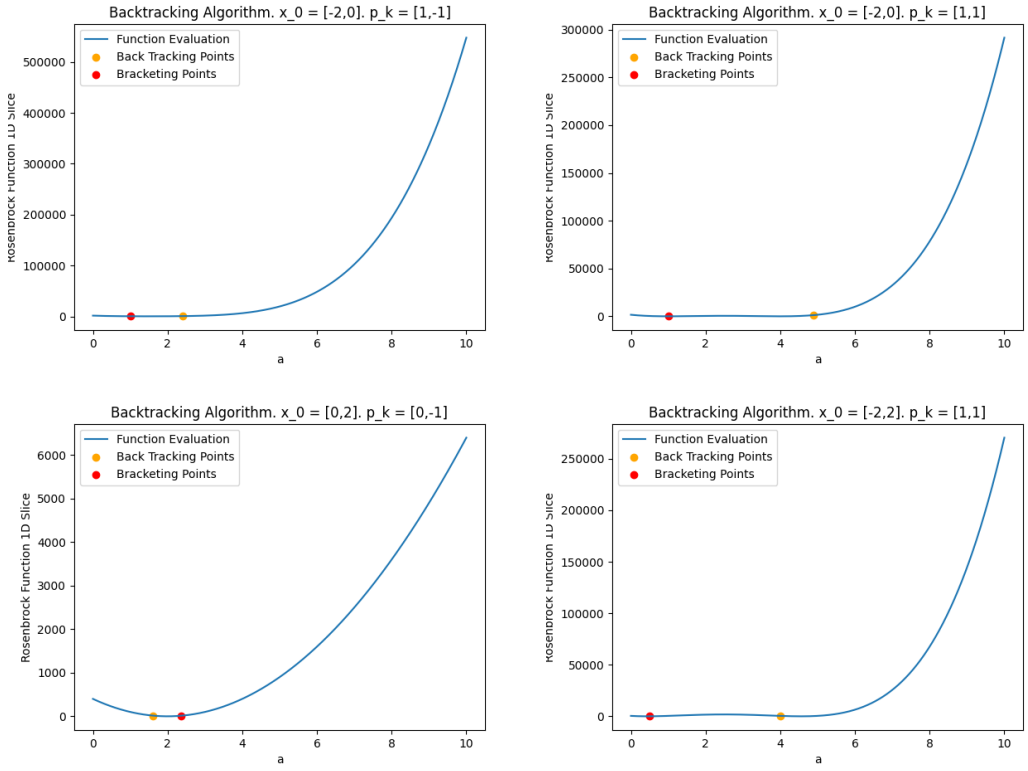

Figure 13: Different starting point for 2D Rosenbrock

We again note the robustness of the bracketing algorithm. From numerical experiments, we have discovered that generally $\mu_2 \approx 0.2$ and $\rho \approx 0.4$ would provide decent optimal for the bracketing and backtracking line search respectively.

3.3.2 N-Dimensional Rosenbrock

The N-dimensional Rosenbrock is defined via the following summation:

$$\sum_{i=1}^{N-1} \left[100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2\right] \tag{19}$$

where $N = 6$. Due to our inability to visualize the N-dimensional Rosenbrock, it is rather difficult to determine the direction of descent. We guess a bunch of initial positions and directions, and obtain the following plots:

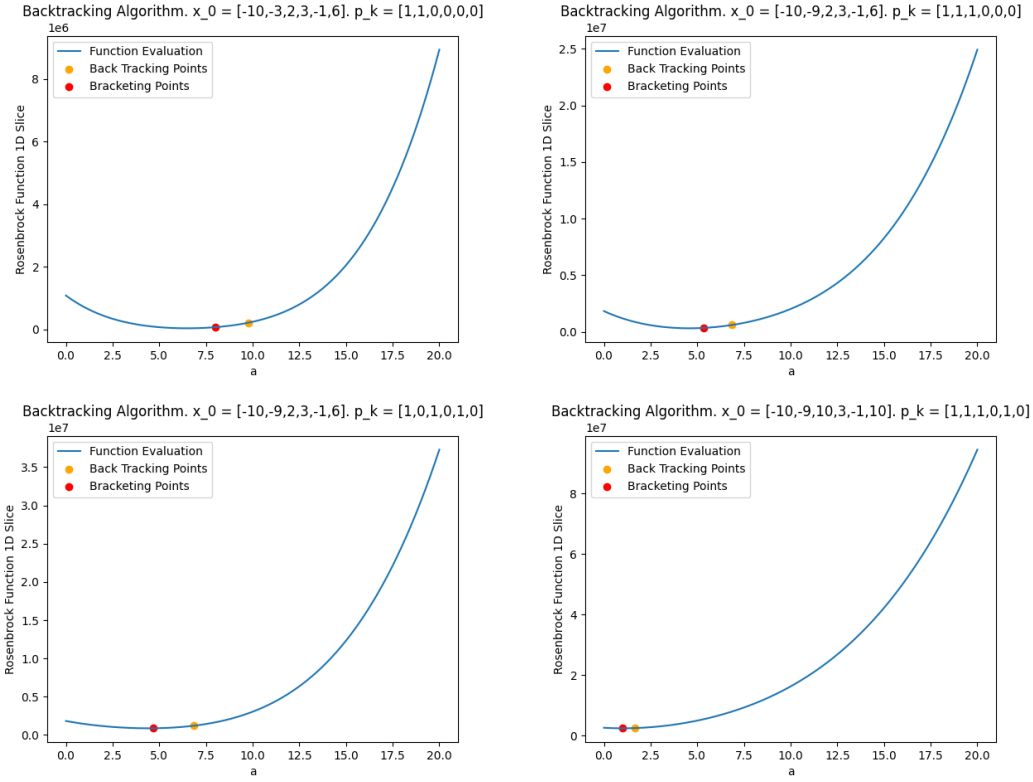

Figure 14: Rosenbrock 6D optimization 1D slice with different starting point and descent directions

We again note that the bracketing algorithm outputs a better minimum along the line than the backtracking algorithm. For this problem, general tuned μ_2 and ρ would be $\mu_2 \approx 0.2$ and $\rho \approx 0.5$ respectively. We have already discussed the effects of tuning μ and ρ in the last section! In essence μ_2 controls the tightness of the slope constraint. If low, the algorithm will find a slope that is close to the minimum ($f' \approx 0$), but may cause longer iterations. If high, the algorithm will converge faster but may not find the best optimal point. ρ on the other hand dictates the fraction at which the algorithm should backtrack if not crossed the line of sufficient decrease. The issue with the backtracking algorithm is that once the line of sufficient decrease is reached it is deemed as "converged". However, this is non-optimal as it often does not guarantee the best minimum. We could tune the line of sufficient decrease via μ_1 but this may cause stepping issues later on. Therefore, we almost always observe that the bracketing/pinpointing algorithm finding a better minimum than the backtracking line search, regardless of dimensions.