Module 4B - Gradient-based Optimization

Omar Betancourt, Payton Goodrich, Emre Mengi

July 24, 2021

Contents

Objectives: To learn how to optimize a non-complex system by formulating an objective function and minimizing it.
Prerequisite Knowledge: N/A
Prerequisite Modules: 1A - Calculus, 1B - Linear Algebra, 1D - Differential Equations, 3C - Generic Time Stepping
Difficulty: Easy
Summary: In this module, you will learn the fundamentals of optimization. You will learn how to describe a system in terms of an objective function and how to apply Newton's Method to find a minimum solution to the system.

1 Theory

1.1 Objective Function

Objective functions - also referred to here as cost functions - are functions that return a scalar value that are used to determine the 'fitness' of whatever it is that you are trying to optimize. Generally, the fittest solution has the smallest cost function.
Let's begin with an example. Consider if one were trying to minimize the cost of fencing materials to enclose a 400-acre rectangle for cattle ranching. Because fencing is priced per linear foot, one would want to minimize the total length of the perimeter. In this case, the cost function would be:

$$\Pi(L) = w_1 L \tag{1.1}$$

where w_1 is a weighting factor, in this case the dollar-cost per linear foot of fence. As you learned in the calculus module or previous courses, in order to find minimums of a function, you take the derivative and set it equal to zero. Thus, this example can be solved analytically:

$$\begin{aligned}
L &= 2(x+y) \\
A &= x \cdot y \\
y &= \frac{A}{x} \\
\Pi = w_1 L &= 2w_1(x + \frac{A}{x}) \\
\Pi_{Min} &= \nabla\Pi \to 0 \\
\nabla\Pi = \frac{\partial}{\partial x} 2w_1(x + \frac{A}{x}) &= 0 \\
2w_1(1 + \frac{A}{x^2}) = 0 &\to x = \sqrt{(}A) \\
y = \frac{A}{x} &= \sqrt{(}A) \\
L = 2(x+y) &= 2 \cdot A \\
\Pi &= 2w_1 A
\end{aligned}$$

Assuming a cost of \$2 per linear foot of fencing, $w_1 = 2$, the cost function Π is plotted as a function of the length of one of the sides of the rectangular area, x, in Figure 1.1.

While a single-variable cost function like the one in this example is relatively easy to solve, they can quickly become more complex as more variables and conditions are introduced. Let's consider again trying to minimize the cost of fencing materials to enclose a 400 acre ranch. However, one of the sides of the rectangle is next to a road, and requires special fencing material valued at \$5 per foot. Now the cost function would become:

$$\Pi = 2w_1 x + w_1 y + w_2 y = w_1 \left(2x + \frac{A_d}{x}\right) + w_2 \frac{A_d}{x}$$

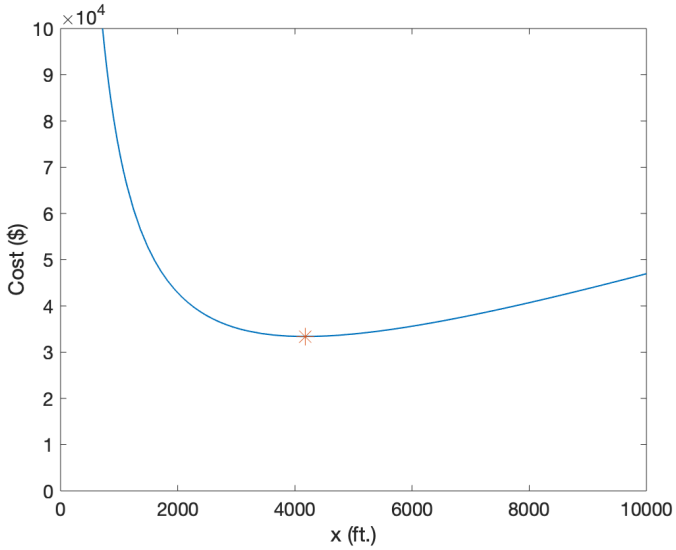

Figure 1.1: The cost of fencing material to enclose an a 400-foot area as a function of the side-length, x. A minimum value of \$33,390 is reached at $x = 4173.73$ ft.

This cost function can still be solved analytically, though it will be more difficult. What if, however, we were to find that because of a limitation in the fencing equipment, it will cost an extra \$0.01 every 100 feet that fencing material is placed away from the road. Or, what if we were to find that there is a pond in the way of our fencing path that will require the fencing to make a path around it?

In the fencing example, the cost function was minimizing a single design variable - the dollar cost of fencing. However, it should not be confused that cost functions necessarily minimize a dollar cost, nor should it be confused that all cost functions have a single objective of the cost function. Many cost functions will maximize, minimize, and tune multiple design variables simultaneously, and the degree of importance of each design variable is adjusted by the weighting term, w. The larger the value of a weighting term is compared to the other weighting terms, then the greater the influence that design variable will have on the cost function. Consider now a scientist is trying to optimize the traits of a new apple species. This cost function may include multiple related and unrelated factors to optimize for. They may want to maximize the size of the apple, sweetness, yield, climate resilience, disease resistance. Conversely, it would be beneficial to minimize the fertilizer use, number of seeds, etc. Each of these different goals or targets are called design variables. We then define $\mathbf{\Lambda}$, a vector of N design variables such that:

$$\Pi(\mathbf{\Lambda}) = \Pi\{\Lambda_1, ..., \Lambda_N\} \tag{1.2}$$

Each of these design variables Λ_i may be written such that it's contribution to the overall cost function is minimized when the design variable is maximized, minimized, approaches a desired value, or fits within a zone of acceptable values. Consider our apple genetics example from before. Using the yield of apples as a design variable, to minimize the value of the cost function as the yield increases, we can define the design variable like so:

$$\Lambda_{yield} = \frac{w_{yield}}{N_{Apples}} \tag{1.3}$$

Thus, as the number of apples increases, the value of Λ_{yield} would decrease, thereby minimizing $\Pi(\mathbf{\Lambda})$. Now consider a design variable that we would want to minimize, such as the average volume of fertilizer needed to keep an apple tree healthy. A viable design variable term for this would be:

$$\Lambda_{fertilizer} = w_{fertilizer} V_{fertilizer} \tag{1.4}$$

In this case, $\Lambda_{fertilizer}$ decreases as the volume decreases, thereby minimizing $\Pi(\mathbf{\Lambda})$. Another type of design variable term that we can use in an objective function is to tune a variable to a pre-determined value. Continuing with our apple genetics example, one might want to tune the size of an apple to be large, but not so large as to not fit into an average persons hand. In this case, let's assume that after some market research and user studies, the team in charge of designing this new breed of apple have decided that the perfect apple has a radius of 10 cm. We can then write the design variable Λ_{size} as:

$$\Lambda_{size} = w_{size} \frac{||r_{actual} - r_{desired}||}{||r_{desired}||} \tag{1.5}$$

In this example, we are normalizing the magnitude of the difference between the actual and the desired size values to ensure a positive value. A final consideration is the inclusion of step-wise, active-inactive constraints that 'turn on' when the design variable is above and/or below threshold values. These types of design variables constrains the solution to a zone of interest by having a variable weighting term. For example, it would be desirable to have the apples fully ripen sometime in October because of the availability of fruit pickers and to take advantage of the autumn orchard tourists. In this case, we can write a term for the design variable as:

$$\Lambda_{harvest} = \hat{w}_{harvest} \frac{||t_{harvest} - TOL_{harvest}||}{||TOL_{harvest}||} \tag{1.6}$$

where:

$$\hat{w}_{harvest} = \begin{cases} 0 & \text{for } -TOL_{harvest} \leq t_{harvest} \leq TOL_{harvest} \\ w_{harvest} & \text{otherwise} \end{cases} \tag{1.7}$$

and $TOL_{harvest}$ is the 'tolerance', i.e. the acceptable range of the zone. In this case, as soon as $t_{harvest}$ falls outside of the acceptable range, $\Lambda_{harvest}$ will have a step-wise increase from 0 to the weighted value of the term.

1.2 Limitations of Gradient Methods

Cost functions are often nonconvex in design parameter space and often nonsmooth due to the variety of design variables. Furthermore, they are not necessarily differentiable because of active-inactive constraints. Finally, sample size effects can induce a degree of stochastic behavior in the objective function.

In many cases, the minimization of a cost function is difficult with direct application of gradient methods. This motivates nonderivative search methods, for example those found in Machine Learning Algorithms (MLA's). One of the most basic subset of MLA's are so-called Genetic Algorithms (GA's). Typically, one will use a GA first in order to isolate multiple local minima, and then use a gradient based algorithm in these locally convex regions or reset the GA to concentrate its search over these more constrained regions. GA's are typically the simplest scheme to start the analysis, and one can, of course, use more sophisticated methods if warranted.

1.3 Gradient-based methods

One can apply a gradient-based method, if the objective function is sufficiently smooth in that region of the parameter space. In other words, if one has located a convex portion of the parameter space with a global genetic search (i.e. a genetic algorithm which is covered in another module) one can employ gradient-based procedures locally to minimize the objective function further, since the are generally much more efficient for convex optimization of smooth functions. In such procedures, in order to obtain a new directional step for $\mathbf{\Lambda}$, one must solve the following system:

$$[\mathbb{H}]\{\Delta\mathbf{\Lambda}\} = -\{\boldsymbol{g}\}, \tag{1.8}$$

where $[\boldsymbol{\mathcal{H}}]$ is the Hessian matrix $(N \times N)$, $\{\Delta\boldsymbol{\Lambda}\}$ is the parameter increment $(N \times 1)$, and $\{\boldsymbol{g}\}$ is the gradient $(N \times 1)$. We shall not employ this second (post-genetic) stage in this work. Specifically, this is determined by forcing the gradient of $\nabla_{\boldsymbol{\Lambda}}\Pi(\boldsymbol{\Lambda}) = \mathbf{0}$. Expanding (linearizing) around a first guess $\boldsymbol{\Lambda}^i$ yields:

$$\nabla_{\boldsymbol{\Lambda}}\Pi(\boldsymbol{\Lambda}^{i+1}) \approx \nabla_{\boldsymbol{\Lambda}}\Pi(\boldsymbol{\Lambda}^i) + \nabla\left(\nabla_{\boldsymbol{\Lambda}}\Pi(\boldsymbol{\Lambda}^i)\right)\cdot(\boldsymbol{\Lambda}^{i+1} - \boldsymbol{\Lambda}^i) + \textit{higher order terms} \approx \mathbf{0} \tag{1.9}$$

or in more streamlined matrix notation, defining the Hessian, $[\boldsymbol{\mathcal{H}}] = \nabla\left(\nabla_{\boldsymbol{\Lambda}}\Pi(\boldsymbol{\Lambda})\right)$ and $\{\boldsymbol{g}\} = \nabla_{\boldsymbol{\Lambda}}\Pi(\boldsymbol{\Lambda})$, thus

$$[\boldsymbol{\mathcal{H}}]\{\Delta\boldsymbol{\Lambda}\} + \{\boldsymbol{g}\} = \mathbf{0}. \tag{1.10}$$

Following a standard Newton-type multivariate search, a new design increment is computed,

$$\Delta = (\Delta\Lambda_1, \Delta\Lambda_2, ...\Delta\Lambda_N), \tag{1.11}$$

for a design vector, $\boldsymbol{\Lambda}$, by solving the following system, $[\boldsymbol{\mathcal{H}}]\{\Delta\boldsymbol{\Lambda}\} = -\{\boldsymbol{g}\}$, where $[\boldsymbol{\mathcal{H}}]$ is the Hessian matrix $(N \times N)$, with components

$$H_{ij} = \frac{\partial^2\Pi(\boldsymbol{\Lambda})}{\partial\Lambda_i\partial\Lambda_j}, \tag{1.12}$$

$\{\boldsymbol{g}\}$ is the gradient $(N \times 1)$, with components

$$g_i = \frac{\partial\Pi(\boldsymbol{\Lambda})}{\partial\Lambda_i} \tag{1.13}$$

and where $\{\Delta\boldsymbol{\Lambda}\}$ is the design increment $(N \times 1)$, with components $\Delta\Lambda_i$. After the design increment has been solved for, one then forms an updated design vector, $\boldsymbol{\Lambda}^{new} = \boldsymbol{\Lambda}^{old} + \Delta\boldsymbol{\Lambda}$, and the process is repeated until $||\Pi|| \leq TOL$. Explicitly, the incremental system is

$$\begin{bmatrix} \frac{\partial^2\Pi(\Lambda)}{\partial\Lambda_1\partial\Lambda_1} & \frac{\partial^2\Pi(\Lambda)}{\partial\Lambda_1\partial\Lambda_2} & \frac{\partial^2\Pi(\Lambda)}{\partial\Lambda_1\partial\Lambda_3} & \frac{\partial^2\Pi(\Lambda)}{\partial\Lambda_1\partial\Lambda_4} & ..... \\ \frac{\partial^2\Pi(\Lambda)}{\partial\Lambda_2\partial\Lambda_1} & \frac{\partial^2\Pi(\Lambda)}{\partial\Lambda_2\partial\Lambda_2} & \frac{\partial^2\Pi(\Lambda)}{\partial\Lambda_2\partial\Lambda_3} & \frac{\partial^2\Pi(\Lambda)}{\partial\Lambda_2\partial\Lambda_4} & ..... \\ \frac{\partial^2\Pi(\Lambda)}{\partial\Lambda_3\partial\Lambda_1} & \frac{\partial^2\Pi(\Lambda)}{\partial\Lambda_3\partial\Lambda_2} & \frac{\partial^2\Pi(\Lambda)}{\partial\Lambda_3\partial\Lambda_3} & \frac{\partial^2\Pi(\Lambda)}{\partial\Lambda_3\partial\Lambda_4} & ..... \\ \frac{\partial^2\Pi(\Lambda)}{\partial\Lambda_4\partial\Lambda_1} & \frac{\partial^2\Pi(\Lambda)}{\partial\Lambda_4\partial\Lambda_2} & \frac{\partial^2\Pi(\Lambda)}{\partial\Lambda_4\partial\Lambda_3} & \frac{\partial^2\Pi(\Lambda)}{\partial\Lambda_4\partial\Lambda_4} & ..... \\ ..... & ..... & ..... & ..... & ..... \\ ..... & ..... & ..... & ..... & ..... \\ \frac{\partial^2\Pi(\Lambda)}{\partial\Lambda_N\partial\Lambda_1} & \frac{\partial^2\Pi(\Lambda)}{\partial\Lambda_N\partial\Lambda_2} & \frac{\partial^2\Pi(\Lambda)}{\partial\Lambda_N\partial\Lambda_3} & \frac{\partial^2\Pi(\Lambda)}{\partial\Lambda_N\partial\Lambda_4} & ..... \end{bmatrix} \begin{Bmatrix} \Delta\Lambda_1 \\ \Delta\Lambda_2 \\ \Delta\Lambda_3 \\ \Delta\Lambda_4 \\ ..... \\ ..... \\ \Delta\Lambda_N \end{Bmatrix} = -\begin{Bmatrix} \frac{\partial\Pi(\Lambda)}{\partial\Lambda_1} \\ \frac{\partial\Pi(\Lambda)}{\partial\Lambda_2} \\ \frac{\partial\Pi(\Lambda)}{\partial\Lambda_3} \\ \frac{\partial\Pi(\Lambda)}{\partial\Lambda_4} \\ ..... \\ ..... \\ \frac{\partial\Pi(\Lambda)}{\partial\Lambda_N} \end{Bmatrix}. \tag{1.14}$$

The derivatives must often be computed numerically:

- For the first derivative of Π at $(\Lambda_1, \Lambda_2, \Lambda_3)$:

$$\frac{\partial\Pi}{\partial\Lambda_1} \approx \frac{\Pi(\Lambda_1 + \Delta\Lambda_1, \Lambda_2, \Lambda_3) - \Pi(\Lambda_1 - \Delta\Lambda_1, \Lambda_2, \Lambda_3)}{2\Delta\Lambda_1} \tag{1.15}$$

- For the second derivative at $(\Lambda_1, \Lambda_2, \Lambda_3)$:

$$\begin{aligned} \frac{\partial}{\partial\Lambda_1}\left(\frac{\partial\Pi}{\partial\Lambda_1}\right) &\approx \frac{\left(\frac{\partial\Pi}{\partial\Lambda_1}\right)|_{\Lambda_1 + \frac{\Delta\Lambda_1}{2}, \Lambda_2, \Lambda_3} - \left(\frac{\partial\Pi}{\partial\Lambda_1}\right)|_{\Lambda_1 - \frac{\Delta\Lambda_1}{2}, \Lambda_2, \Lambda_3}}{\Delta\Lambda_1} \\ &= \frac{1}{\Delta\Lambda_1}\Big(\Big(\frac{\Pi(\Lambda_1 + \Delta\Lambda_1), \Lambda_2, \Lambda_3 - \Pi(\Lambda_1, \Lambda_2, \Lambda_3)}{\Delta\Lambda_1}\Big) \\ &- \Big(\frac{\Pi(\Lambda_1, \Lambda_2, \Lambda_3) - \Pi(\Lambda_1 - \Delta\Lambda_1, \Lambda_2, \Lambda_3)}{\Delta\Lambda_1}\Big)\Big). \end{aligned} \tag{1.16}$$

- For the cross-derivative at (Λ_1, Λ_2):

$$\begin{aligned}\frac{\partial}{\partial\Lambda_2}\left(\frac{\partial\Pi}{\partial\Lambda_1}\right) &\approx \frac{\partial}{\partial\Lambda_2}\left(\frac{\Pi(\Lambda_1+\Delta\Lambda_1,\Lambda_2,\Lambda_3)-\Pi(\Lambda_1-\Delta\Lambda_1,\Lambda_2,\Lambda_3)}{2\Delta\Lambda_1}\right)\\ &\approx \frac{1}{4\Delta\Lambda_1\Delta\Lambda_2}\left(\Pi(\Lambda_1+\Delta\Lambda_1,\Lambda_2+\Delta\Lambda_2,\Lambda_3)-\Pi(\Lambda_1-\Delta\Lambda_1,\Lambda_2+\Delta\Lambda_2,\Lambda_3)\right)\\ &- \left(\Pi(\Lambda_1+\Delta\Lambda_1,\Lambda_2-\Delta\Lambda_2,\Lambda_3)-\Pi(\Lambda_1-\Delta\Lambda_1,\Lambda_2-\Delta\Lambda_2,\Lambda_3)\right). \end{aligned} \tag{1.17}$$

The overall numerical strategy for solving a cost function using a gradient-based method, i.e. Newton's Method, is shown in Figure 1.3.

2 Example

Apply Newton's Method to the equation $f(x) = x^3 + x5 = 0$. Begin with the given initial guess, $x_0 = 0$, and find x_1 and x_2.
First, we find the gradient of f:

$$\nabla f = \frac{\partial}{\partial x}(f) = 3x^2 + 1 \tag{2.1}$$

Second, we apply Newton's method and linearize around the first guess, x_0:

$$x_{n+1} = x_n + \frac{f(x_n)}{f'(x_n)} = x_n + \frac{x_n^3 + x5 = 0}{3x_n^2 + 1} \tag{2.2}$$

$$\ldots \tag{2.3}$$

$$x_{n+1} = \frac{2x_n^3 + 5}{3x_n^2 + 1} \tag{2.4}$$

Let $x_0 = 0$ to obtain:

$$x_1 = \frac{2x_0^3 + 5}{3x_0^2 + 1} = \frac{2(0)^3 + 5}{3(0)^2 + 1} = 5 \tag{2.5}$$

and:

$$x_2 = \frac{2x_1^3 + 5}{3x_1^2 + 1} = \frac{2(5)^3 + 5}{3(5)^2 + 1} = \frac{255}{76} \approx 3.355263158 \tag{2.6}$$

Repeating this process is simply done numerically, and can be repeated until the solution converges.

3 Assignment

In this assignment, you will use Newton's Method on a simple problem to demonstrate it's strengths and weaknesses.

The two objective functions to be minimized in this assignment are:

$$\Pi_a(x) = x^2 \tag{3.1}$$

$$\Pi_b(x) = \left(x + \frac{\pi}{2} sin(x)\right)^2 \tag{3.2}$$

NEWTON'S METHOD

1. Plot both objective functions on the same axes on the domain $-20 \leq x \leq 20$.

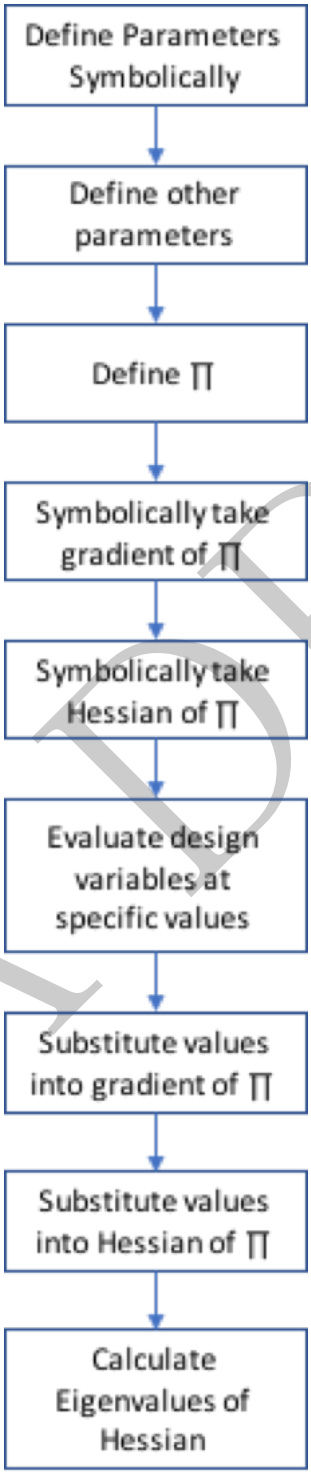

Figure 1.2: Flowchart of Newton's Method

2. Show (analytically) that both Π_a and Π_b have a unique global minimum at $x = 0$.
   Hint: one method is to expand Π_b, then use growth rates, bounds, and the basic behavior of the terms.

3. Show that, in the limit as $x \to \pm\infty$, Π_b has periodic local extrema and inflection points.

4. Explain succinctly why the presence of these local extrema and inflection points may make minimizing Π_b challenging using a gradient-based method.

5. Write a function to use Newton's method, with the syntax (if you are using Matlab):

   ```
   [sol, its, hist] = myNewton(f, df, x0, TOL, maxit)
   ```

   - `sol`: $M \times 1$ vector, the final value of the independent variable x.
   - `its`: scalar, the number of iterations performed.
   - `hist`: $M \times (its + 1)$ array, where `hist(:, i + 1)` $= x_i$.
   - `f`: $M \times 1$ anonymous function array, *gradient* of Π.
   - `df`: $M \times M$ anonymous function array, *Hessian* of Π.
   - `x0`: $M \times 1$ vector, initial guess to start iterations.
   - `TOL`: scalar, maximum allowable norm of `f(sol)`.
   - `maxit`: scalar, maximum allowable iterations.

   Include a printed copy of `myNewton()`. Your function should be less than 10 lines.

6. To use `myNewton()` to minimize Π_a and Π_b, define anonymous function arrays (1 by 1 arrays for now) of `f` $= \nabla\Pi$ and `df` $= \nabla^2\Pi$ for both Π_a and Π_b. You should do this using symbolic functions in your language of choice. If working with Matlab, use `diff()`, `hessian()`, and `matlabFunction()`, which converts symbolic functions to anonymous functions. You may also perform the necessary calculations by hand. Show your values for `f` and `df`.

7. Use `myNewton()` to minimize Π_a and Π_b, with `x0` $= 2 \times 10^k$, for $k \in \{-1, 0, 1\}$. Use `TOL` $= 10^{-8}$ and `maxit` $= 20$. Plot Π(`hist`) for Π_a and Π_b. Explain any significant results.

8. Find, through trial and error, the range of `x0` values for which Newton's method reaches the global minimum of Π_a and Π_b. Estimate `ng`, the number of equally-spaced points required as `x0` trials on the interval [-20, 20] to ensure Newton's method will find the global minimum of Π_b. Round `ng` up to the nearest power of 2.[1]

9. Write new versions of Π_b in 2D and 3D, defined as:

$$\Pi_{b,2}(x, y) = \Pi_b(x) + \Pi_b(y) \tag{3.3}$$

$$\Pi_{b,3}(x, y, z) = \Pi_b(x) + \Pi_b(y) + \Pi_b(z) \tag{3.4}$$

   Create function arrays[2] for the gradient and Hessian of $\Pi_{b,2}$ and $\Pi_{b,3}$. Show their values.

10. Attempt to minimize Π_b, $\Pi_{b,2}$, and $\Pi_{b,3}$ using a grid search in the search space $-20 \le x \le 20$, $-20 \le y \le 20$, $-20 \le z \le 20$. Use `linspace(-20, 20, ng)` for x, y, and z. Run each search several times (at least 10 but the more the better) and calculate the average run time. How does the run time scale with `ng` and the total number of variables `d`? Extrapolate from your measurements to estimate how long it would take to optimize $\Pi_{b,15}$ with a grid search.

[1] Since we are considering a symmetric interval, any odd number of grid points include a point at 0, trivially reaching the optimal input.

[2] Hint: write out the gradient and Hessian for some general function $F(x, y, z) = f(x) + g(y) + h(z)$. You can reuse a lot of work.

4 Solution

The solution to the assignment is given to each question below.

Plot both objective functions on the same axes on the domain $-20 \leq x \leq 20$.

```
%% Problem 1

x = linspace(-20,20,1000);
piA = x.^2;
piB = (x + (pi/2)*sin(x)).^2;

plot(x,piA,'LineWidth',3)
hold on
plot(x,piB,'LineWidth',3)
xlabel('x')
ylabel('$\Pi(x)$','interpreter','Latex')
legend('\Pi_a(x)','\Pi_b(x)')
set(gca,'FontSize',26)
```

Show (analytically) that both Π_a and Π_b have a unique global minimum at $x = 0$.
Hint: one method is to expand Π_b, then use growth rates, bounds, and the basic behavior of the terms.

There are multiple methods to solve this problem, one of which is shown here.

$$\begin{gathered} x^2 \geq 0 \forall x \in R \\ x^2 = 0 \rightarrow \sqrt{0} = \pm 0 = 0 \end{gathered} \tag{4.1}$$

Since the function is strictly non-negative and has a unique solution for $\Pi_a = 0$, this must be a unique global minimum.

Show that, in the limit as $x \to \pm\infty$, Π_b has periodic local extrema and inflection points.

There are multiple methods to solve this problem, one of which is shown here.

$$\begin{gathered} \tfrac{d}{dx}\left(\Pi_b\right) = 2x + \pi\sin(x) + \pi x\cos(x) + \tfrac{\pi^2}{2}\cos(x)\sin(x) \\ \lim_{x\to\pm\infty}\left(\tfrac{d}{dx}\left(\Pi_b\right)\right) = \lim_{x\to\pm\infty}\left(2x + \pi\sin(x) + \pi x\cos(x) + \tfrac{\pi^2}{2}\cos(x)\sin(x)\right) \\ = 2x + \pi x\cos(x) = x(2 + \pi\cos(x)) \\ \underbrace{2-\pi}_{<0} \leq 2 + \pi\cos(x) \leq \underbrace{2+\pi}_{>0} \end{gathered} \tag{4.2}$$

Since the derivative of Π_b will change sign periodically, there will be periodic local extrema in the limit as $x \to 1$

Explain succinctly why the presence of these local extrema and inflection points may make minimizing Π_b challenging using a gradient-based method.

Gradient-based methods using only the first derivative will become trapped by local minima. The inflection points will cause second-order gradient-based methods like Newton's method to behave erratically since Newton's method assumes convexity. Newton's method applied to the gradient will find nearby critical points, which will only be minima if the cost function is strictly convex around the point being evaluated.

Write a function to use Newton's method.

The function is encoded in Matlab below.

```
%% Problem 5

function [sol, its, hist] = myNewton(f, df, x0, TOL, maxit)
    myTol = norm(f(x0));
    its = 0;
    sol = x0;
    hist = [];
    while myTol > TOL && its < maxit
        hist = [hist sol];
        sol = sol - df(sol)\f(sol);
        its = its + 1;
        myTol = norm(f(sol));
    end
end
```

To use `myNewton()` to minimize Π_a and Π_b, define anonymous function arrays (1 by 1 arrays for now) of `f` $= \nabla\Pi$ and `df` $= \nabla^2\Pi$ for both Π_a and Π_b. You should do this using symbolic functions in your language of choice. If working with Matlab, use `diff()`, `hessian()`, and `matlabFunction()`, which converts symbolic functions to anonymous functions. You may also perform the necessary calculations by hand. Show your values for `f` and `df`.

The solution is encoded in matlab below.

```
%% Problem 6

syms x
piA = x^2;
piB = (x + (pi/2)*sin(x))^2;
fA = diff(piA)
dfA = hessian(piA)
fB = diff(piB)
dfB = hessian(piB)
```

Use `myNewton()` to minimize Π_a and Π_b, with `x0` $= 2 \times 10^k$, for $k \in \{-1, 0, 1\}$. Use `TOL` $= 10^{-8}$ and `maxit` $= 20$. Plot Π(`hist`) for Π_a and Π_b. Explain any significant results.

Newton's method assumes the function you want to find local extrema of behaves like a parabola. This means that, if the function is a parabola, it will always be solved exactly in one iteration no matter where you start from. Thus, we can see that a is solved in a single iteration (counting the initial guess as "iteration zero").

With Π_b, convergence is still fast (3 or 4 iterations) but it converges to different values. The point reached with an initial guess of 2 is slightly too high, and the point reached from 20 is very high. You can also see that the value goes up from the starting guess, showing that Newton's method is converging to a local maximum.

This shows the importance of having a good intial guess that puts you into a region of the function where it looks essentially parabolic.
The solution is encoded in matlab below.

```
%% Problem 7

MpiA = matlabFunction(piA);
MpiB = matlabFunction(piB);
MfA = matlabFunction(fA);
```

```
MdfA = matlabFunction(dfA);
MfB = matlabFunction(fB);
MdfB = matlabFunction(dfB);

TOL = 10^-8;
maxit = 20;
x0 = 2E0;

[solA, itsA, histA] = myNewton(MfA, MdfA, x0, TOL, maxit);

figure
plot(histA,MpiB(histA))
```

Find, through trial and error, the range of `x0` values for which Newton's method reaches the global minimum of Π_a and Π_b. Estimate `ng`, the number of equally-spaced points required as `x0` trials on the interval [-20, 20] to ensure Newton's method will find the global minimum of Π_b. Round `ng` up to the nearest power of 2.[1]

You can approximate the size of the region where Newton's method should work by determining the size of the convex region around the true global minimum. To do this, you can find the smallest inflection points using the second derivative of b and a solver like Wolfram Alpha (finding roots of the expression is not trivial). The closest inflection points to the global minimum are located at approximately ± 1.038. If you instead use trial and error, you find that Newton's method starts to work once you are within the approximate range $\pm .75$. Either of these estimates are close enough to find a reasonable grid spacing. To approximate the number of grid points required to cover the domain x 2 [20, 20] densely enough that at least one point is guaranteed to be within this region. You can estimate the number of grid points required by finding the relative size of the "target" interval where Newton's method will converge correctly to the size of the overall domain:

$$\begin{aligned} ng_{\text{est}} &= \frac{2 * 20}{2 * .75} \approx 27 \\ ng_{\text{est}} &= \frac{2 * 20}{2 * 1.038} \approx 19 \end{aligned} \tag{4.3}$$

In either case, rounding up to the next power of two yields the final answer of ng = 32.

Write new versions of Π_b in 2D and 3D

There are several methods to build the higher-dimensional versions of these functions with vector-valued input. The example code below takes advantage of the fact that all the derivatives are essentially the same to copy the 1D anonymous functions into 3D without using symbolics again. There are also ways to build a general symbolic expression for the Hessian and gradient that accept a single vector-valued argument.

```
%% Problem 9

syms x % define x symbolic
Pi_A = x.^2; dPi_A = diff(Pi_A); ddPi_A = hessian(Pi_A, x); % initialize symbolic functions

% DEFINE NONCONVEX PROBLEM
Pi_B(x) = (x+(pi/2)*sin(x)).^2; dPi_B = diff(Pi_B, x); ddPi_B = hessian(Pi_B);

% CONVERT EVERYTHING TO ANONYMOUS
Pi_B = matlabFunction(Pi_B); dPi_B = matlabFunction(dPi_B); ddPi_B = matlabFunction(ddPi_B
Pi_A = matlabFunction(Pi_A); dPi_A = matlabFunction(dPi_A); ddPi_A = matlabFunction(ddPi_A
```

[1] Since we are considering a symmetric interval, any odd number of grid points include a point at 0, trivially reaching the optimal input.

```
ddPi_A = @(x) ddPi_A(); % hessian(x.^2) is just @() 2.0, which denies input arguments.

% DEFINE 2 D AND 3 D VERSIONS OF Pi_B by reusing 1D information
Pi_B2 = @(x) Pi_B(x(1)) + Pi_B(x(2)); dPi_B2 = @(x) [dPi_B(x(1)); dPi_B(x(2))]; ddPi_B2 = @
Pi_B3 = @(x) Pi_B(x(1)) + Pi_B(x(2)) + Pi_B(x(3)); dPi_B3 = @(x) [dPi_B(x(1)); dPi_B(x(2))
ddPi_B3 = @(x) [ddPi_B(x(1)), 0, 0; 0, ddPi_B(x(2)), 0; 0 , 0 , ddPi_B(x(3))];
```

Attempt to minimize Π_b, $\Pi_{b,2}$, and $\Pi_{b,3}$ using a grid search in the search space $-20 \leq x \leq 20$, $-20 \leq y \leq 20$, $-20 \leq z \leq 20$. Use `linspace(-20, 20, ng)` for x, y, and z. Run each search several times (at least 10 but the more the better) and calculate the average run time. How does the run time scale with `ng` and the total number of variables `d`? Extrapolate from your measurements to estimate how long it would take to optimize $\Pi_{b,15}$ with a grid search.

Runtimes will vary somewhat depending on your exact implementation, but the trends should be fairly consistent. The key here is to recognize how the total number of points that need to be checked scales with the number of dimensions. In 1D, we merely need to check a total of ng points. In 2D, we have to check grid points determined by each combination of ng points in the x direction and the y direction for a total of ng^2 points. This trend continues. As long as we are using the same number of points per dimension, the total number of points to be checked is given, in d dimensions, by:

$$total\,points = ng^d \tag{4.4}$$

$$t \approx c \cdot ng^d \tag{4.5}$$

where c is a constant that depends on your machines run time and implementation strategy. Regardless of what you are actually doing at each of these points, the run time will be roughly proportional to this exponential term, and will certainly not scale better than this. It is possible to fit the data with a more general exponential expression, but starting from this intuitive scaling law is very helpful. Your model should certainly be exponential in the number of dimensions.

5 References

Zohdi, T. I. and Wriggers, P. (2005, 2008) Introduction to Computational Micromechanics

Zohdi, T. I. (2018) Modeling and Simulation of Functionalized Materials for Additive Manufacturing and 3D printing: Continuous and Discrete media

Zohdi, T. I. (2012) Dynamics of Charged Particulate Systems. Modeling, Theory and Computation

Zohdi, T. I. (2012) Electromagnetic Properties of Multiphase Dielectrics. A Primer on Modeling, Theory and Computation

Zohdi, T. I. (2007) Introduction to the modeling and simulation of particulate flows

Zohdi, T. I. (2001). Computational optimization of vortex manufacturing of advanced materials. Computer Methods in Applied Mechanics and Engineering. 190. 46-47, 6231-6256.