Appendix B

Interview Questions and Answers

By now readers should be able to answer all pre-interview questions given in the preface. For easy reference and completeness, I will go through the seven pre-interview questions again with my answers and additional comments.

B.1 Area tolerance

No two components can be manufactured exactly the same. In practice, dimensional or geometric tolerances (e.g., distance and angular tolerances) are introduced in Computer Aided Design systems and manufacturing processes. The smaller the tolerance required, the more expensive the component will be to machine. Therefore, it is often desirable to specify the largest possible tolerance while maintaining proper functionality.

Denoting the distance tolerance by ε, then two circles are considered to be the same if

$$|R_1 - R_2| < \varepsilon,$$

where R_1 and R_2 are the radii of circles 1 and 2 respectively. Assume that we know the areas of the two circles (denoted by A_1 and A_2) and the radius of circle 1 (i.e., R_1). Please derive the "area tolerance" Δ in terms of the given distance tolerance ε such that, when $|A_1 - A_2| < \Delta$, we know two circles are equal with respect to the distance tolerance. One should avoid computing radii via $R_i = \sqrt{A_i/\pi}, i = 2, 3, \cdots, n$, as the square root computation is an expensive arithmetic operation. Although a non-calculus based approach is acceptable, a calculus based approach is preferable since calculus is the study of change.

Answer: I have seen in different occasions that developers use ε^2 to check if one lamina (i.e., a closed surface in plane) is equal to the other. Their reasoning is that, since ε is for one dimensional measurement and area is the two-dimensional analog of the length of a curve (a one-dimensional concept), the area tolerance is simply ε^2 to match the square unit. This is mathematically incorrect and also a much stricter condition than

necessary. As will be seen below, the area tolerance of a circle is usually much larger than the distance tolerance since R is, in practice, larger than ε.

- Non-calculus: Assume that $R_2 = R_1 \pm \varepsilon$. Then,

$$\Delta = |A_2 - A_1| = \pi|(R_1 \pm \varepsilon)^2 - R_1^2| = \pi|\pm 2R_1\varepsilon + \varepsilon^2| = \pi|2R_1\varepsilon \pm \varepsilon^2|.$$

  Since ε^2 is negligible, we have $\Delta = 2\pi R_1\varepsilon$.

- Calculus: $A = \pi R^2$. By the rule of differential, $dA = 2\pi R\, dR$. Since dA measures the change of the area with respect to the small change of the radius (i.e., dR), we have $\Delta = 2\pi R\varepsilon$. Replacing R by the known R_1, we obtain $\Delta = 2\pi R_1\varepsilon$.

Comments: This question was designed to assess candidates' ability to apply their math skills to solve basic engineering problems. In calculus, the derivative measures how fast the function is changing and the differential represents the *principal part* of the change in a function with respect to the change in a variable. This principal part of function change is what we need to derive the area tolerance. The same technique may be used in other cases. For example, in CAD/CAM systems, parametric polynomial curves and surfaces are used to describe wine glasses, telephone handsets, car bodies, etc. For many geometric operations, it is desirable to compute the parametric tolerance for a given parametric curve. If the change of parameter is less than the parametric tolerance, the change of the curve length is then guaranteed to be within the distance tolerance. Representing a parametric curve in a vector-valued form $\boldsymbol{r}(u)$ and differentiating it with respect to the parameter $u \in [a, b]$ gives

$$\frac{d\boldsymbol{r}(u)}{du} = \boldsymbol{r}'(u) \quad \Longrightarrow \quad \|d\boldsymbol{r}(u)\| = \|\boldsymbol{r}'(u)\|\, du.$$

If we can find the upper bound of the magnitude of the first derivative in $[a, b]$ and denote it by $\boldsymbol{r}'(\xi)$, we can then derive the parametric tolerance as

$$du \le \frac{\|d\boldsymbol{r}(u)\|}{\|\boldsymbol{r}'(\xi)\|} \le \frac{\varepsilon}{\|\boldsymbol{r}'(\xi)\|}.$$

As an example, let's see how the parametric tolerance of a circle is derived. Representing a circle in parametric form gives

$$\begin{aligned} x(\theta) &= x_0 + \rho\cos\theta, \\ y(\theta) &= y_0 + \rho\sin\theta, \end{aligned}$$

where (x_0, y_0) is the center of the circle and ρ is the radius. Differentiating $x(\theta)$ and $y(\theta)$ with respect to θ, we have

$$\begin{aligned} x'(\theta) &= -\rho\sin\theta, \\ y'(\theta) &= \rho\cos\theta. \end{aligned}$$

Accordingly,

$$\|\boldsymbol{r}'(\theta)\| = \sqrt{x'(\theta)^2 + y'(\theta)^2} = \rho.$$

Therefore, we have

$$d\theta = \frac{\varepsilon}{\rho},$$

where $d\theta$ is the parametric tolerance. This result can be confirmed by a geometric approach. When the angle changes by $\Delta\theta$, the corresponding arc length change is $\rho \times \Delta\theta$. When $\Delta\theta = 2\pi$, we have $2\pi\rho$ – the circumference of the circle. If we want $\rho \times \Delta\theta < \varepsilon$, we have

$$\Delta\theta = \frac{\varepsilon}{\rho}.$$

This geometric approach leads to the next interview question.

B.2 Angular tolerance

Again, we denote the distance tolerance by ε and assume that the model space (the longest model we would create) is 10^4 meters. Based on ε and the limit of model space, I would like you to derive the angular tolerance Δ such that it can be used to determine whether two lines confined in the model space are collinear.

Hint: Two lines are considered to be collinear if the maximum deviation between these two lines is less than the distance tolerance ε. From this criterion, you may derive the angular tolerance such that you know two lines are collinear with respect to ε if the angle between these two lines is less than the angular tolerance Δ. Drawing two lines on a piece of paper may help you analyze the problem.

Answer: Assume two longest lines passing through the same point are not exactly collinear and the angle between two lines is Δ. Then, the sweeping arc length between two lines is $\Delta \times 10^4$, which needs to be smaller than the distance tolerance ε. This is to say that $\Delta \times 10^4 < \varepsilon$. Therefore, $\Delta < \varepsilon/10^4$. Alternatively, this can be derived from the right triangle rules: let 10^4 be the hypotenuse and the cathetus (the opposite leg) be the h. Then,

$$\frac{h}{10^4} = \sin\Delta \approx \Delta.$$

Since h needs to be less than ε, we have $\Delta < \varepsilon/10^4$.

Comments: The above angular tolerance is much stricter than necessary when the geometry size is much less than the maximum allowed size 10^4. For example, the angular tolerance for a circle is ε/ρ, where ρ is the radius of the circle. When ρ is shorter than 10^4 unit, ε/ρ is much larger than $\Delta = \varepsilon/10^4$. Using the smallest tolerance is a sufficient but not necessary condition.

We may alternatively explain how the angular tolerance and model space limit are derived as follows: For 32-bit computer, a double-precision floating-point format uses 52-bit for the significand, which is equivalent to 15 or 16 significant decimal digits ($\log_{10} 2^{52} \approx 15.654$). We consider five of the least significant digits to represent numeric round-off errors that occur during calculations. Thus, there are roughly 10 digits to represent the dynamic range of numbers (smallest and largest numbers) within an object

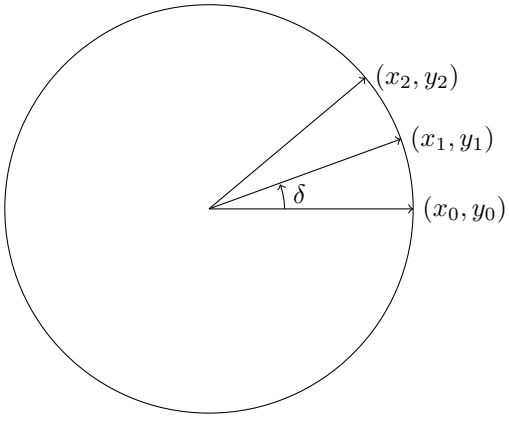
space. Accordingly, the angular tolerance, which is the smallest of all three tolerances, is set to 10^{-10}. When the distance and angular tolerances are defined, the longest line we can model would be

$$\frac{10^{-6}}{10^{-10}} = 10,000 \text{ units.}$$

B.3 Improve performance of circle stroking

Most graphics programming tools provide only a line drawing capability. To display a circle on computer screen, we first need to break the circle into evenly spaced tiny pieces such that each tiny piece can be well-approximated by a line. We then draw these lines on the screen. Because these lines are so small, a circle looks smooth on the screen.

Breaking a circle into many tiny pieces is equivalent to computing many evenly spaced points on the circle, a process known as circle stroking. If we want to stroke $n + 1$ points on circle and start the first point at $\theta_0 = 0$, then the angle increment δ would be $2\pi/n$ as shown below.



Accordingly, all other points may be computed as follows:

$$\begin{aligned}
x_1 &= r\cos(\theta_1) = r\cos(\theta_0 + \delta) \\
y_1 &= r\sin(\theta_1) = r\sin(\theta_0 + \delta) \\
x_2 &= r\cos(\theta_2) = r\cos(\theta_1 + \delta) \\
y_2 &= r\sin(\theta_2) = r\sin(\theta_1 + \delta) \\
&\cdots \\
x_n &= r\cos(\theta_n) = r\cos(\theta_{n-1} + \delta) \\
y_n &= r\sin(\theta_n) = r\sin(\theta_{n-1} + \delta)
\end{aligned}$$

The above formula indicates that we need to call trigonometric functions (i.e., sin and cos) for $2(n + 1)$ times. Since trigonometric functions are relatively more expensive to compute than multiplication and division, the above approach is not optimized in terms of CPU usage. Please provide a suggestion on how you can speed up the computation.

Answer: Form high school math, it is known that

$$\cos(\theta_i + \delta) = \cos(\theta_i)\cos(\delta) - \sin(\theta_i)\sin(\delta)$$

$$\sin(\theta_i + \delta) = \sin(\theta_i)\cos(\delta) + \cos(\theta_i)\sin(\delta)$$

With the above formula, we can compute the $(i+1)$th point as follows:

$$x_{i+1} = r\cos(\theta_i + \delta) = r\cos(\theta_i)\cos(\delta) - r\sin(\theta_i)\sin(\delta) = x_i\cos(\delta) - y_i\sin(\delta)$$

$$y_{i+1} = r\sin(\theta_i + \delta) = r\sin(\theta_i)\cos(\delta) + r\cos(\theta_i)\sin(\delta) = y_i\cos(\delta) + x_i\sin(\delta)$$

Therefore, we compute $\sin(\delta)$ and $\cos(\delta)$ once and use them to derive all stroking points via the recursive relation. The code fragment is given below:

```
sin_delta = sin(delta);
cos_delta = cos(delta);
x[1] = r;
y[1] = 0.0;
for (i=1; i<=n; i++)
{
    x[i+1] = x[i] * cos_delta - y[i] * sin_delta;
    y[i+1] = y[i] * cos_delta + x[i] * sin_delta;
}
```

Alternatively, we may create a rotation transformation with the rotating angle being δ and then compute (x_{i+1}, y_{i+1}) as follows:

$$\begin{pmatrix} x_{i+1} \\ y_{i+1} \end{pmatrix} = \begin{pmatrix} \cos\delta & -\sin\delta \\ \sin\delta & \cos\delta \end{pmatrix} \begin{pmatrix} x_i \\ y_i \end{pmatrix}$$

Comments: This question was designed to evaluate candidates' ability to use their basic math knowledge to simplify a problem. However, it should be pointed out that trigonometric functions are widely available across programming languages and platforms. Most CPU architectures have built-in instructions for trigonometric functions. Therefore, it will not be terribly slow if our circle stroking method is implemented as follows:

```
for(i=0; i<nStrkPts; i++)
{
   pStrkPts[i].x = center.x + radius * cos(i * delta);
   pStrkPts[i].y = center.y + radius * sin(i * delta);
}
```

Based on our timing, it is roughly 3 to 4 times slower when repeatedly calling the standard sin and cos math functions. As was seen in chapter 11, the above code fragment can be improved by using multi-thread programming. Even so, it is still outperformed by our first implementation that computes sin and cos once (Timing test was based on a laptop computer with a Core i7 processor that has eight-hyperthread).

B.4 Approximation by circular arc

A circle in quadratic form is

$$x^2 + y^2 + 2Ax + 2By + C = 0. \tag{B.4.1}$$

Its center and radius are given by $(-A, -B)$ and $\sqrt{A^2 + B^2 - C}$ respectively. Given n points (x_i, y_i) $(i = 1, 2, \cdots, n)$, we want to find a circle that interpolates (x_1, y_1) and (x_n, y_n) and approximates the remaining points in a least squares sense. Substituting (x_1, y_1) and (x_n, y_n) in the above equation gives

$$x_1^2 + y_1^2 + 2x_1A + 2y_1B + C = 0 \tag{B.4.2}$$

$$x_n^2 + y_n^2 + 2x_nA + 2y_nB + C = 0 \tag{B.4.3}$$

Subtracting the second equation from the first yields

$$2(x_1 - x_n)A + 2(y_1 - y_n)B + x_1^2 + y_1^2 - x_n^2 - y_n^2 = 0.$$

Assume that $|x_1 - x_n| \neq 0$ so we may express A in terms of B, i.e.,

$$A = \frac{y_1 - y_n}{x_n - x_1}B + \frac{x_1^2 + y_1^2 - x_n^2 - y_n^2}{2(x_n - x_1)} = \alpha B + \beta \tag{B.4.4}$$

Replacing A in (B.4.1) gives

$$2(x\alpha + y)B + C + x^2 + y^2 + 2x\beta = 0.$$

If an arbitrary point $\boldsymbol{p}_i = (x_i, y_i)$ is not on the circle, then

$$e_i = \left|2(x_i\alpha + y_i)B + C + x_i^2 + y_i^2 + 2x_i\right| > 0$$

is the approximation error. The sum of all squared errors is

$$\mathcal{E} = \sum_{i=2}^{n-1} \left(2(x_i\alpha + y_i)B + C + x_i^2 + y_i^2 + 2x_i\beta\right)^2$$

From calculus, $\mathcal{E}$ is minimized if

$$\frac{\partial \mathcal{E}}{\partial B} = 0 \quad \text{and} \quad \frac{\partial \mathcal{E}}{\partial C} = 0.$$

Optimized B and C are obtained by solving the above two linear equations and A is computed via (B.4.4).

Implementing the above approach will not result in a circle that interpolates (x_1, y_1) and (x_n, y_n). Do you see any reasoning problem in this approach?

Answer: A circle is uniquely defined by three non-collinear points. If a circle is required to interpolate two endpoints, two of the three unknowns (A, B, and C) need to be eliminated. The above approach failed to do so. Let's see where it went wrong. Relation

(B.4.4) is not equivalent to relations (B.4.2) and (B.4.3). We need to eliminate two unknowns (e.g., A and B) from the two endpoint conditions (or two boundary conditions) and solve the reduced equation for ONE unknown. Relation (B.4.4) destroyed the given conditions. There are several possible ways to solve this least squares approximation problem. A simpler and numerically stabler approach would be: compute the mid-point of the chord that links (x_1, y_1) and (x_n, y_n), and transform all the points to a new coordinate system in which the x-axis is parallel to the chord and the origin coincident with the mid-point. In this case, $A = 0$ and equation (B.4.1) involves only two variables:

$$x^2 + y^2 + 2By + C = 0.$$

In the new coordinate system, (x_1, y_1) sits on the x-axis. Therefore, $y_1 = 0$ and $C = x_1^2$ (note: we can also use (x_n, y_n) to derive $C = x_n^2$ since it lies on the x-axis as well). This further reduces the equation to one unknown

$$x^2 + y^2 + 2By + x_1^2 = 0.$$

Given n points, the sum of the squared errors is

$$\mathcal{E} = \sum_{i=2}^{n-1} \left(x_i^2 + y_i^2 + 2By_i + x_1^2\right)^2,$$

which can be solved by letting $d\mathcal{E}/dB = 0$.

B.5 Evaluation of e^x

Exponential function e^x and trigonometric functions such as $\sin x$ and $\cos x$ are often evaluated via the Maclaurin series (a special form of Taylor series). Expressing e^x by the Maclaurin series gives

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{i=0}^{\infty} \frac{x^i}{i!}.$$

This Maclaurin series converges to e^x absolutely and uniformly for $x \in (-\infty, +\infty)$. Please write a program to evaluate e^x via Maclaurin series at x =1 and ± 25 and compare your results with the ones obtained by using either a pocket calculator or the C/C++ internal math function `exp(x)`. An explanation of your implementation and findings is expected.

Hint: Let $P_n = \sum_{i=0}^{n} \frac{x^i}{i!}$. Then, P_n is said to be converged to e^x with respect to the given tolerance ε if $|P_n - P_{n-1}| = \left|\frac{x^n}{n!}\right| < \varepsilon$. Let's assume that $\varepsilon = 10^{-15}$.

Answer: One should not directly compute the factorial $i!$ because it is inefficient to do so and can cause an overflow when i is large. By examining the above series, it is noted that the ith term is equal to multiplying the $(i-1)$th term by $\frac{x}{i}$. Accordingly, we can evaluate the above polynomial efficiently via the variation of Horner's method:

```
void apiEvalExp(double x, int n, double &f)
{
   int         i;
   double      term, f;

   f = 0.0;
   term = 1.0;
   for (i=1; i<=n; i += 1)
   {
      f += term;
      term = term * x / i;
   }
}
```

The above implementation requires the user to specify n, which is not practical. A feasible approach is specifying the convergence tolerance rather than n. Denoting

$$P_n(x) = \sum_{i=0}^{n} \frac{x^i}{i!},$$

then $P_n(x)$ is said to be converged to e^x with respect to the given tolerance ε if

$$|P_n(x) - P_{n-1}(x)| < \varepsilon.$$

It is noted that $|P_n(x)-P_{n-1}(x)|$ gives the last term of $P_n(x)$. Therefore, the convergence criterion is to check whether `term` is less than the given accuracy. So the complete implementation is:

```
void apiEvalExp(double x, double epsilon, double &f)
{
   int         i;
   double      term;

   f = 0.0;
   term = 1.0;
   i = 1;
   while (true)
   {
      f += term;
      if (fabs(term) < epsilon)
         break;
      term = term * x / i;
      i++;
   }
}
```

Since the Maclaurin series converges to e^x uniformly and absolutely for all x, one may assume that the above implementation will always give a correct result for any x. This is true only if $x \geq 0$. If $x = -25$ is used to test the above implementation, the result is

$$-7.1297804036720779e - 007.$$

But the correct answer is

$$1.3887943864964021e-011,$$

which is obtained via the standard C internal function `exp(x)`. Two results are not only significantly different but also have different signs. How could a converging series lead to a diverging result? Many standard computation rules apply only to infinite precision arithmetic. In floating-point computation, it is very important to design an algorithm that minimizes the loss of significant digits. Referring to the above Maclaurin series, the even terms are positive and odd terms are negative, so the numerical instability is caused by subtraction of similar numbers. To avoid such catastrophic problem, we should always treat x as a positive value and compute f. If $x < 0$, we have $e^{-x} = 1/e^x$. Accordingly, $1/f$ is the result for e^{-x}.

Comments: Without thoughtful analysis of numerical stability (the necessary step for a good design), one may implement the evaluation as follows:

```
inline long long Factorial(long long n)
{
   return (n == 1 ? n : n * Factorial(n - 1));
}

void EvaluateExpX(int n, double x, double &f)
{
   int i;

   f = 1.0;
   for (i=1; i<=n; i += 1)
   {
      f += pow(x, i) / Factorial(i);
   }
}
```

It is noted that `Factorial` is declared as an *inline function*. The `inline` specifier instructs the compiler to insert a copy of the function body into each place the function is called. Using inline functions can make our program faster because they eliminate the overhead associated with function calls, but at the cost of larger code size. In computing the factorial $n!$, a `long long` data type is used to avoid overflow. In programming, an overflow occurs when an arithmetic operation attempts to create a numeric value that is too large to be represented within the available storage range. For a 32bit machine, `long long` is equivalent to `_int64` and has the data range of

$$-9,223,372,036,854,775,808 \quad \text{to} \quad 9,223,372,036,854,775,807.$$

This means n should be less than or equal to 20. When n is larger than 20, we will have an integer overflow. Some candidates sent me the following improved code, in which a `double` was used to store the factorial and a call to the `inline` function was eliminated:

```
void EvaluateExpX(int n, double x, double &f)
```

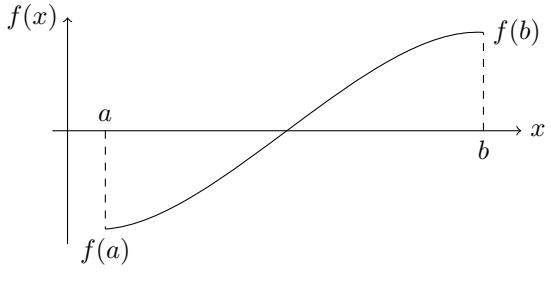
```
{
   int      i;
   double   factorial;

   f = factorial = 1.0;
   for (i=1; i<=n; i += 1)
   {
      factorial *= i;
      f += pow(x, i) / factorial;
   }
}
```

This is a big improvement in terms of eliminating an integer overflow and boosting performance. But the performance can be further improved by eliminating a call to `pow(x, i)` as demonstrated in my implementation.

B.6 Implementation of bisection method

Assume a continuous function f is monotonic on the interval $[a, b]$ and $f(a) \times f(b) < 0$. Then, there exists a solution x in $[a, b]$ such that $f(x) = 0$ as illustrated below.



To find the solution numerically, we can use the bisecting method. As the name suggests, we start by selecting x at the mid of the interval $[a, b]$ and compute $f(x)$. If $|f(x)| < \varepsilon$, x is the root of function f. Otherwise, we reduce the interval by half using the following criterion:

```
if (f(a) * f(x) < 0.0)
{
   b = x;
}
else
{
   a = x;
}
```

Based on the reduced interval, we compute again $x = 0.5(a + b)$ and check if $|f(x)| < \varepsilon$. Repeating this process should find the root of $f = 0$.

If you are not familiar with object oriented programming, you may simply implement a workable recursive function to compute the root of $f(x) = x^3 + 2x - 2$ in $[0, 1]$. Otherwise, I would like you to demonstrate how you can implement, say, `Bisection` class to compute the root of the above function. Furthermore, you may want to show me how you can compute the root of another function $f(x) = \sin(x)$ in $[2, 4]$ based on `Bisection` class with minimum additional implementation.

Answer: It is noted that we did not explicitly specify any particular function when discussing how to find the root of $f = 0$ via a recursion method. In other words, this bisecting recursive algorithm is universal to any function. For this reason, we may declare a `CBisection` class as follows:

```
class CBisection
{
public:

   // pure virtual function to be implemented by derived class
   virtual double EvaluateFunc(double x) = 0;

   double ComputeRoot(double a, double b, double tol)
   {
      double f_a, f_b, f_mid, x_mid;

      // Terminate condition
      x_mid = 0.5 * (a + b);
      f_mid = this->EvaluateFunc(x_mid);
      if (fabs(f_mid) < tol)
         return x_mid;

      // Recuce the complexity via recursion
      f_a = this->EvaluateFunc(a);
      f_b = this->EvaluateFunc(b);
      if (f_a * f_mid < 0.0)
         return ComputeRoot(a, x_mid, tol);
      else
         return ComputeRoot(x_mid, b, tol);
   };
};
```

The method `ComputeRoot` is a generic implementation of the bisection algorithm. `EvaluateFunc` is declared as a *pure virtual* function by placing `= 0` at the end of its declaration, indicating the derived class must implement it. It is left for the derived classes to do so because the implementation of `EvaluateFunc` varies with respect to different applications. The pure virtual function is also known as an *abstract* function. It is noted that neither a constructor nor a destructor is implemented since we do not have any specific data needs to be initialized or released.

With the above base class, we can implement a derived class to solve any specific

question with minimal coding effort. For example, to find a solution of $\sin x = 0$ in the given interval $[a, b]$, we first define the `CSineRoot` class whose parent class is the base class `CBisection`. Since a derived class inherits both data and member functions from a parent class, `CSineRoot` class has access to the generic bisection method `ComputeRoot` defined in the parent class. The only thing we need to do inside the derived class is to implement the pure virtual function `EvaluateFunc` to evaluate $\sin x$:

```
// A class to find x such that sin(x) = 0
class CSineRoot : public CBisection
{
public:
   // Implementation of pure virtual function
   double EvaluateFunc(double x)
   {
      return sin(x);
   }
};
```

Then, the root of $\sin x$ is computed by creating an instance of `CSineRoot` class and calling the bisection method as follows:

```
 CSineRoot  objSin;
 x = objSine.ComputeRoot(a, b, tol);
```

Similarly, to find a root of the polynomial function

$$f(x) = \sum_{i=0}^{n} a_i x^i,$$

we derive the `CPolynomialRoot` class from the base class `CBisection`. Inside this derived class, we implement the pure virtual function to evaluate the polynomial:

```
// A class to find x such that a polynoomial function f = 0
class CPolynomialRoot : public CBisection
{
private:
   int      m_ndeg;
   double   *m_pA;

public:
   // Use constructor to initialize polynomial
   CPolynomialRoot(int ndeg, double *pA)
   {
      m_ndeg = ndeg;
      m_pA = new double[ndeg + 1];
      memcpy(m_pA, pA, (ndeg + 1) * sizeof(double));
   };

   // Destructor is responsible to delete memory
```

```
    ~CPolynomialRoot()
    {
       if (m_pA)
          delete [] m_pA;
    }

    // Implementation of pure virtual function
    double EvaluateFunc(double x)
    {
       double f;
       apiHornerEval(m_ndeg, m_pA, x, f);
       return f;
    }
};
```

Accordingly, the root of $f(x) = x^3 + 2x - 2$ in $[a, b]$ is computed by creating an instance of `CPolynomialRoot` class and calling the bisection method as follows:

```
double          A[4] = {-2.0, 2.0, 0.0, 1.0};
CPolynomialRoot objCubic(3, A);
x = objCubic.ComputeRoot(a, b, tol);
```

As seen above, there is no need to modify the base class and its implementation to evaluate a different function. Instead, we simply overrides the virtual function with an evaluation method specific to the application. This is known as *inheritance with polymorphism* in object oriented design/programming. If you are C programmers, you can use a callback function to achieve the task of minimizing the code change to find a root of any function.

Comments: Object-oriented programming has been around for many decades and taught as a core course in computer science. Many candidates I have interviewed knew the terminologies of *data abstraction, encapsulation, polymorphism*, and *inheritance*. When asked to demonstrate their knowledge and skill to design a bisection method that is capable of finding a root of different functions, very few of them sent back a satisfactory solution.

It appears to me that there is a gap between academic training and real world computing. For example, mammals (or people) are often used in many text books to teach inheritance and polymorphism. Engineering students are taught how mammals form a base class and how dogs and cats are derived from mammals and thus form the derived classes. The need for inheritance and polymorphism sounds very artificial and may fail to draw students' attention to the topic and may even cause their resistance to the idea. Therefore, when asked to use inheritance with polymorphism to design a bisection method that can be used to solve any function, candidates had trouble connecting the dots between what they have learned in the classroom and real engineering applications.

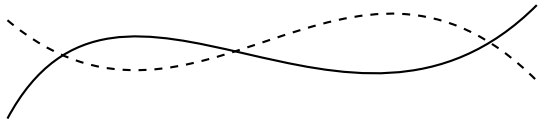
The interview question was designed to evaluate candidates' understanding of object oriented programming. In C++ programming, inheritance allows one data type to

acquire properties of other data types. Inheritance from a base class may be declared as *public*, *protected*, or *private*. This access specifier determines whether unrelated and derived classes can access the inherited public and protected members of the base class. Inheritance allows code to be reused between related types. Polymorphism enables one common interface for many implementations, and for objects to act differently under different circumstances. Polymorphism is often achieved by method overriding, which is not the same as method overloading. Method *overloading* refers to methods that have the same method name but different signatures inside the same class. Method *overriding* is where a subclass replaces the implementation of one or more of its parent's methods.

B.7 Curve/Curve intersection

Simple curves such as lines and arcs are widely used in Computer Aided Design and Manufacturing (CAD/CAM) systems. In addition to a line and arc, a piecewise polynomial curve (either a composite Bézier curve or a B-spline curve) is also used to model complex geometric shapes.

Assume we already have an API to solve a line/line intersection problem. I would like you to derive a method to compute the intersection points of two generic curves as shown below by calling the line/line intersection API.

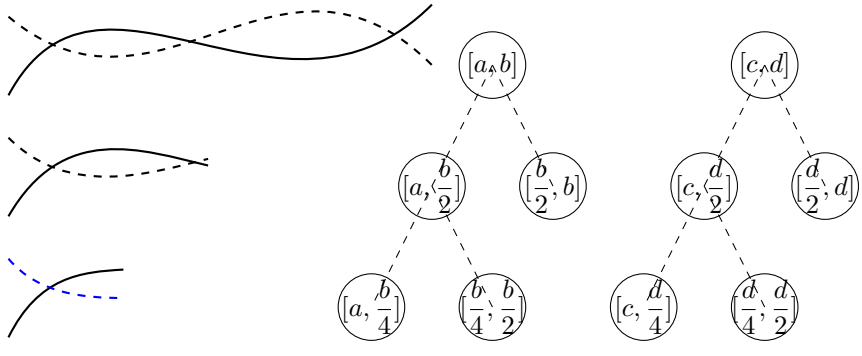


Two curves intersect each other at multiple places

I am not interested in C/C++ code implementation. Instead, I would like to see how you analyze the problem (math) and organize your data (computer science). If you want, you can write the pseudo code to clarify your approach.

Answer: If you paid attentions to Question 3 (stroking a circle) and 5 (bisecting algorithm to solve $f(x) = 0$), you should have figured out the solution to this question. In essence, you bisect the solid and dashed curves and save the halved segments (or intervals) in two binary trees (referring to the figure below). It is assumed that the solid curve is defined in the interval $[a, b]$ and dashed curve in $[c, d]$. After the first subdivision, the left- and right-halved solid curves are defined respectively in $[a, b/2]$ and $[b/2, b]$. The intervals are continuously halved when the subdivision process is repeated.

When the segment is almost linear with respect to the given tolerance, you call the line/line intersection routine to obtain an intersection point of two approximating lines. This intersection is then used as a good estimation for the Newton iteration to refine the solution at a much faster convergent speed. To optimize the computation, range boxes (or bounding boxes) are often used to filter out the halved segments whose ranges boxes do not intersect each other. Accordingly, you proceed the operations with preference on the most likely candidates as illustrated below.



Each node stores the halved candidate curve

Since the two curves may intersect at multiple places, you will need to transverse the two binary trees to make sure you have checked every halved segment in the trees.

A brute force solution to this problem is to stroke each curve with respect to certain tolerance and save the stroking points in two arrays. It then loops through the two arrays to create lines from two adjacent stroking points and to call the line/line intersection routine. Such an approach is an $O(n^2)$ method and should be avoided by all means.