1 算法

1.1 数值逼近

1.1.1 基本思想

在提供的参考文献《Computational Mirror Cup and Saucer Art》中，作者介绍了两种计算圆柱镜面反射的方法。其中一个便是数值逼近，这种方法控制 T 点在第二象限的四分之一圆上移动，并即时判断 $\angle PTV$ 和 $\angle QTV$ 是否相等，若两角相等，则找到 T 点。

找到 T 点后，利用解析法可以求出与圆相切于点 T 的直线：

$$l : -\frac{x_T}{y_T} \cdot x - y + y_T - \left(-\frac{x_T}{y_T}\right) \cdot x_T = 0$$

再解出 Q 点关于该切线的对称点：

$$x_R = x_Q - 2\left(-d\frac{x_T}{y_T}\right) \cdot \frac{-\frac{x_T}{y_T} \cdot x_Q - y_Q + \left[y_T - \left(-\frac{x_T}{y_T}\right) \cdot x_T\right]}{\left(-\frac{x_T}{y_T}\right)^2 + 1}$$

$$y_R = y_Q + 2 \cdot \frac{-\frac{x_T}{y_T} \cdot x_Q - y_Q + \left[y_T - \left(-\frac{x_T}{y_T}\right) \cdot x_T\right]}{\left(-\frac{x_T}{y_T}\right)^2 + 1}$$

即为最终的要求的像点。

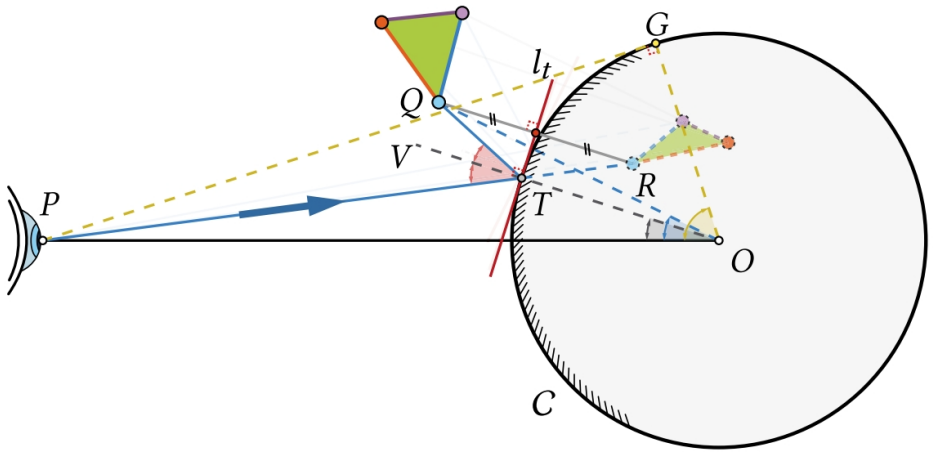

图 1: 给定圆柱形反射器 C 之外的点 Q，P 处的观察者在点 $T \in C$ 处看到它的反射 R。对于每个点 Q，可以通过求解平面几何问题找到点 R 的位置。

1.1.2 程序实现

在我的算法中，我利用 `while` 循环使得点 T 在第二象限的四分之一圆上移动，同时判断当前点 T 是否满足 $\angle PTV = \angle QTV$ 的条件，若满足则返回结果。

Listing 1: Find T

```
vector CalculateT(vector P, vector Q, vector T, circle C){
    vector TP = VectorSubtract(P, T);
    vector TQ = VectorSubtract(Q, T);

    while (!isEqual(Angel(T, TP), Angel(T, TQ))){
        T.cordinary[0] += 0.00000001;
        if (Angel(T, P) >= Angel(P, Q)){
            printf("Processing accuracy is exceeded...");
            exit(-1);
        }
        T.cordinary[1] = sqrt(pow(C.radius, 2) - pow((T.cordinary[0] -
            C.center[0]), 2)) + C.center[1];
        TP = VectorSubtract(P, T);
        TQ = VectorSubtract(Q, T);
    }

    return T;
}
```

其中，函数 `VectorSubtract(vector A, vector B)` 利用对应坐标相减实现向量减法；函数 `Angel(vector A, vector B)` 利用向量内积和向量模长的关系计算两向量之间的夹角；函数 `isEqual(double A, double B)` 会根据两个值的差值是否小于指定精度 ϵ 来判定值是否相等。得到需要的点 T 后，便可以利用解析法计算像点。

Listing 2: Calculate R

```
vector CalculateR(vector Q, vector T){
    double a = -1.0 / (T.cordinary[1]/T.cordinary[0]);
    double b = T.cordinary[1] - a * T.cordinary[0];
    double x = Q.cordinary[0] - (2 * a * (a * Q.cordinary[0] - Q.
       cordinary[1] + b))/(pow(a, 2) + 1);
    double y = Q.cordinary[1] - (2 * (-1.0) * (a * Q.cordinary[0] - Q.
       cordinary[1] + b))/(pow(a, 2) + 1);
    vector result = {x, y, CalculateMol(result)};
    return result;
}
```

1.1.3 算法缺陷

上述方法的缺陷在于：

(1) 计算量大，循环次数较多；

(2) 数据精度不高，在之后的实验结果部分将会看到；

(3) 时间复杂度高。

1.2 Newton iteration

1.2.1 基本思想

当 $\vec{P}$ 和 $\vec{Q}$ 不共线时，可以设 $\vec{T}=x\vec{P}+y\vec{Q}$，列出方程组：

$$\begin{cases} f(x,y)=2\langle P,Q\rangle xy+2|Q|^2y^2-y-1+x \\ g(x,y)=|P|^2x^2+2\langle P,Q\rangle xy+|Q|^2y^2-1 \end{cases}$$

求导后为：

$$\begin{cases} \dfrac{\partial f(x,y)}{\partial x}=2\langle P,Q\rangle y+1 \\ \dfrac{\partial f(x,y)}{\partial y}=2\langle P,Q\rangle x+4|Q|^2y-1 \end{cases} \qquad \begin{cases} \dfrac{\partial g(x,y)}{\partial x}=2|P|^2x+2\langle P,Q\rangle y \\ \dfrac{\partial g(x,y)}{\partial y}=2\langle P,Q\rangle x+2|Q|^2y \end{cases}$$

利用 Newton 迭代法，得出：

$$\begin{pmatrix} x_{k+1}-x_k \\ y_{k+1}-y_k \end{pmatrix}=\begin{pmatrix} \dfrac{\partial f(x_k,y_k)}{\partial x} & \dfrac{\partial f(x_k,y_k)}{\partial y} \\ \dfrac{\partial g(x_k,y_k)}{\partial x} & \dfrac{\partial g(x_k,y_k)}{\partial y} \end{pmatrix}^{-1}\begin{pmatrix} -f(x_k,y_k) \\ -g(x_k,y_k) \end{pmatrix}$$

如此迭代直至 $\max\left(|x_{k+1}-x_k|,|y_{k+1}-y_k|\right)<\epsilon$ 为止。将得到的 x 和 y 作为组合系数，算出 $T=x\vec{P}+Y\vec{Q}$ 即为所求，最后调用数值分析时采用的解析法计算出像点 R 即可。

1.2.2 程序实现

首先定义函数：

Listing 3: Functions Definition

```
double function_A(vector P, vector Q, vector T){
    return 2 * InnerProduct(P, Q) * T.cor[0] * T.cor[1] + 2 * pow(Q.
       mol, 2) * pow(T.cor[1], 2) - T.cor[1] - 1 + T.cor[0];
}
```

```
double function_B(vector P, vector Q, vector T){
    return pow(P.mol, 2) * pow(T.cor[0], 2) + 2 * InnerProduct(Q, P) *
        T.cor[0] * T.cor[1] + pow(Q.mol, 2) * pow(T.cor[1], 2) - 1;
}

vector d_function_A(vector P, vector Q, vector T){
    vector df;
    df.cor[0] = 2 * InnerProduct(Q, P) * T.cor[1] + 1;
    df.cor[1] = 2 * InnerProduct(Q, P) * 4 * pow(Q.mol, 2) * T.cor[1]
       - 1;
    return df;
}

vector d_function_B(vector P, vector Q, vector T){
    vector df;
    df.cor[0] = 2 * pow(P.mol, 2) * T.cor[0] + 2 * InnerProduct(P, Q)
       * T.cor[1];
    df.cor[1] = 2 * InnerProduct(Q, P) * T.cor[0] + 2 * pow(Q.mol, 2)
       * T.cor[1];
    return df;
}
```

接下来进行迭代：

Listing 4: Newton iteration

```
vector Newton_iteration(vector T_initial, vector P, vector Q, double
   epsilon){
    vector T = T_initial;
    double delta_x;
    double delta_y;

    do{
        double jacobian[D][D];
        double jacobian_inv[D][D];
        jacobian[0][0] = d_function_A(P, Q, T).cor[0];
        jacobian[0][1] = d_function_A(P, Q, T).cor[1];
        jacobian[1][0] = d_function_B(P, Q, T).cor[0];
        jacobian[1][1] = d_function_B(P, Q, T).cor[1];

        MatrixInv(jacobian, jacobian_inv);
        delta_x = jacobian_inv[0][0] * (-function_A(P, Q, T)) +
```

```
            jacobian_inv[0][1] * (-function_B(P, Q, T));
        delta_y = jacobian_inv[1][0] * (-function_A(P, Q, T)) +
            jacobian_inv[1][1] * (-function_B(P, Q, T));

        T.cor[0] += delta_x;
        T.cor[1] += delta_y;
    }while (abs(delta_x) >= epsilon || abs(delta_y) >= epsilon);

    T.cor[0] = T.cor[0] * P.cor[0] + T.cor[1] * Q.cor[0];
    T.cor[1] = T.cor[0] * P.cor[1] + T.cor[1] * Q.cor[1];

    return T;
}
```

其中 jacobian 矩阵的求逆算法，我使用的是高斯消元法，对目标矩阵和单位阵做同样的初等变换，直至目标矩阵变为单位阵，单位阵变为目标矩阵的逆。

Listing 5: MatrixInv

```
void MatrixInv(double Matrix[][D], double result[][D]){
    result[0][0] = 1;
    result[0][1] = 0;
    result[1][0] = 0;
    result[1][1] = 1;

    double temp = Matrix[1][0];
    Matrix[1][0] += (Matrix[0][0]/Matrix[0][0]) * (-temp);
    Matrix[1][1] += (Matrix[0][1]/Matrix[0][0]) * (-temp);
    result[1][0] += (result[0][0]/Matrix[0][0]) * (-temp);
    result[1][1] += (result[0][1]/Matrix[0][0]) * (-temp);

    temp = Matrix[0][1];
    Matrix[0][0] += (Matrix[1][0]/Matrix[1][1]) * (-temp);
    Matrix[0][1] += (Matrix[1][1]/Matrix[1][1]) * (-temp);
    result[0][0] += (result[1][0]/Matrix[1][1]) * (-temp);
    result[0][1] += (result[1][1]/Matrix[1][1]) * (-temp);

    result[0][0] /= Matrix[0][0];
    result[0][1] /= Matrix[0][0];
    result[1][0] /= Matrix[1][1];
    result[1][1] /= Matrix[1][1];
}
```

最后 R 点的计算方法和前面数值逼近时采取的方法一致，这里不再赘述。

1.2.3 算法缺陷

上述方法的缺陷在于：

(1) 数据精度相较数值逼近法有所提高但仍有一组数据得不到结果，在之后的实验结果部分将会看到；

(2) 矩阵求逆算法的稳定性不够好。

1.2.4 尝试换用 python

在利用 C 处理结果仍不理想后，我尝试换用 python 实现 Newton 迭代算法，整体步骤和上述基本一致，唯一区别便是矩阵求逆算法。在 python 的实现过程中，我调用了 `numpy` 包的求逆方法，稳定性应该有所提高。其次 python 的数据处理精度高于 C，因此实现了所有数据的处理。

2 实验结果

2.1 数值逼近法

```
T: (-0.885670, 0.464316)        R: (-0.380057, 0.674993)
T: (-0.959312, 0.282350)        R: (0.304214, 0.321811)
Processing accuracy is exceeded...  //当 Q 和 P 接近共线时
Processing accuracy is exceeded...  //该方法很难满足处理的精度要求
T: (-0.989279, 0.146038)        R: (1.182424, 0.382590)
T: (-0.922615, 0.385721)        R: (-0.786920, 0.410917)
T: (-0.827028, 0.562160)        R: (8.380296, 2.944148)
T: (-0.987408, 0.158192)        R: (1.187435, 0.329136)
T: (-0.959312, 0.282350)        R: (0.304214, 0.321811)
T: (-0.970066, 0.242842)        R: (7.000894, 0.244735)
```

2.2 Newton iteration

2.2.1 C 语言

```
T: (-0.885670, 0.464316)        R: (-0.380057, 0.674993)
T: (-0.959312, 0.282350)        R: (0.304214, 0.321811)
T: (-0.959592, -0.281394)       R: (-1.316007, 2.200576)
T: (-1.000000, 0.000001)        R: (-1.000000, 0.000001)  //精度不够算不出结果
T: (-0.989279, 0.146038)        R: (1.182424, 0.382590)   //后面的结果都因为
T: (-0.730870, -0.682516)       R: (-1.892229, -0.333199) //求逆算法的不稳定
T: (-0.827028, 0.562160)        R: (8.380296, 2.944148)   //而有较大误差
T: (-0.987408, 0.158192)        R: (1.187435, 0.329136)
```

```
T: (-0.959312, 0.282350)        R: (0.304214, 0.321811)
T: (-0.282156, -0.959368)       R: (-9.456053, -0.950766)
```

2.2.2 Python

```
T:[-0.8856698487824481, 0.46431553813939425]
R:[-0.38005699057773024, 0.6749926934609212]
T:[-0.9593115080061771, 0.2823498372709191]
R:[0.3042141579153159, 0.321811020617353]
T:[-0.999999999998, 1.999982000402126e-06]
R:[7.99991600208827e-06, 1.9999960000199994]
T:[-0.9999999999995, 9.999999999984996e-07]
R:[-0.999999999999, 9.99999999999e-07]
T:[-0.9892790407963629, 0.14603759598482993]
R:[1.1824239075132619, 0.3825896358311274]
T:[-0.9226152820422197, 0.3857214556414452]
R:[-0.7869201371823826, 0.410916850768]
T:[-0.8505029362555685, 0.5259702989909756]
R:[8.585911268465807, 2.835009787518561]
T:[-0.9874084509531342, 0.15819150099272705]
R:[1.1874350420740996, 0.32913615031771126]
T:[-0.9593115080061771, 0.2823498372709191]
R:[0.3042141579153159, 0.321811020617353]
T:[-0.9700657361905205, 0.24284247459854214]
R:[7.000894359310763, 0.24473458704773465]
```

3 思考

从上面的结果可以看出，C 语言环境下当 Q 和 P 相近时，`double` 类型很难满足计算精度要求，因此或许可以通过设计精度更高的数据结构和计算规则来解决这一问题，或者采用二进制进行运算；其次便是求逆算法的不稳定性导致数值偏差与理想值较大，事实上除了本报告中提到的算法，我还尝试了 LU 分解求逆算法，稳定性也并不高，因此优化求逆算法也是需要考虑的问题。