NEPath

A Classical Toolpath and Optimization-Based Non-Equidistant Toolpath Planning Library (In C++)

The **NEPath** library plans toolpaths for additive manufacturing (AM, 3D printing) and CNC milling. Toolpath planning is to generate some 1D toolpaths to filling given 2D slices. The **NEPath** library is able to plan the following toolpaths:

- Optimization-based non-equidistant toolpath:
    - **Isoperimetric-Quotient-Optimal Toolpath (IQOP)**.
    - Variants of IQOP, like toolpaths that minimizing the perimeter, the isoperimetric quotient, and the area.
- Classical toolpath:
    - **Contour-Parallel Toolpath (CP)**.
    - **Zigzag Toolpath**.
    - **Raster Toolpath**.
- Toolpath connection. (Temporarily unavailable)
- Other functions:
    - Tool Compensating.
    - Calculating underfill rate.
    - Determining sharp corners.

Among them, the IQOP is proposed by Wang et al., with an article (accept), i.e.,

```
Yunan Wang, Chuxiong Hu, et al. Optimization-Based Non-Equidistant Toolpath Planning for Robotic Additive Manufacturing with Non-Underfill Orientation[J]. Robotics and Computer-Integrated Manufacturing, 2023. (accept)
```

After the article is published, the **NEPath** library would provide the API and details of IQOP. More non-equidistant toolpaths would be designed soon.

Complier

C++17

Statement and Dependence

- This project cites AngusJohnson/Clipper2 as a dependent package.
- This project depends on Gurobi optimizer for solving quadratically constrained quadratic program with second-order cone constraints. If you need to use another optimizer, you can rewrite the method in the `MyOptimization` function (Temporarily unavailable). If you don't need IQOP and other optimization-based toolpaths, you can comment out `#define IncludeGurobi` in `NEPath-master/setup_NEPath.h` to avoid the dependence on Gurobi.

About Citing

If you need to use the **NEPath** project, please cite "Yunan Wang, Chuxiong Hu, et al. Optimization-Based Non-Equidistant Toolpath Planning for Robotic Additive Manufacturing with Non-Underfill Orientation[J]. Robotics and Computer-Integrated Manufacturing. 2023."

Introduction to IQOP

Framework

IQOP is an optimization-based non-equidistant toolpath planning method for AM and CNC milling. IQOP tries to optimize the smoothness and material cost of the child toolpath from a parent toolpath. IQOP has the following advantages:

- Compared with the equidistant toolpath, i.e., CP, IQOP can generate smooth toolpaths. Specially, toolpaths insides tends to transform into a smooth circle.
- IQOP can be applied for slices with arbitrary shapes and topological holes. Extra toolpaths would be added if underfill with large area exists.
- IQOP achieves obviously lower underfill rates, higher printing efficiency, and higher toolpath smoothness than CP.
- A general framework of non-equidistant toolpath planning for complex slices is provided.

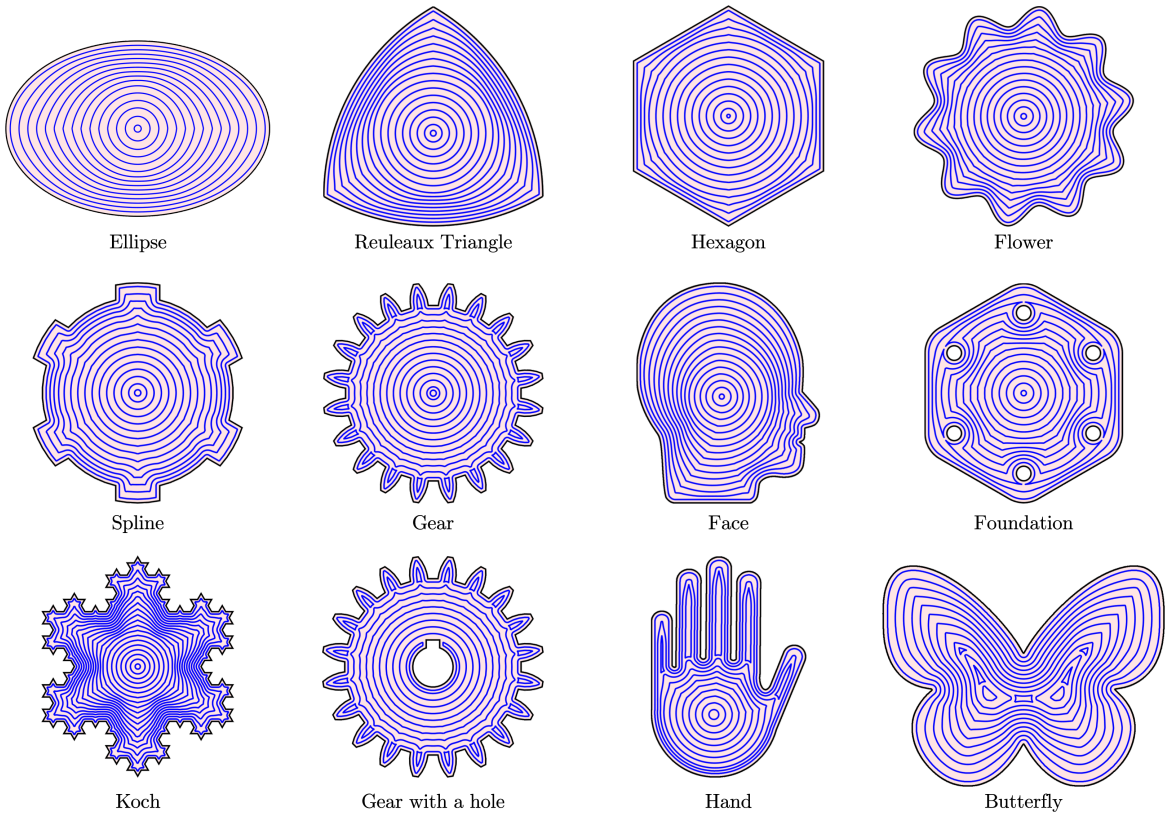

Figure. Some demos of IQOP.

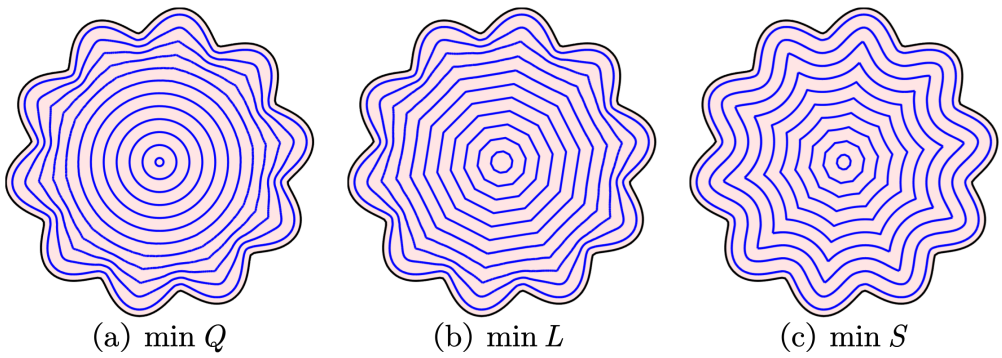

Figure. Toolpaths generated by different object functions.

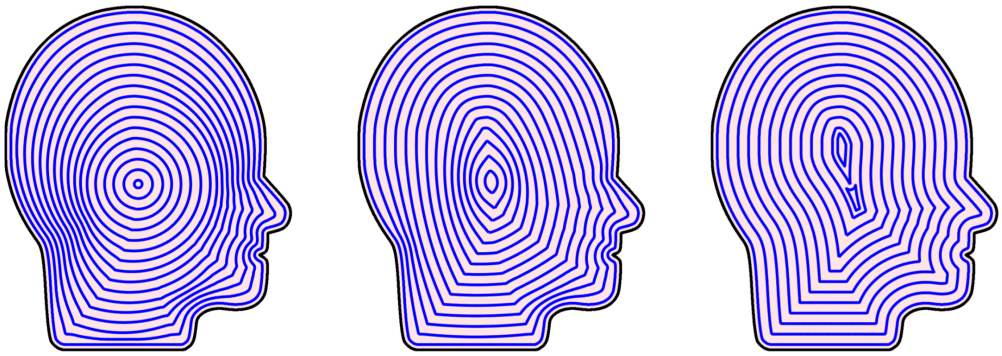

Figure. Toolpaths generated by different weighting coefficient.

More details of IQOP would be provided after the article is published.

Optimization Problem of IQOP

The toolpaths can be planned by offsetting non-equidistantly. The offsetting distances $\{\delta_i\}_{i=1}^{n}$ can be seen as optimization variables. δ_i is the offsetting distance at (x_i, y_i).

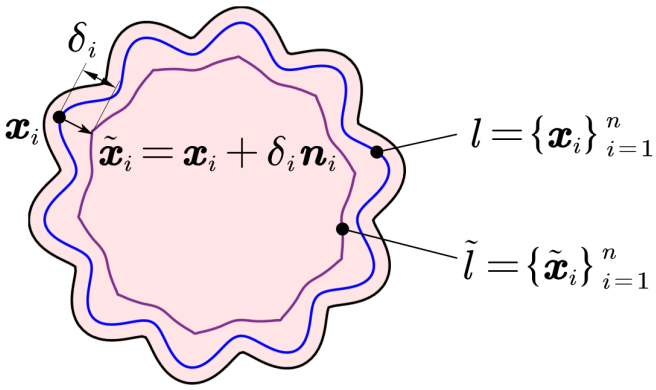

Figure. Optimization variables.

Given l, the optimization problem for generating $\tilde{l}$ can be written as:

$$
\begin{aligned}
\min \quad & \lambda_Q Q + \lambda_S S + \lambda_L L \\
\text{s.t.} \quad & L \text{ is the length of } \tilde{l} \\
& S \text{ is the area of the region enclosed by } \tilde{l} \\
& Q = \frac{L^2}{4\pi S} \text{ is the isoperimetric quotient of } \tilde{l} \\
& \alpha\delta_{\mathrm{m}} \leq \delta_i \leq \delta_{\mathrm{m}}, \forall i \in [n] \\
& \left|\dot{\delta}_i\right| \leq \dot{\delta}_{\mathrm{m}}, \forall i \in [n] \\
& \left|\ddot{\delta}_i\right| \leq \ddot{\delta}_{\mathrm{m}}, \forall i \in [n]
\end{aligned}
$$

In our paper `Optimization-Based Non-Equidistant Toolpath Planning for Robotic Additive Manufacturing with Non-Underfill Orientation`, the above optimization problem is convexified, and the problem of self-intersection is solved. The above method can be applied for slices with arbitrary shapes and topological structures.

API

`NEPath-master/path.h`

- `(struct)path` is a struct to store information of toolpaths. `(double*)path::x` and `(double*)path::y` are waypoints of a toolpath.
- `paths` is a `vector` of `path`, i.e., `typedef vector<path> paths;`

`NEPath-master/NEPathPlanner.h`

The package `NEPathPlanner.h` include the key class of **NEPath**, i.e., `NEPathPlanner`. All operations on toolpath planning is based on `NEPathPlanner`. The API of `NEPathPlanner` is as follows:

- `(void)NEPathPlanner::set_contour()`: Set the contour, i.e., the outer boundary, of the slice. Every slice only have one closed contour. The start point and the end point of the contour are connected by default. If you want to set the outmost toolpath has a distance from the actual outer boundary, you can call `NEPathPlanner::tool_compensate()` with a **negative** distance to obtain the outmost toolpath firstly, and set the obtained outmost toolpath as the boundary of a new slice. See the example of Zigzag and CP for details.
    - `(const double*)x`, `(const double*)y`, `(int)length`: The number of waypoints is `length`. The `i`-th waypoint is

`(x[i],y[i])`. It can be substituted as `(const path&)contour_new`.
    - `(bool)wash`, `(double)wash_dis`, `(int)num_least`: If `wash==true`, the contour would be resampled with a uniformly-distributed distance no more than `wash_dis`, and the number of waypoints are no less than `num_least`. By default, `wash=true, washdis=0.2, num_least=50`.
    - The contour is stored in a public member variable `(path)contour`.
- `(void)NEPathPlanner::addhole()`: Add a new hole, i.e., the inner boundaries, onto the slice. The start point and the end point of every hole are connected by default. A slice is allowed to have no holes. The same as `(void)NEPathPlanner::set_contour()`, you can call `NEPathPlanner::tool_compensate()` to offset the added hole.
    - `(const double*)x`, `(const double*)y`, `(int)length`: The number of waypoints is `length`. The `i`-th waypoint is `(x[i],y[i])`. It can be substituted as `(const path&)hole_new`.
    - `(bool)wash`, `(double)wash_dis`, `(int)num_least`: If `wash==true`, the contour would be resampled with a uniformly-distributed distance no more than `wash_dis`, and the number of waypoints are no less than `num_least`. By default, `wash=true, washdis=0.2, num_least=50`.
    - The holes are stored in a public member variable `(paths)holes`.
- `(void)NEPathPlanner::addholes()`: Add some new holes, i.e., the inner boundaries, onto the slice. The start point and the end point of every hole are connected by default. A slice is allowed to have no holes. The same as `(void)NEPathPlanner::set_contour()`, you can call `NEPathPlanner::tool_compensate()` to offset the added hole.
    - `(const paths&)holes_new`: Add `path`s in `holes_new` onto the slice.
    - `(bool)wash`, `(double)wash_dis`, `(int)num_least`: If `wash==true`, the contour would be resampled with a uniformly-distributed distance no more than `wash_dis`, and the number of waypoints are no less than `num_least`. By default, `wash=true, washdis=0.2, num_least=50`.
    - The holes are stored in a public member variable `(paths)holes`.

- `(paths)NEPathPlanner::tool_compensate()`: Offset the contour and holes of the slice with a distance, i.e., tool compensating.
    - `(const ContourParallelOptions&)opts`:
        - The offsetting distance is `opts.delta`. If `opts.delta>0`, the contour will be offset outside and the holes will be offset inside. If `opts.delta<0`, the contour will be offset inside and the holes will be offset outside.
        - If `opts.wash==true`, the contour would be resampled with a uniformly-distributed distance no more than `opts.wash_dis`, and the number of waypoints are no less than `opts.num_least`.
    - The order of outputs is the offsetting results of `contour`, `holes[0]`, `holes[1]`, ..., `holes[holes.size()-1]`. Note that the offsetting results of each toolpath can be one, serval, or even zero toolpath.
    - `(paths)NEPathPlanner::tool_compensate()` is achieved based on AngusJohnson/Clipper2.
- `(paths)NEPathPlanner::IQOP()`: Generate the **IQOP** toolpath of a slice. The optimization problem of IQOP is provided above. If you don't need IQOP and other optimization-based toolpaths, you can comment out `#define IncludeGurobi` in `NEPath-master/setup_NEPath.h` to avoid the dependence on Gurobi.
    - `(const NonEquidistantOptions&)opts`:
        - `opts.delta` is the maximum distance between toolpaths. `opts.alpha` the scale of the minimum distance. The distances between toolpaths at every point are between `opts.alpha*opts.delta` and `opts.delta`, i.e., $\forall i, \delta_i \in$ (`opts.alpha*opts.delta`, `opts.delta`). `opts.dot_delta` is $\dot{\delta}_{\mathrm{m}}$, i.e., the upper bound of $\frac{\mathrm{d}\delta}{\mathrm{d}s}$. `opts.dot_delta` is $\ddot{\delta}_{\mathrm{m}}$, i.e., the upper bound of $\frac{\mathrm{d}^2\delta}{\mathrm{d}s^2}$.
        - `opts.optimize_Q` is true if Q is in the objective function. `opts.optimize_S` is true if S is in the objective function. `opts.optimize_L` is true if L is in the objective function. `opts.lambda_Q`, `opts.lambda_S`, and `opts.lambda_L` are $\lambda_Q, \lambda_S, \lambda_L$, respectively.

        - `opts.epsilon` is the upper bound of error in $\|\cdot\|_\infty$. `opts.set_max` is the maximum iteration steps.
        - If `opts.wash==true`, the contour would be resampled with a uniformly-distributed distance no more than `opts.wash_dis`, and the number of waypoints are no less than `opts.num_least`.
- `(paths)NEPathPlanner::Raster()`: Generate the **Raster** toolpath of a slice.
    - `(const DirectParallelOptions&)opts`: `opts.delta` is the distance between toolpaths. `opts.angle` is the angle between Raster toolpaths and the x-axis. The unit of `opts.angle` is rad, and you can use `acos(-1.0)` to obtain a accurate $\pi = 3.1415926\cdots$.
    - Every Raster toolpath has two waypoints, i.e., the start point and the end point.
- `(paths)NEPathPlanner::Zigzag()`: Generate the **Zigzag** toolpath of a slice.
    - `(const DirectParallelOptions&)opts`: `opts.delta` is the distance between toolpaths. `opts.angle` is the angle between Zigzag toolpaths and the x-axis. The unit of `opts.angle` is rad, and you can use `acos(-1.0)` to obtain a accurate $\pi = 3.1415926\cdots$.
    - Every Zigzag toolpath has an even numbers of waypoints.
- `(paths)NEPathPlanner::CP()`: Generate the **CP** toolpath of a slice.
    - `(const ContourParallelOptions&)opts`: `opts.delta` is the distance between toolpaths. If `opts.wash==true`, the contour would be resampled with a uniformly-distributed distance no more than `opts.wash_dis`, and the number of waypoints are no less than `opts.num_least`.
    - `(paths)NEPathPlanner::CP()` is achieved based on AngusJohnson/Clipper2.
- Other toolpath generation algorithms and toolpath connection algorithm will be added into `NEPathPlanner` latter.

`NEPath-master/Curve.h`

`Curve.h` has some fundamental methods on geometry.

- **Underfill**. For a slice $D \subset \mathbb{R}^2$ and some toolpaths $\{l_i\}_{i=1}^N$, underfill is defined as $D \bigcap \left(\bigcup_{i=1}^{n} B_{\frac{\delta}{2}}(l_i) \right)^C$, where $\delta > 0$ is the line width.
    - `(static UnderFillSolution)Curve::UnderFill()`: API of calculate underfill. Return a `UnderFillSolution`.
        - `(const path&)contour`: the contour of slice.
        - `(const paths&)holes`: the holes of slice. If the slice has no hole, you can input `paths()` as an empty set of holes.
        - `(const paths&)ps`: the toolpaths planned before.
        - `(double)delta`: the line width δ. Note that for every toolpath, only a width of $\frac{\delta}{2}$ on each side is determined as fill.
        - `(double)reratio`: the resolution ratio. `xs` and `ys` are sampled with a distance of `reratio` between 2 points.
    - `(struct)UnderFillSolution` is a struct to store information of underfill.
        - `(double*)xs` and `(double*)ys` are discrete points on x-axis and y-axis.
        - `(int)nx` and `(int)ny` are the lengths of `xs` and `ys`.
        - `(bool**)map_slice` stores information of slice D. `map_slice[i][j]==true` if and only if the point `(xs[i],ys[j])` $\in D$.
        - `(bool**)map_delta` stores information of neighborhood of toolpaths $\bigcup_{i=1}^{n} B_{\frac{\delta}{2}}(l_i)$. `map_delta[i][j]==true` if and only if the point `(xs[i],ys[j])` $\in \bigcup_{i=1}^{n} B_{\frac{\delta}{2}}(l_i)$.
        - `(double)underfillrate` is the underfill rate, i.e.,

$$\text{underfill rate} = \frac{\text{area of underfill}}{\text{area of slice}} = 1 - \frac{\text{number of pixels in } D \bigcap \left(\bigcup_{i=1}^{n} B_{\frac{\delta}{2}}(l_i) \right)}{\text{number of pixels in } D}.$$

- **Sharp corner.** To avoid computational sensitivity, sharp corners are determined by area invariant (Helmut Pottmann, et al. 2009).
    - `(static SharpTurnSolution)Curve::SharpTurn_Invariant()`: determine sharp corners on a toolpath:

        - `(const path&)p`: the input toolpath.
        - `(double)radius`: the radius of the rolling circle.
        - `(double)threshold`: the threshold to determine a sharp corner.
        - `(bool)close`: `close` is true if and only if the toolpath is closed.
        - `(bool)washdis`: sharp corners would be determined with a uniformly-distributed distance no more than `washdis`.
    - `(struct)SharpTurnSolution` is a struct to store information of sharp corners for a toolpath `p`.
        - `(int)length`: length of the toolpath.
        - `(double)radius`: the radius of the rolling circle.
        - `(double)threshold`: the threshold to determine a sharp corner.
        - `(double*)AreaPercent`: `AreaPercent[i]` is the percent of area on one side of the toolpath at `(p.x[i],p.y[i])`.
        - `(bool*)SharpTurn`: `SharpTurn[i]==ture` if and only if `AreaPercent[i]>threshold`.
        - `(bool)close`: `close` is true if and only if the toolpath is closed.

Examples

Toolpath Generation

IQOP (Isoperimetric-Quotient-Optimal Toolpath, Wang Y et al., 2023)

```
    NEPathPlanner planner;

    // Obtain the contour of the outer boundary of slices
    path contour;
    contour.length = 1000; // the number of waypoints
    contour.x = new double[contour.length](); // x-coordinate of waypoints
    contour.y = new double[contour.length](); // y-coordinate of waypoints
    const double pi = acos(-1.0); // pi == 3.1415926...
    for (int i = 0; i < contour.length; ++i) {
```

```
        double theta = 2.0 * pi * i / contour.length;
        double r = 15.0 * (1.0 + 0.1 * cos(10.0 * theta));
        contour.x[i] = r * cos(theta);
        contour.y[i] = r * sin(theta);
    }

    // The out boundary should be offset with half of the line width to obtain the outmost toolpath
    NEPathPlanner planner_toolcompensate;
    planner_toolcompensate.set_contour(contour);
    ContourParallelOptions opts_toolcompensate;
    opts_toolcompensate.delta = -1.0 * 0.5; // half of the line width of toolpaths
    opts_toolcompensate.wash = true; // it is recommended to set opt.wash=true
    // if wash==true, then all toolpaths would have yniformly distributed waypoints, with a distance near opts.washdis
    opts_toolcompensate.washdis = 0.2;
    paths path_outmost = planner_toolcompensate.tool_compensate(opts_toolcompensate);

    planner.set_contour(path_outmost[0]);
    // or `planner.set_contour(contour.x, contour.y, contour.length)`

    // Set the toolpath parameters
    NonEquidistantOptions opts;
    opts.delta = 1.0; // the line width of toolpaths
    opts.alpha = 0.5; // the scale of minimum distance
    opts.dot_delta = 1.0; // the upper bound of \dot{delta_i}
    opts.ddot_delta = 0.1; // the upper bound of \ddot{delta_i}

    opts.optimize_Q = true; // the isoperimetric quotient is in the objective function
    opts.optimize_S = false; // the area is not in the objective function
    opts.optimize_L = false; // the length is not in the objective function
    opts.lambda_Q = 1.0; // the weighting coefficient of the isoperimetric quotient

    opts.wash = true; // it is recommended to set opt.wash=true
```

```
    // if wash==true, then all toolpaths would have yniformly distributed waypoints, with a distance near opts.washdis
    opts.washdis = 0.2;


    paths IQOP_paths = planner.IQOP(opts, true); // all IQOP paths
    cout << "There are " << IQOP_paths.size() << " continuous toolpaths in total." << endl;
```

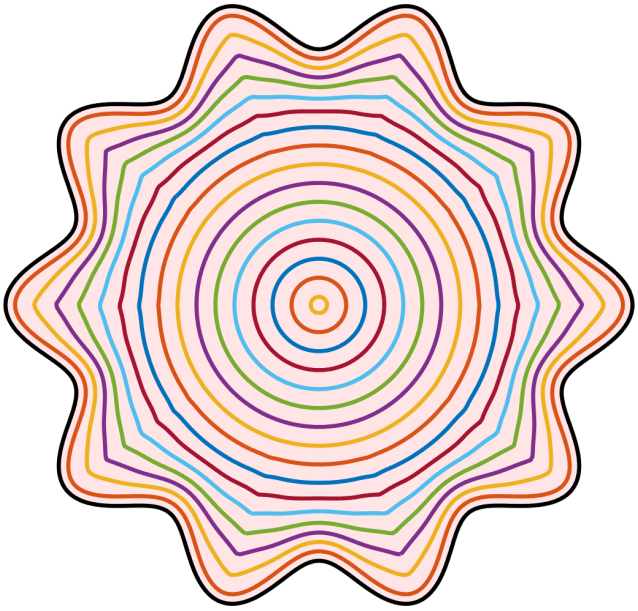

Figure. IQOP toolpath minimizing Q.

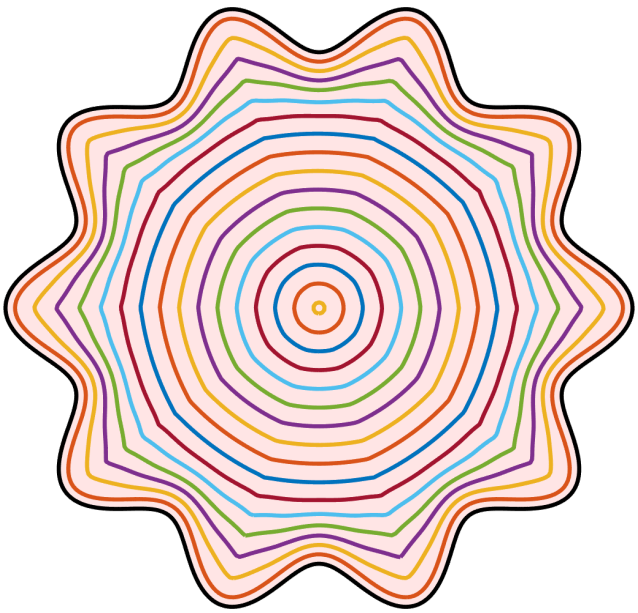

Figure. IQOP toolpath minimizing Q+1.0S.

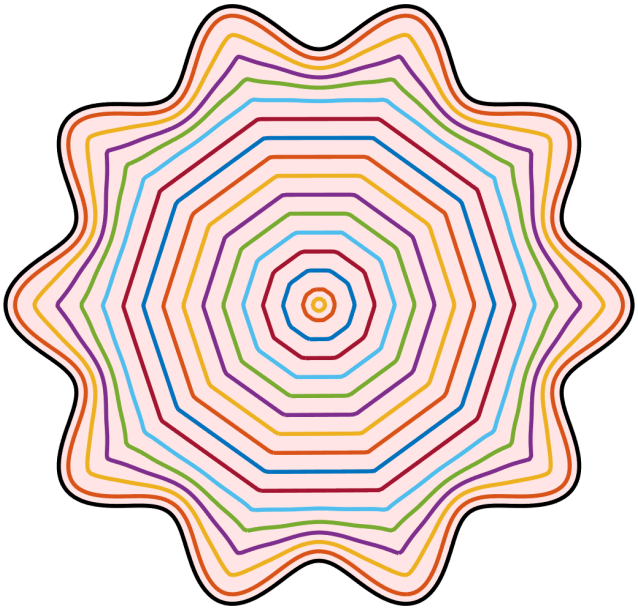

Figure. IQOP toolpath minimizing L.

CP (Contour-Parallel)

```
#include "NEPath-master/NEPathPlanner.h"
#include <iostream>
using namespace std;

int main() {
    NEPathPlanner planner;

    // Obtain the contour of the outer boundary of slices
    path contour;
    contour.length = 1000; // the number of waypoints
    contour.x = new double[contour.length](); // x-coordinate of waypoints
    contour.y = new double[contour.length](); // y-coordinate of waypoints
    const double pi = acos(-1.0); // pi == 3.1415926...
    for (int i = 0; i < contour.length; ++i) {
        double theta = 2.0 * pi * i / contour.length;
        double r = 15.0 * (1.0 + 0.15 * cos(10.0 * theta));
        contour.x[i] = r * cos(theta);
```

```
        contour.y[i] = r * sin(theta);
    }

    // The out boundary should be offset with half of the line width to obtain the outmost toolpath
    NEPathPlanner planner_toolcompensate;
    planner_toolcompensate.set_contour(contour);
    ContourParallelOptions opts_toolcompensate;
    opts_toolcompensate.delta = -1.0 * 0.5; // half of the line width of toolpaths
    opts_toolcompensate.wash = true; // it is recommended to set opt.wash=true
    // if wash==true, then all toolpaths would have yniformly distributed waypoints, with a distance near opts.washdis
    opts_toolcompensate.washdis = 0.2;
    paths path_outmost = planner_toolcompensate.tool_compensate(opts_toolcompensate);

    planner.set_contour(path_outmost[0]);
    // or `planner.set_contour(contour.x, contour.y, contour.length)`

    // Set the toolpath parameters
    ContourParallelOptions opts;
    opts.delta = 1.0; // the line width of toolpaths
    opts.wash = true; // it is recommended to set opt.wash=true
    // if wash==true, then all toolpaths would have yniformly distributed waypoints, with a distance near opts.washdis
    opts.washdis = 0.2;

    paths CP_paths = planner.CP(opts); // all CP paths
    cout << "There are " << CP_paths.size() << " continuous toolpaths in total." << endl;
    for (int i = 0; i < CP_paths.size(); ++i) {
        // CP_paths[i] is the i-th continuous toolpath
        cout << "Toopath " << i << " has " << CP_paths[i].length << " waypoints." << endl;
    }

    return 0;
}
```

Figure. CP toolpath.

Zigzag

```
#include "NEPath-master/NEPathPlanner.h"
#include <iostream>
using namespace std;

int main() {
    NEPathPlanner planner;

    // Set the contour
    path contour;
    contour.length = 1000; // the number of waypoints
    contour.x = new double[contour.length](); // x-coordinate of waypoints
    contour.y = new double[contour.length](); // y-coordinate of waypoints
    const double pi = acos(-1.0); // pi == 3.1415926...
    for (int i = 0; i < contour.length; ++i) {
        double theta = 2.0 * pi * i / contour.length;
        double r = 15.0 * (1.0 + 0.15 * cos(10.0 * theta));
        contour.x[i] = r * cos(theta);
        contour.y[i] = r * sin(theta);
    }
    planner.set_contour(contour);
```

```
    // or `planner.set_contour(contour.x, contour.y, contour.length)`

    // Set the toolpath parameters
    DirectParallelOptions opts;
    opts.delta = 1.0; // the line width of toolpaths
    opts.angle = pi / 3.0; // the angle of Zigzag toolpaths, unit: rad

    paths zigzag_paths = planner.Zigzag(opts); // all zigzag paths
    cout << "There are " << zigzag_paths.size() << " continuous toolpaths in total." << endl;
    for (int i = 0; i < zigzag_paths.size(); ++i) {
        // zigzag_paths[i] is the i-th continuous toolpath
        cout << "Toopath " << i << " has " << zigzag_paths[i].length << " waypoints." << endl;
    }

    return 0;
}
```

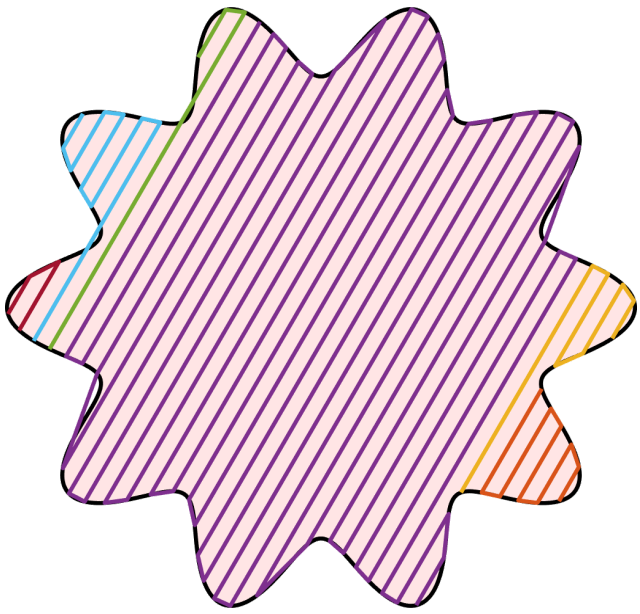

Figure. Zigzag toolpath.

Raster

```cpp
#include "NEPath-master/NEPathPlanner.h"
#include <iostream>
using namespace std;

int main() {
    NEPathPlanner planner;

    // Set the contour
    path contour;
    contour.length = 1000; // the number of waypoints
    contour.x = new double[contour.length](); // x-coordinate of waypoints
    contour.y = new double[contour.length](); // y-coordinate of waypoints
    const double pi = acos(-1.0); // pi == 3.1415926...
    for (int i = 0; i < contour.length; ++i) {
        double theta = 2.0 * pi * i / contour.length;
        double r = 15.0 * (1.0 + 0.15 * cos(10.0 * theta));
        contour.x[i] = r * cos(theta);
        contour.y[i] = r * sin(theta);
    }
    planner.set_contour(contour);
    // or `planner.set_contour(contour.x, contour.y, contour.length)`

    // Set the toolpath parameters
    DirectParallelOptions opts;
    opts.delta = 1.0; // the line width of toolpaths
    opts.angle = - pi / 3.0; // the angle of raster toolpaths, unit: rad

    paths raster_paths = planner.Raster(opts); // all raster paths
    cout << "There are " << raster_paths.size() << " continuous toolpaths in total." << endl;

    return 0;
}
```

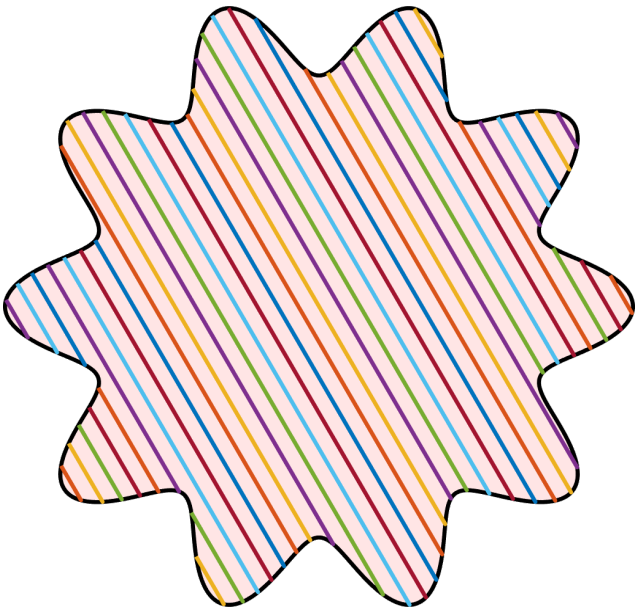

Figure. Raster toolpath.

Toolpath Connection

The API and examples of toolpath connection would be available soon.

Others

Tool compensate

```
#include "NEPath-master/NEPathPlanner.h"
#include <iostream>
using namespace std;

int main() {
    NEPathPlanner planner;

    // Obtain the contour of the outer boundary of slices
    path contour;
    contour.length = 1000; // the number of waypoints
    contour.x = new double[contour.length](); // x-coordinate of waypoints
    contour.y = new double[contour.length](); // y-coordinate of waypoints
    const double pi = acos(-1.0); // pi == 3.1415926...
    for (int i = 0; i < contour.length; ++i) {
        double theta = 2.0 * pi * i / contour.length;
```

```
        double r = 15.0 * (1.0 + 0.15 * cos(10.0 * theta));
        contour.x[i] = r * cos(theta);
        contour.y[i] = r * sin(theta);
    }
    planner.set_contour(contour);

    // Obtain the hole
    double x_hole[] = { -5,5,5,0,-5 };
    double y_hole[] = { -5,-5,5,0,5 };
    planner.addhole(x_hole, y_hole, 5);

    // Tool compensate
    ContourParallelOptions opts;
    opts.delta = -1.5; // the offset distance
    opts.wash = true; // it is recommended to set opt.wash=true
    // if wash==true, then all toolpaths would have yniformly distributed waypoints, with a distance near opts.washdis
    opts.washdis = 0.2;
    paths ps_toolcompensate = planner.tool_compensate(opts); // Tool compensate

    cout << "There are " << ps_toolcompensate.size() << " continuous toolpaths in total." << endl;
    for (int i = 0; i < ps_toolcompensate.size(); ++i) {
        // ps_toolcompensate[i] is the i-th continuous toolpath
        cout << "Toopath " << i << " has " << ps_toolcompensate[i].length << " waypoints." << endl;
    }

    return 0;
}
```

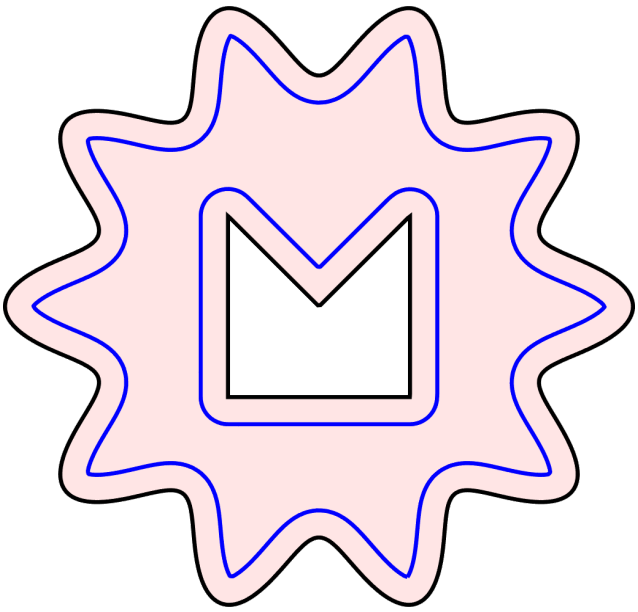

Figure. Tool compensate.

Underfill

```
#include "NEPath-master/NEPathPlanner.h"
#include <iostream>
using namespace std;

int main() {
    NEPathPlanner planner;

    // Obtain the contour of the outer boundary of slices
    path contour;
    contour.length = 1000; // the number of waypoints
    contour.x = new double[contour.length](); // x-coordinate of waypoints
    contour.y = new double[contour.length](); // y-coordinate of waypoints
    const double pi = acos(-1.0); // pi == 3.1415926...
    for (int i = 0; i < contour.length; ++i) {
        double theta = 2.0 * pi * i / contour.length;
        double r = 15.0 * (1.0 + 0.15 * cos(10.0 * theta));
        contour.x[i] = r * cos(theta);
        contour.y[i] = r * sin(theta);
    }

```

```
    // The out boundary should be offset with half of the line width to obtain the outmost toolpath
    NEPathPlanner planner_toolcompensate;
    planner_toolcompensate.set_contour(contour);
    ContourParallelOptions opts_toolcompensate;
    opts_toolcompensate.delta = -1.0 * 0.5; // half of the line width of toolpaths
    opts_toolcompensate.wash = true; // it is recommended to set opt.wash=true
    // if wash==true, then all toolpaths would have yniformly distributed waypoints, with a distance near opts.washdis
    opts_toolcompensate.washdis = 0.2;
    paths path_outmost = planner_toolcompensate.tool_compensate(opts_toolcompensate);

    planner.set_contour(path_outmost[0]);
    // or `planner.set_contour(contour.x, contour.y, contour.length)`

    // Set the toolpath parameters
    ContourParallelOptions opts;
    opts.delta = 1.0; // the line width of toolpaths
    opts.wash = true; // it is recommended to set opt.wash=true
    // if wash==true, then all toolpaths would have yniformly distributed waypoints, with a distance near opts.washdis
    opts.washdis = 0.2;

    paths CP_paths = planner.CP(opts); // all CP paths

    double delta_underfill = opts.delta; // the line width for underfill computation
    double reratio = 0.03; // resolution ratio for underfill computation

    UnderFillSolution ufs = Curve::UnderFill(contour, paths(), CP_paths, delta_underfill, reratio); // Obtain the results of underfill

    cout << "The underfill rate is " << ufs.underfillrate * 100 << "%." << endl;

    return 0;
}
```

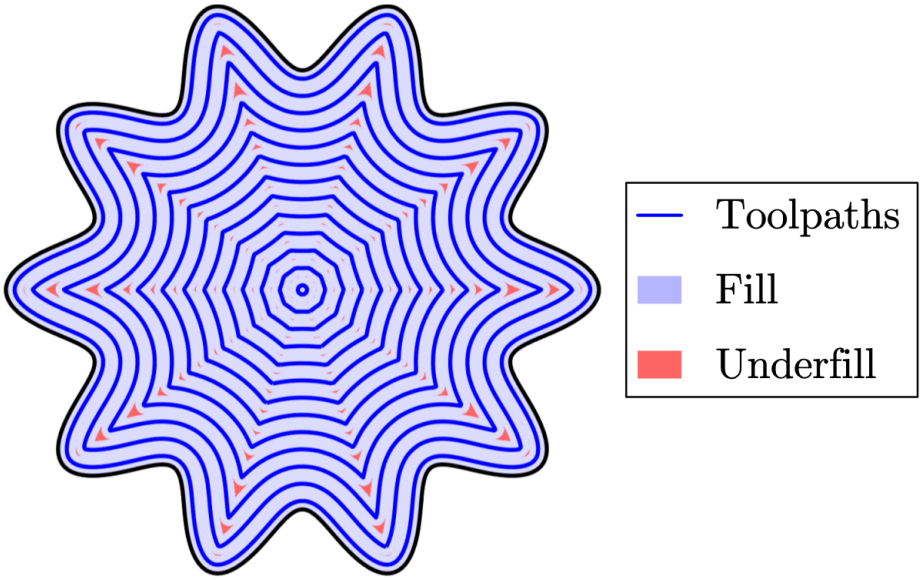

Figure. Underfill. The underfill rate is 1.2401% in this example.

Sharp corner

```
#include "NEPath-master/NEPathPlanner.h"
#include <iostream>
using namespace std;

int main() {
    NEPathPlanner planner;

    // Obtain the contour of the outer boundary of slices
    path contour;
    contour.length = 1000; // the number of waypoints
    contour.x = new double[contour.length](); // x-coordinate of waypoints
    contour.y = new double[contour.length](); // y-coordinate of waypoints
    const double pi = acos(-1.0); // pi == 3.1415926...
    for (int i = 0; i < contour.length; ++i) {
        double theta = 2.0 * pi * i / contour.length;
        double r = 15.0 * (1.0 + 0.15 * cos(10.0 * theta));
        contour.x[i] = r * cos(theta);
        contour.y[i] = r * sin(theta);
    }
```

```
20
21      // The out boundary should be offset with half of the
    line width to obtain the outmost toolpath
22      NEPathPlanner planner_toolcompensate;
23      planner_toolcompensate.set_contour(contour);
24      ContourParallelOptions opts_toolcompensate;
25      opts_toolcompensate.delta = -1.0 * 0.5; // half of the
    line width of toolpaths
26      opts_toolcompensate.wash = true; // it is recommended to
    set opt.wash=true
27      // if wash==true, then all toolpaths would have yniformly
    distributed waypoints, with a distance near opts.washdis
28      opts_toolcompensate.washdis = 0.2;
29      paths path_outmost =
    planner_toolcompensate.tool_compensate(opts_toolcompensate);
30
31      planner.set_contour(path_outmost[0]);
32      // or `planner.set_contour(contour.x, contour.y,
    contour.length)`
33
34      // Set the toolpath parameters
35      ContourParallelOptions opts;
36      opts.delta = 1.0; // the line width of toolpaths
37      opts.wash = true; // it is recommended to set
    opt.wash=true
38      // if wash==true, then all toolpaths would have yniformly
    distributed waypoints, with a distance near opts.washdis
39      opts.washdis = 0.2;
40
41      paths CP_paths = planner.CP(opts); // all CP paths
42
43      double radius = 1.0; // radius of the rolling circle
44      double threshold = 0.3; // threshold of area on one side
    to determine a sharp corner
45
46      // Obtain the results of underfill
47      int num = 0;
48      for (int i = 0; i < CP_paths.size(); ++i) {
49          SharpTurnSolution sol =
    Curve::SharpTurn_Invariant(CP_paths[i], radius, threshold,
    true, 0.5);
50          for (int j = 0; j < sol.length; ++j) {
51              num += sol.SharpTurn[j];
52          }
53      }
```

```
54
55      cout << "There exist " << num << " sharp corners." <<
    endl;
56
57      return 0;
58  }
```

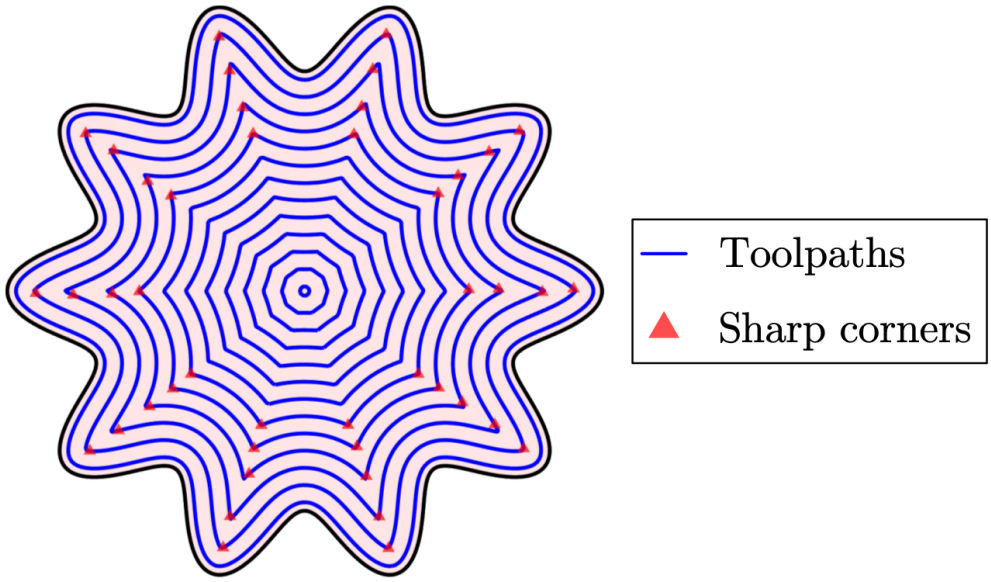

Figure. Sharp corners. There exist 44 sharp corners in this example.