

# Visualize Force closure for 2D Robotic Grasp

Ghoshan Jaganathamani<sup>1</sup>

**Abstract**—This paper presents an approach to robotic grasping analysis. It shows how to check for feasibility of the grasp considered and then measure the quality of the grasp.

## I. INTRODUCTION

Grasping and manipulation with complex grippers, such as multifingered and/or under actuated hands, is an active research area in robotics. The goal of a grasp is to achieve a desired object constraint in the presence of external disturbances (including the objects own weight). Dexterous manipulation involves changing the objects position with respect to the hand without any external support.

## II. RELATED WORK

In this paper our focus is the interaction between the robot and its environment. The desired behavior of the robot hand or end-effector, the contact interface between the robot and objects. The focus is on manipulation. We will assume that the manipulator, objects, and obstacles in the environment are rigid. The key aspects of robotics grasping are that,

### A. Disturbance resistance

A grasp can resist disturbances in any direction when object immobility is ensured, either by finger positions (form closure) or, up to a certain magnitude, by the forces applied by the fingers (force closure). Main problem: determination of contact points on the object boundary.

### B. Dexterity

A grasp is dexterous if the hand can move the object in a compatible way with the task to be performed. When there are no task specifications, a grasp is considered dexterous if the hand is able to move the object in any direction. Main problem: determination of hand configuration.

### C. Equilibrium

A grasp is in equilibrium when the resultant of forces and torques applied on the object (by the fingers and external disturbances) is null. Main problem: determination and control of the proper contact forces.

### D. Stability

A grasp is stable if any error in the object position caused by a disturbance disappears in time after the disturbance vanishes. Main problem: control of restitution forces when the grasp is moved away from equilibrium.

<sup>\*</sup>This work was not supported by any organization

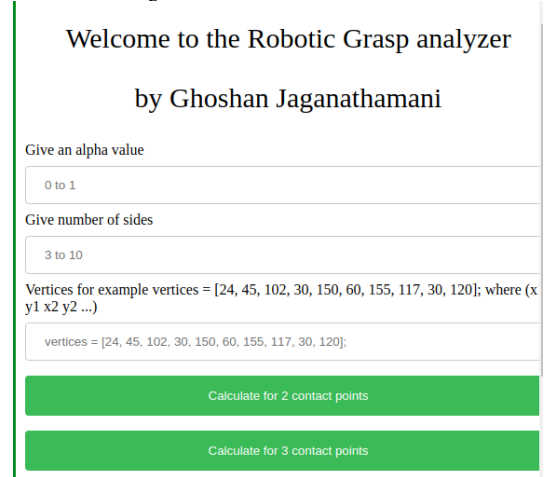
<sup>1</sup>Ghoshan Jaganathamani is a 3rd year Computer Engineering Student, Hong Kong University of Science and Technology, Hong Kong [github.com/ghoshanjega](https://github.com/ghoshanjega)

## III. UI

The user of the program is greeted with a simple website built using HTML and JavaScript for maximum portability. By using JavaScript the program can be embedded in websites, allowing it to be highly portable. The sketching library used in this case is `two.js`, a simple 2D sketching suite. The user for example can input a set of instructions in the respective boxes in the following manner.

```
alpha : 0.7,  
number of sides : 5,  
vertices: 24 45 102 30 150 60 155 117 30 120
```

Fig. 1. Shows the user interface



## IV. 2 CONTACT POINT GRASP

The user inputs the friction coefficient (alpha) and the number of sides the polygon should be. Then the user needs to input the x,y vertices coordinates of the polygon. After the polygon is constructed(make sure the polygon is not self intersecting), the polygon will be displayed on the screen.

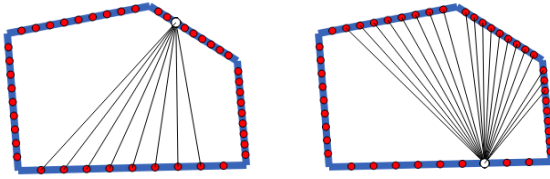
After the polygon sides are constructed, each side has evenly spaced 9 points. So if it is a 5 sided polygon, there will be totally 45 ( 9 \* 5 ) points. Then a point is chosen in random using the `Math.rand()` function. The selected point `p1` will then be checked with all the other points for force closure.

The wrench matrix is constructed from the 2 contact points. The coefficient of friction produces the alpha angle. The two contact points yield force close if

- rank  $F = n$
- there exists a solution to linear optimization to find the minimum for the given wrench matrix.

If the given two contact points yield force closure, a line is drawn between the points indicating compatibility. The points are stored in a multidimensional array where `points[i] = [x coord, y coord, colour, gradient of line, force closure]`. The results are updated on each calculation step. The user is able to click on the "Calculate for 2 contact points" button repeatedly to find another random point and perform the analysis on the given point.

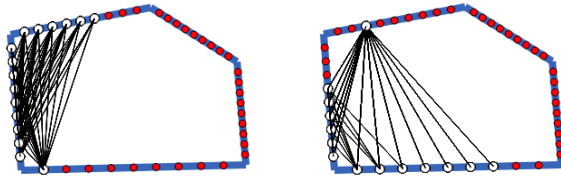
Fig. 2. Shows 2 instances of 2 contact points calculation



### V. 3 CONTACT POINT GRASP

Similar to 2 contact point, 3 contact point is calculated with a given random point and 2 compatible points are chosen. This is done recursively when 2 contact points are compatible.

Fig. 3. Shows 2 instances of 3 contact points calculation

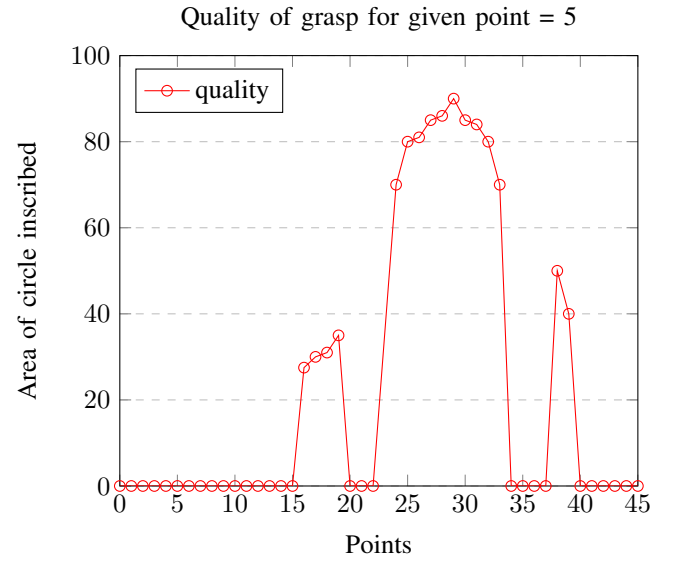


### VI. QUALITY OF GRASP

The quality of the grasp can be measured in many ways. The most accurate way to measure the quality would be to create a 3D Sphere between the contact points 3D wrench space. Since we assume for 2D space, the way to calculate the quality is to find the area of the circle inscribed between the wrench space of the compatible 2 or 3 contact points.

$$Q_{VOP} = \text{Area}(\mathcal{P}) \quad (1)$$

A plot of for a given point is visualized as follows, where the dependant axis shows all the points that are compare, and independent axis has the the area of the circle inscribed by the wrench matrices.



Another quality measure proposed for polyhedral objects and based on a set ICRS is given by the sum of the distances between each one of the  $i$ -th actual contact points  $(x_i, y_i)$  and the center of the corresponding independent contact region  $(x_{i0}, y_{i0}, z_{i0})$ , i.e.

$$Q_{ICR''} = \frac{1}{n} \sum_{i=1}^n \sqrt{(x_i - x_{i0})^2 + (y_i - y_{i0})^2} (2)$$

### VII. CONCLUSION

This paper has presented a simple way to visualize and analyse grasping a 2D polygon with 2 or 3 contact points. The most widely used grasp classifications and closure properties can all be derived from these models under the rigid-body assumption. Linearizing these models leads to metrics and tests that can be computed efficiently using computational linear algebra and linear programming techniques.

Other important research topics within the area of grasping are: grasp synthesis, force distribution, stability, and dexterous manipulation. Grasp synthesis is the problem of choosing the posture of the hand and contact point locations to optimize a grasp quality metric.

The next step would be to prove the stability of the optimal grasp. This could be done using a simulation of external forces acting on the grasp. To further improve on the grasp, it would be ideal to check the dexterity for better motion planning. Dexterity of the grasp is ability of the hand to move the object in a compatible way with the task to be performed.

### VIII. REFERENCES

- Grasp quality measures: review and performance, (Article) <https://link.springer.com/article/10.1007/s10514-014-9402->
- Modern Robotics, (Textbook) <http://hades.mech.northwestern.edu/index.php/ModernRobotics>
- Force Closure grasp, (Article) <http://www.centropiaggio.unipi.it/sites/default/files/grasp-IJRR95.pdf>

- Baker, B.S., Fortune, S., and Grosse, E. 1985. Stable Prehension with a Multifinngered Hand. Proc. IEEE Int. Conf. on Robotics and Automation, pp. 570575.
- Fearing, R. 1986. Simpli  
ed Grasping and Manipulation with Dextrous Robot Hands. IEEE J. Robotics and Automation, vol. 2, no. 4, pp. 188195.

The next file is index.html

```
1
2 <!DOCTYPE html>
3 <html>
4
5 <head>
6   <meta charset="utf-8" />
7   <meta http-equiv="X-UA-Compatible" content="IE=edge">
8   <title>Page Title</title>
9   <meta name="viewport" content="width=device-width, initial-scale=1">
10  <!-- <script src='js/require.js'></script> -->
11  <style>
12    #instructions {
13      font-family: Times New Roman, Times, serif;
14      font-size: 30px;
15      text-align: center
16    }
17
18    body {
19      margin: auto;
20      border: 3px solid green;
21      padding: 10px;
22      justify-content: center;
23    }
24
25    input[type=text],
26    select {
27      width: 100%;
28      padding: 12px 20px;
29      margin: 8px 0;
30      display: inline-block;
31      border: 1px solid #ccc;
32      border-radius: 4px;
33      box-sizing: border-box;
34    }
35
36    input[type=submit] {
37      width: 100%;
38      background-color: #4CAF50;
39      color: white;
40      padding: 14px 20px;
41      margin: 8px 0;
42      border: none;
43      border-radius: 4px;
44      cursor: pointer;
45    }
46
47
48
49
50    /* input[type=submit]:hover {
51      background-color: #45a049;
52    } */
53  </style>
54  <!-- <link rel="stylesheet" type="text/css" media="screen" href="main.css" /> -->
55  <script type="text/javascript" src="js/two.js"></script>
56  <script type="text/javascript" src="js/SimplexJS.js"></script>
57  <!-- <script src="https://cdnjs.cloudflare.com/ajax/libs/p5.js/0.6.1/p5.js"></script> -->
58
59 </head>
60
61 <body style="align-content: center; width: 100%">
62   <p id="instructions">Welcome to the Robotic Grasp analyzer</p>
63
64
65   <div style>
66     <label for="fname">Give an alpha value</label>
67     <input type="text" id="input1" placeholder="0 to 1" />
68     <label for="fname">Give number of sides</label>
69     <input type="text" id="input2" placeholder="3 to 10" />
70     <label for="fname">Vertices for example vertices = [24, 45, 102, 30, 150, 60, 155, 117, 30,
71       120]; where (x1 y1 x2 y2 ...)</label>
72     <input type="text" id="input3" placeholder="vertices = [24, 45, 102, 30, 150, 60, 155, 117,
73       30, 120];" defaultValue="24 45 102 30 150 60 155 117 30 120" />
74     <!-- <script type="text/javascript" src="js/bundle.js"></script> -->
```

```

73     <input type="submit" onclick="myJsFunction()" value="Calculate for 2 contact points"></input
74     <input type="submit" onclick="myJsFunction3()" value="Calculate for 3 contact points"></
       input>
75 </div>
76 <div style="padding-top: 30px; padding-left: 90px" id="draw-shapes"></div>
77 <script type="text/javascript" src="js/2contactpoint.js"></script>
78 <script type="text/javascript" src="js/3contactpoint.js"></script>
79
80
81 </body>
82
83 </html>

```

---

The next file is 2contactpoint.js.

```

1
2 var alpha, numberofsides, vertices = [];
3 var sections = 10;
4 var points = [];
5 var white = "#FFFFFF";
6 var red = "#FF0000";
7
8 var elem = document.getElementById('draw-shapes');
9 var params = { width: 500, height: 400 };
10 var two = new Two(params).appendTo(elem);
11
12 function isEmpty(str) {
13     return (!str || 0 === str.length);
14 }
15
16 function myJsFunction() {
17     two.clear();
18     points = [];
19     alpha = document.getElementById('input1').value;
20     numberofsides = document.getElementById('input2').value;
21     if(!document.getElementById('input3').value)
22         vertices = [24, 45, 102, 30, 150, 60, 155, 117, 30, 120];
23     else
24         vertices = ((document.getElementById('input3').value).split(" "));
25
26     console.dir(vertices);
27     console.log(alpha,numberofsides,vertices.length,vertices,document.getElementById('input3').value
28     );
29     if (drawusershape()) {
30         console.log("shape draw called");
31     }
32 }
33
34
35 var usershape = [];
36 function drawusershape() {
37     for (var i = 0, y = 0; i <= vertices.length-3; i += 2, y++) {
38         var x1 = vertices[i] * 2;
39         var y1 = vertices[i + 1] * 2;
40         var x2 = vertices[i + 2] * 2;
41         var y2 = vertices[i + 3] * 2;
42         console.log(x1, y1, x2, y2);
43         var line = two.makeLine(x1, y1, x2, y2);
44         line.linewidth = 8;
45         line.stroke = '#1C75BC';
46         two.update();
47         makepoints(x1, y1, x2, y2);
48     }
49 }
50
51 var finalline = two.makeLine(vertices[vertices.length - 2] * 2, vertices[vertices.length - 1] *
52     2, vertices[0] * 2, vertices[1] * 2);
53 finalline.linewidth = 8;
54 finalline.stroke = '#1C75BC';
55 two.update();
56 makepoints( vertices[0] * 2, vertices[1] * 2,vertices[vertices.length - 2] * 2, vertices[
57     vertices.length - 1] * 2);
58 console.log(points);
59 two.update();

```

```

58     drawpoints();
59     runcalc();
60     return 1;
61 }
62 }
63
64 console.log(red);
65 function makepoints(x1, y1, x2, y2) {
66     for (var e = 1; e < sections; e++) {
67         var gradient = (y2 - y1) / (x2 - x1);
68         console.log(y2, " - ", y1, " ) / ( ", x2, " - ", x1, " ", -gradient);
69         var angle = Math.tan(-1/gradient);
70         points.push([x1 + ((x2 - x1) / sections) * e, y1 + ((y2 - y1) / sections) * e, red, -
            gradient]); //x coord, y coord, colour, gradient
71     }
72 }
73
74 function drawpoints() {
75     for (var i = 0; i < points.length; i++) {
76         //console.log(points[i], points[i][0], 1, points[i][0][0])
77         var thispoint = two.makeCircle(points[i][0], points[i][1], 4);
78         thispoint.fill = points[i][2];
79     }
80     two.update();
81 }
82
83 function updatepoint(thepoint) {
84     var thispoint = two.makeCircle(points[thepoint][0], points[thepoint][1], 4);
85     thispoint.fill = points[thepoint][2];
86     two.update();
87 }
88
89 function getRandomInt(min, max) {
90     return Math.floor(Math.random() * (max - min + 1)) + min;
91 }
92
93 function runcalc() {
94     var selectedpoint = getRandomInt(0, points.length-1);
95     console.log(selectedpoint);
96     points[selectedpoint][2] = "#9932CC";
97     updatepoint(selectedpoint);
98     //var perpofpoint = -1/points[selectedpoint][3]; //perpendicular to the given point
99     for(var i = 0; i < points.length; i++)
100     {
101         if(i != selectedpoint)
102             selectedtwopoints(selectedpoint, i);
103     }
104 }
105
106
107 }
108
109 function selectedtwopoints(apoint, bpoint) {
110     var p1 = apoint, p2 = bpoint;
111     var cond1 = false;
112     var cond2 = false;
113     // if(points[apoint][3] != points[bpoint][3]) {
114     //     p1 = apoint;
115     //     p2 = bpoint;
116     // }
117     // else {
118     //     p1 = bpoint;
119     //     p2 = apoint;
120     // }
121
122     var aa = two.makeCircle(points[p1][0], points[p1][1], 5);
123     var alpha1gradient = (Math.tan(Math.atan(-1/(points[p1][3])) + Math.atan(alpha)));
124     var alpha2gradient = (Math.tan(Math.atan(-1/(points[p1][3])) - Math.atan(alpha)));
125     var gradientbetween = -(points[p2][1] - points[p1][1]) / (points[p2][0] - points[p1][0]);
126     //console.log(Math.tan(-1/(points[p1][3])), Math.tan(alpha));
127     console.log(points[p2][1], "-", points[p1][1], " ) / ( ", points[p2][0], "-", points[p1][0],
        gradientbetween, alpha1gradient, alpha2gradient);
128     if(Math.sign(alpha1gradient) == Math.sign(gradientbetween)) {
129         if(Math.abs(gradientbetween) > Math.abs(alpha1gradient)) {
130             console.log("Condition 1 satisfied");

```

```

131         cond1 = true;
132     }
133 }
134 if(Math.sign(alpha2gradient)==Math.sign(gradientbetween)){
135     if(Math.abs(gradientbetween)>Math.abs(alpha2gradient)){
136         console.log("conditon 1 satisfied");
137         cond1 = true;
138     }
139 }
140
141 var alphasgradient1 = (Math.tan(Math.atan(-1/(points[p2][3]))+Math.atan(alpha)));
142 var alpha2gradient1 = (Math.tan(Math.atan(-1/points[p2][3])-Math.atan(alpha)));
143 console.log(points[p2][1],"-",points[p1][1],"/(",points[p2][0],"-",points[p1][0],
    gradientbetween, alphasgradient1, alpha2gradient1);
144 if(Math.sign(alphasgradient1)==Math.sign(gradientbetween)){
145     if(Math.abs(gradientbetween)>Math.abs(alphasgradient1)){
146         console.log("conditon 2 satisfied");
147         cond2 = true;
148     }
149 }
150 if(Math.sign(alpha2gradient1)==Math.sign(gradientbetween)){
151     if(Math.abs(gradientbetween)>Math.abs(alpha2gradient1)){
152         console.log("conditon 2 satisfied");
153         cond2 = true;
154     }
155 }
156
157 if(cond1&cond2){
158     var linebetween = two.makeLine(points[apoint][0],points[apoint][1],points[bpoint][0],points[
        bpoint][1]);
159     two.update();
160 }
161
162 }
163 }
164
165 function TestPrimalSimplex() {
166     var test = new Object();
167     test.A = [[-1, 0, 0, 0],
168         [0, -1, 0, 0],
169         [0, 0, -1, 0],
170         [0, 0, 0, -1]];
171     test.b = [-1,-1,-1,-1];
172     test.c = [[0,0,-1,2], [-1,0,1,0], [0,-1,0,1]];
173     test.m = 4;
174     test.n = 3;
175     test.xLB = [2, 0, 0, 0];
176     test.xUB = [3, Infinity, Infinity, Infinity];
177     SimplexJS.PrimalSimplex(test);
178     console.log(test.x, test.z);
179     // Should be 3, 34, 0, 6
180 }
181
182 function TestPrimalSimplex() {
183     var test = new Object();
184     test.A = [[-1, 0, 0, 0],
185         [0, -1, 0, 0],
186         [0, 0, -1, 0],
187         [0, 0, 0, -1]];
188     test.b = [-1,-1,-1,-1];
189     test.c = [[0,0,-1,2], [-1,0,1,0], [0,-1,0,1]];
190     test.m = 4;
191     test.n = 3;
192     test.xLB = [2, 0, 0, 0];
193     test.xUB = [3, Infinity, Infinity, Infinity];
194     SimplexJS.PrimalSimplex(test);
195     console.log(test.x, test.z);
196     // Should be 3, 34, 0, 6
197 }
198
199 TestPrimalSimplex();
200
201
202 two.update();

```

---

The next file is 3contactpoint.js

```
1
2
3 function find3rdpoint(p1,p2){
4     //the first 2 compatible points are p1 and p2
5     for(var i = 0;i<points.length;i++){
6         if(i==p1||i==p2)
7             continue;
8         if(selectedtwopoints(p1,i)&&selectedtwopoints(p2,i)){
9             var linebetween = two.makeLine(points[p1][0],points[p1][1],points[i][0],points[i][1]);
10            var linebetween2 = two.makeLine(points[p2][0],points[p2][1],points[i][0],points[i][1]);
11            two.update();
12        }
13    }
14 }
```

---