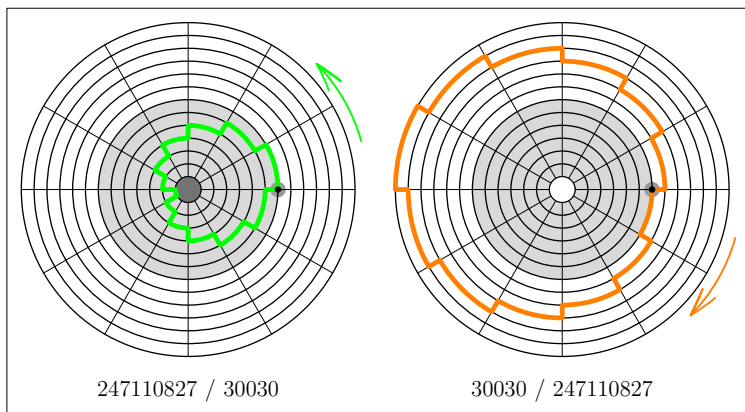


ORBITAL

PETER H. N. LUSCHNY



## CONTENTS

---

1	Synopsis of <code>Orbital.asy</code>	1
2	Synopsis of <code>OrbitalGenerator.asy</code>	3
3	Examples (from <code>OrbitalUsage.asy</code> )	4
3.1	The orbital and its invers.	4
3.2	Using color and other options.	4
3.3	Using the option <i>seperate</i> .	5
3.4	Using the option <i>quad</i> .	5
3.5	The second form of <i>draw</i> .	6
3.6	A colorful lattice.	7
3.7	The usage of generators.	8
3.8	Using the visit hook.	9
3.9	Prime ordered orbitals.	9
3.10	... and now our highlight!	11
4	Source <code>Orbital.asy</code>	14
5	Source <code>OrbitalGenerator.asy</code>	15

```
// -----  
// -- An Asymptote package for drawing orbitals.  
// -- Free under any GNU public license.  
// -- (C) Copyright: Peter Luschny, 2008  
// -- Version 1 Release 1  
// -----
```

## SYNOPSIS OF ORBITAL.ASY

```
-----
--
-- An orbital is here an object of combinatorics, not of
-- celestial mechanics. The set of all orbitals of given
-- length is a lattice.
--
-----
```

The struct Orbital provides 10 application functions.

Two graphical function:

- \* draw(picture dest=currentpicture, int[] jumps)
  - Draws an orbital and/or its transformed side by side
  - or in superposition according to the options set.
- \* draw(picture dest=currentpicture, int[] j1, int[] j2)
  - Draws two orbitals as given in one box.
  - It is required that length(j1) = length(j2).

Jumps may have the values -1 or 0 or 1, at most one occurrence of 0 is allowed and the sum over all jumps must be 0.

Eight functions to set the options are provided:

- \* settransform(string option)
  - Options: (Default is "invers".)
  - "revers": reversing the orbital.
  - "invers": inverting the orbital.
  - "dual": first reversing, then inverting.
  - "identity": draw as given
  - The second draw routine always uses "identity".
- \* setdisplay(string separate)
  - Options: (Default is "inone".)
  - "inone": all orbitals in one picture.
  - "separate": two pictures in row.
  - "quad": four pictures in a square.
  - The second draw routine always uses "inone".
- \* setplot(string option)
  - Options: (Default is "orbandtrans".)
  - "oronly": plot only the given orbital.
  - "transonly": plot only the transformed orbital.
  - "orbandtrans": plot the orbital and its trans.
  - The second draw routine ignores this option.

```

* setlabel(string option)
Options:      (Default is "none".)
-- "none":    write no label.
-- "symbolic": write a symbolic representation.
-- "numeric": write the primorial of the orbital.
-- "info":    both "symbolic" and "numeric".

* sethome(string option)
Options:      (Default is "bighome".)
-- "nohome":  Set no marker at the homeposition.
-- "tinyhome": Set tiny marker at the homeposition.
-- "bighome": Set big marker at the homeposition.

* setarrow(bool option)
Options:      (Default is 'false'.)
-- 'true':    Show arrow indicating the orientation.
-- 'false':   Show no arrow.

* setorbpen(pen orbcolor, pen transcolor)
The color of the orbitals and the transformed
orbital. Default is 'green' and 'orange'.

* setfillpen(pen innercolor, pen outercolor)
The color of the inner and the outer part of the disk.
Default colors are 'gray(0.85)' and 'white'.

```

How to interpret the output:

All orbitals start in the same position, the 'home position', and return to it (an orbital path is closed.)

Note the following conventions:

The given orbital runs in the counterclockwise sense (is mathematically positive oriented) and has by default the color green. Any transformed orbital runs in the clockwise sense (is mathematically negative oriented) and has the default color orange.

In the case that both the orbital and the transformed orbital are displayed in the same picture, the label and other visual hints always refer to the original orbital.

---

*Even the wisest and most prudent people often suffer from what logicians call insufficient enumeration of cases.* JAKOB BERNOULLI

The `OrbitalGenerator` generates all orbitals of given length  $n$ .  $n$  must be smaller than 13, because the number of orbitals gets huge when  $n$  gets large. An `OrbitalGenerator` is constructed by a call to

```
OrbitalGenerator OrbitalGenerator(int len, VISIT hook = null)
```

Say you constructed a `OrbitalGenerator` with length 6

```
OrbitalGenerator og = OrbitalGenerator(6);
```

then you can use it by the call `og.generate()`. All orbitals will be generated one by one as a list out of  $\{-1, 0, 1\}$ . These encoded orbitals (called 'jumps') can be accessed via a `VISIT` function which has to be given to the constructor. Now every time the generator has finished a list of jumps this `VISIT` function is called and the list is handed over for processing. A `VISIT` function must have the type

```
typedef void VISIT(int[]); .
```

One thing you can do during this *visit* is to put the generated orbitals in a different order. In fact this is what a second constructor does:

```
PrimeOrbitalGenerator PrimeOrbitalGenerator(int len).
```

Using this constructor will give you a list of orbitals of length  $len$  which are sorted in *primorial order* when you call the function

```
PrimeOrbital[] generate() .
```

For example

```
PrimeOrbitalGenerator pog = PrimeOrbitalGenerator(6);
```

```
PrimeOrbital[] po = pog.generate();
```

will return a list of orbitals of length 6 sorted in primorial order. A `PrimeOrbital` is the third and last structure provided by this module. It contains the member `int[] jumps` which you can hand over to the `draw` member of the orbital structure for visual display, for example in a call similar to

```
orbital.draw(picture, primeorbital.jumps); .
```

# 3

## EXAMPLES (FROM ORBITALUSAGE.ASY)

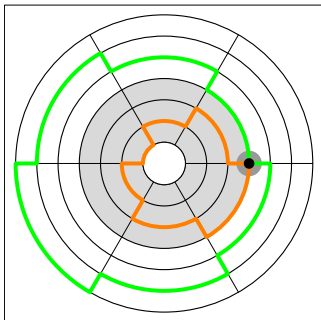
---

```
import orbital;  
import orbitalgenerator;
```

### 3.1 THE ORBITAL AND ITS INVERS.

The most simple case. The orbital and its invers in one picture.

```
void demo1()  
{  
    int[] jump = {1,1,1,-1,-1,-1};  
  
    Orbital orb = Orbital(200); // -- size 200  
    orb.draw(jump);  
  
    shipout("orbdemo1", bbox(0.2cm));  
}
```



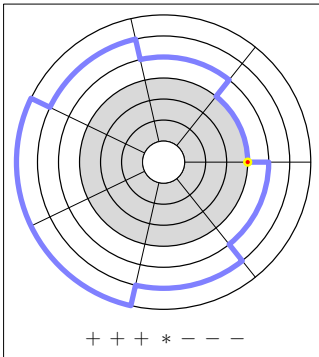
### 3.2 USING COLOR AND OTHER OPTIONS.

```
void demo2()  
{  
    int[] jump = {1,1,1,0,-1,-1,-1};  
  
    Orbital orb = Orbital(200); // -- size 200  
  
    // -- Path option  
    orb.setplot("orbonly");  
  
    // -- Label option  
    orb.setlabel("symbolic");  
  
    // -- Color options  
    pen orbpen = linewidth(6*linewidth()+lightblue;  
    orb.setorbpen(orbpen, orbpen);  
    orb.sethome("tinyhome");
```

```

orb.draw(jump);
shipout("orbdemo2", bbox(0.2cm));
}

```



### 3.3 USING THE OPTION *seperate*.

The orbital and its invers in two pictures.

```

void demo3()
{
    int[] jump = {-1,-1,-1,-1,-1,-1,1,1,1,1,1,1};

    Orbital orb = Orbital(200);

    orb.setlabel("numeric");
    orb.setdisplay("seperate");
    orb.setarrow(true);
    orb.draw(jump);

    shipout("orbdemo3", bbox(0.25cm));
}

```

The picture is displayed on the front page of this document.

### 3.4 USING THE OPTION *quad*.

The orbital and all its transformed forms in 4 pictures.

```

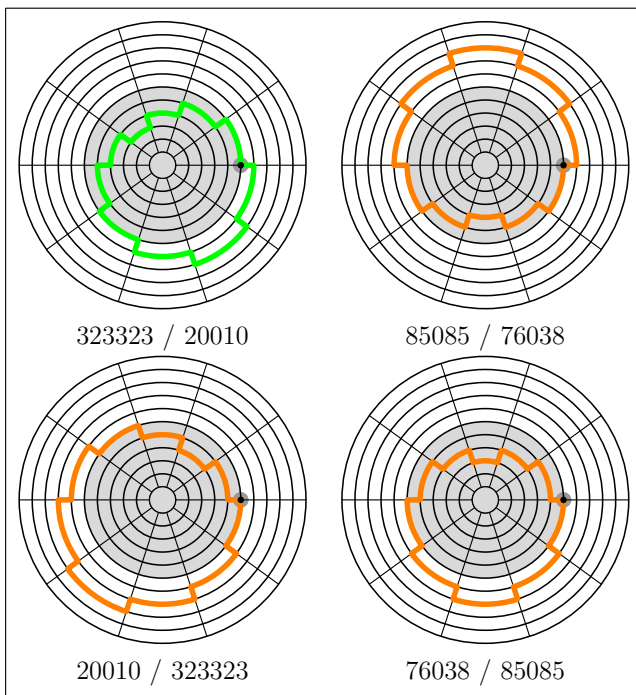
void demo4()
{
    int[] jump = {-1,-1,-1,1,1,1,1,1,-1,-1};

    Orbital orb = Orbital(150);

    orb.setlabel("numeric");
    orb.setdisplay("quad");
    orb.draw(jump);

    shipout("orbdemo4", bbox(0.2cm));
}

```



### 3.5 THE SECOND FORM OF *draw*.

Two orbitals as given in one picture.

```
void demo6()
{
    int[][]orbits = {
        {1, 1, -1, -1}, {1, -1, 1, -1},
        {-1, 1, -1, 1}, {-1,-1, 1, 1} };

    int mag = 100; real margin = 5mm;
    Orbital orb = Orbital(mag);

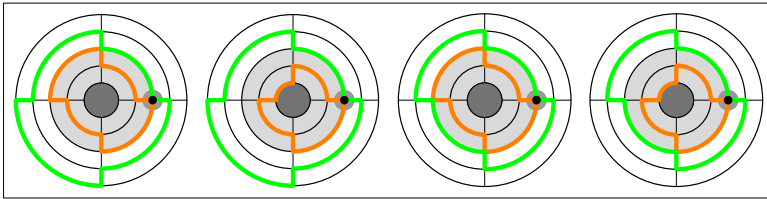
    picture pic1; orb.draw(pic1, orbits[0], orbits[2]);
    picture pic2; orb.draw(pic2, orbits[0], orbits[3]);
    picture pic3; orb.draw(pic3, orbits[1], orbits[2]);
    picture pic4; orb.draw(pic4, orbits[1], orbits[3]);

    // -- add to current picture
    size(currentpicture, 4*mag + 3*margin, mag);

    add(currentpicture, pic1.fit(),(0,0));
    add(currentpicture, pic2.fit(),(240+margin,0));
    add(currentpicture, pic3.fit(),(480+2*margin,0));
    add(currentpicture, pic4.fit(),(720+3*margin,0));

    shipout("orb6demo", bbox(0.2cm));
}

```



### 3.6 A COLORFUL LATTICE.

```

void demo7()
{
    size(520);
    int[][]orbits = {
    { 1,1,-1,-1}, { 1,-1, 1,-1}, { 1,-1,-1,1},
    {-1,1, 1,-1}, {-1, 1,-1, 1}, {-1,-1, 1,1} } ;

    pair[] pos = {(3,1),(3,5/2),(1,4),(5,4),(3,11/2),(3,7)};

    void zeige(int i, int k)
    {
        draw(pos[i]--pos[k],black+linewidth(3.6*linewidth()));
    }

    zeige(1,0); zeige(2,1); zeige(3,1);
    zeige(4,2); zeige(4,3); zeige(5,4);

    Orbital orb = Orbital(1);
    orb.setplot("orbonly");

    orb.setFillpen(gray(0.85),cmk(0.15,0.0,0.69,0.0));
    pen p = cmk(0.0,1.0,0.0,0.0)+linewidth(4*linewidth());
    orb.setorbpen(p,p);

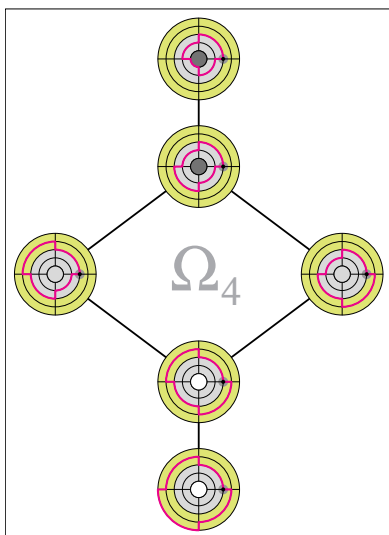
    for(int i = 0; i < 6; ++i)
    {
        pair Z = pos[i];
        picture pic;
        orb.draw(pic, orbits[i]);
        add(scale(8)*pic, Z);
    }

    // Needs: import fontsize; but did not work.
    // pen p = Symbol(series="m",shape="n");
    // p = p+fontsize(56)+gray(0.85);
    // label("\Omega_4$", (3,4),p);
    // Workaround with postscript:

    postscript("/inch{ 72 mul 30 sub } def");
    postscript("0.0 0.0 0.0 0.4 setcmykcolor");
    postscript("/Symbol findfont 64 scalefont setfont");
    postscript("2.95 inch 4.1 inch moveto (\127) show");
    postscript("/Symbol findfont 40 scalefont setfont");
    postscript("3.65 inch 3.9 inch moveto (4) show");

    shipout("orb7demo", bbox(0.3cm));
}

```



### 3.7 THE USAGE OF GENERATORS.

```
void demo8()
{
    OrbitalGenerator og = OrbitalGenerator(6);
    int n = og.generate();
    write("Orbitals"+"(string)6+") = "+(string)n);
}
```

The output:

```
1,1,1,-1,-1,-1
1,1,-1,1,-1,-1
1,1,-1,-1,1,-1
1,1,-1,-1,-1,1
1,-1,1,1,-1,-1
1,-1,1,-1,1,-1
1,-1,1,-1,-1,1
1,-1,-1,1,1,-1
1,-1,-1,1,-1,1
1,-1,-1,-1,1,1
-1,1,1,1,-1,-1
-1,1,1,-1,1,-1
-1,1,1,-1,-1,1
-1,1,-1,1,1,-1
-1,1,-1,1,-1,1
-1,-1,1,1,1,-1
-1,-1,1,1,-1,1
-1,-1,1,-1,1,1
-1,-1,-1,1,1,1
```

Orbitals(6) = 20

### 3.8 USING THE VISIT HOOK.

```
void myVisit(int[] orb)
{
    string s;
    for(int o : orb) { s = s + (string)o + " "; }
    write(reverse(s) + " ...the way I like it...");
}

void demo10()
{
    int len = 4, count;
    write("Generate all orbitals of length "+(string)len+" !");

    // -- without visit
    OrbitalGenerator og = OrbitalGenerator(len);
    count = og.generate();
    write("Orbitals("+string)len+" = "+(string)count);
    write("And now processing the orbitals my way!");

    // -- with visit
    OrbitalGenerator myog = OrbitalGenerator(len, myVisit);
    count = myog.generate();
    write("Orbitals("+string)len+" = "+(string)count);
}
```

Generate all orbitals of length 4 !

```
1,1,-1,-1,
1,-1,1,-1,
1,-1,-1,1,
-1,1,1,-1,
-1,1,-1,1,
-1,-1,1,1,
```

Orbitals(4) = 6

And now processing the orbitals my way!

```
1- 1- 1 1 ...the way I like it...
1- 1 1- 1 ...the way I like it...
1 1- 1- 1 ...the way I like it...
1- 1 1 1- ...the way I like it...
1 1- 1 1- ...the way I like it...
1 1 1- 1- ...the way I like it...
```

Orbitals(4) = 6 .. Ups!

### 3.9 PRIME ORDERED ORBITALS.

```
void demo9()
{
    for(int i = 1; i < 6; ++i)
    {
        write(" Omega("+string)i +" ) prime ordered.");

        PrimeOrbitalGenerator pog = PrimeOrbitalGenerator(i);
        pog.generate();
        pog.write();
    }
}
```

$\Omega_1$  prime ordered  
 0            1/1            1.00

$\Omega_2$  prime ordered  
 1   -1        2/3        0.66  
 -1   1        3/2        1.50

$\Omega_3$  prime ordered  
 1   0   1        2/5        0.40  
 0   1   -1        3/5        0.60  
 1   -1   0        2/3        0.66  
 -1   1   0        3/2        1.50  
 0   -1   1        5/3        1.67  
 -1   0   1        5/2        2.50

$\Omega_4$  prime ordered  
 1   1   -1   -1        6/35        0.17  
 1   -1   1   -1        10/21        0.47  
 1   -1   -1   1        14/15        0.93  
 -1   1   1   -1        15/14        1.07  
 -1   1   -1   1        21/10        2.10  
 -1   -1   1   1        35/6        5.83

$\Omega_5$  prime ordered  
 1   1   0   -1   -1        6/77        0.07  
 1   1   -1   0   -1        6/55        0.10  
 1   0   1   -1   -1        10/77        0.13  
 1   1   -1   -1   0        6/35        0.17  
 0   1   1   -1   -1        15/77        0.19  
 1   0   -1   1   -1        14/55        0.25  
 1   -1   1   0   -1        10/33        0.30  
 0   1   -1   1   -1        21/55        0.38  
 1   -1   0   1   -1        14/33        0.42  
 1   -1   1   -1   0        10/21        0.47  
 1   0   -1   -1   1        22/35        0.62  
 -1   1   1   0   -1        15/22        0.68  
 1   -1   -1   1   0        14/15        0.93  
 0   1   -1   -1   1        33/35        0.94  
 -1   1   0   1   -1        21/22        0.95  
 1   -1   0   -1   1        22/21        1.05  
 0   -1   1   1   -1        35/33        1.06  
 -1   1   1   -1   0        15/14        1.07  
 1   -1   -1   0   1        22/15        1.47  
 -1   0   1   1   -1        35/22        1.59  
 -1   1   -1   1   0        21/10        2.10  
 -1   1   0   -1   1        33/14        2.36  
 0   -1   1   -1   1        55/21        2.62  
 -1   1   -1   0   1        33/10        3.30  
 -1   0   1   -1   1        55/14        3.93  
 0   -1   -1   1   1        77/15        5.13  
 -1   -1   1   1   0        35/6        5.83  
 -1   0   -1   1   1        77/10        7.70  
 -1   -1   1   0   1        55/6        9.17  
 -1   -1   0   1   1        77/6        12.80

### 3.10 ... AND NOW OUR HIGHLIGHT!

Display all orbitals (and their inverted form) of length 6 in primorial order!

```
void demo11()
{
    PrimeOrbitalGenerator pog = PrimeOrbitalGenerator(6);
    PrimeOrbital[] po = pog.generate();
    pog.write();

    Orbital orb = Orbital(1);
    orb.setlabel("info");
    int i = -1;

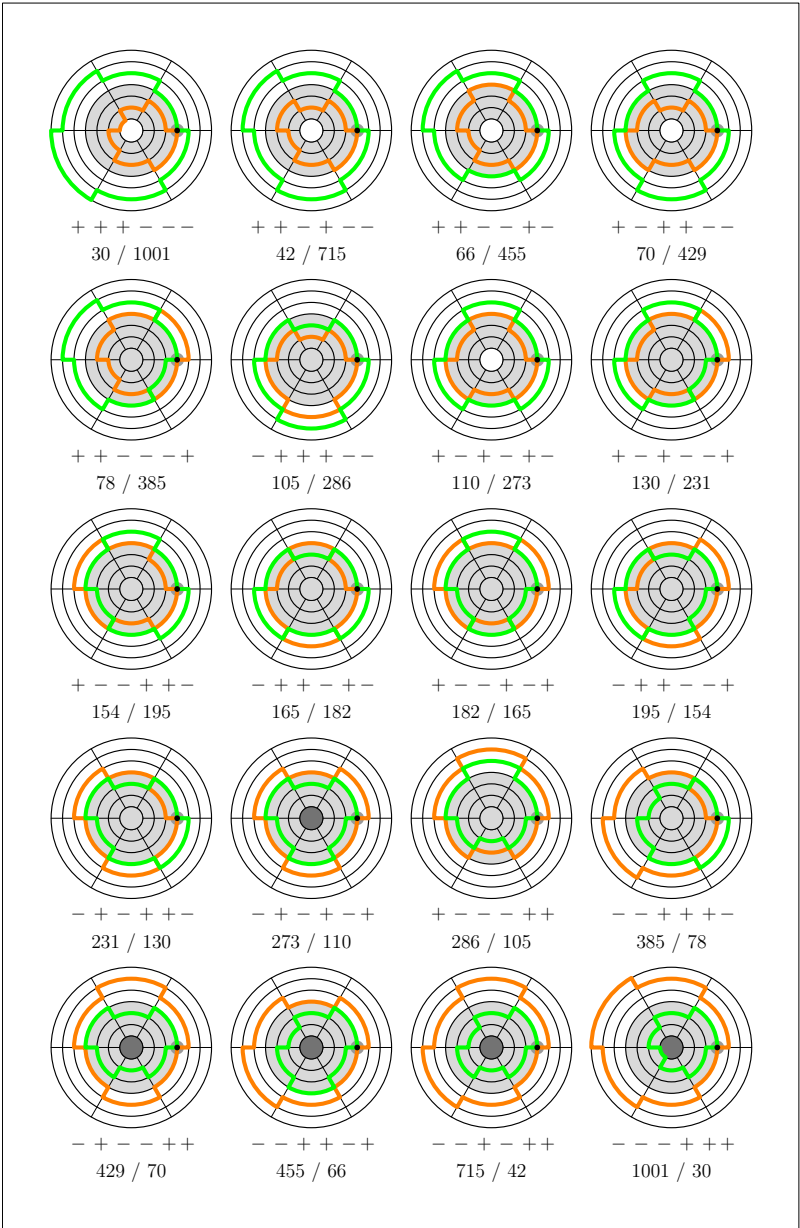
    for(int y = 690; y > 100; y -= 140)
    {
        for(int x = 135; x < 500; x += 110)
        {
            pair Z = (x, y+18);
            picture pic;
            orb.draw(pic, po[++i].jumps);
            add(scale(7)*pic, Z);
        }
    }
    shipout("orb11demo", bbox(1cm));
}
```

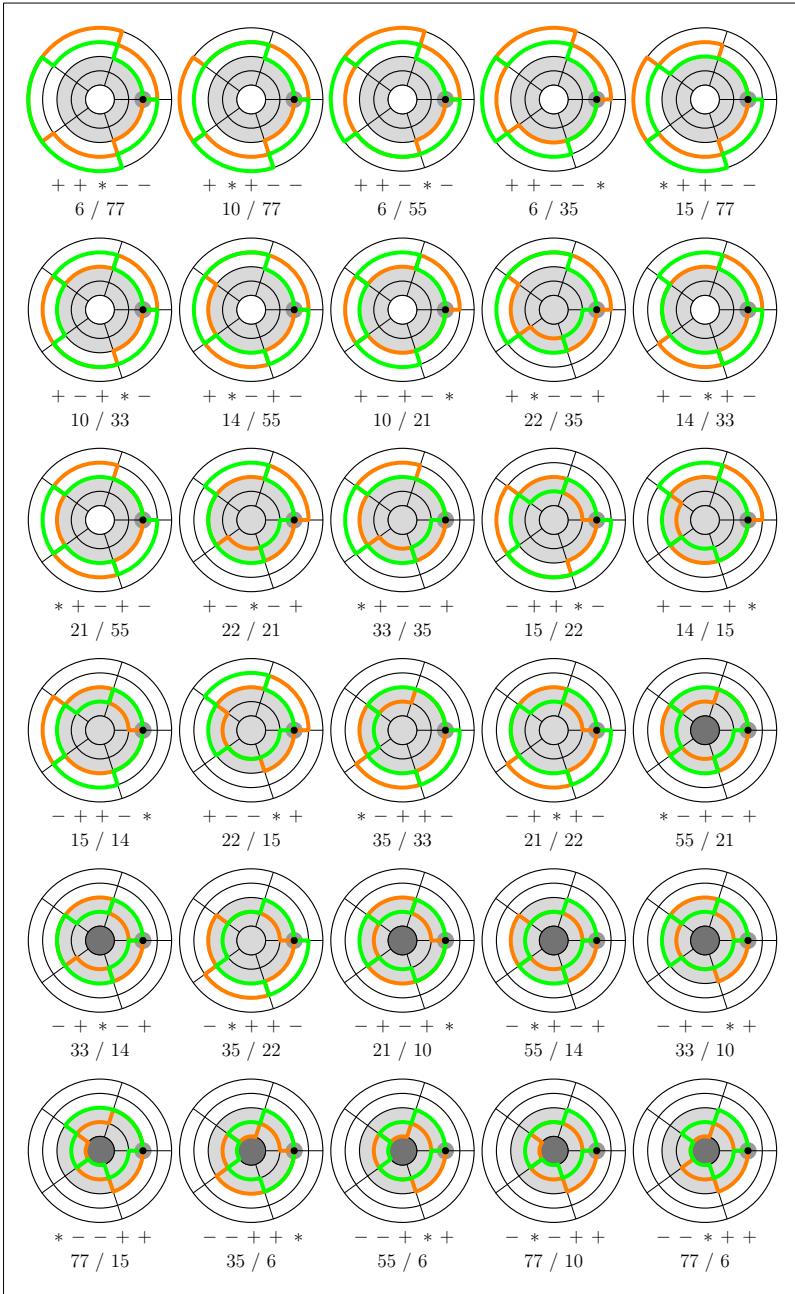
Display all orbitals (and their dual form) of length 5 in primorial order!

```
void demo12()
{
    PrimeOrbitalGenerator pog = PrimeOrbitalGenerator(5);
    PrimeOrbital[] po = pog.generate();
    pog.write();

    Orbital orb = Orbital(1);
    orb.settransform("dual");
    orb.setlabel("info");
    int i = -1;

    for(int y = 845; y > 60; y -= 132)
    {
        for(int x = 135; x < 550; x += 95)
        {
            pair Z = (x,y + 18);
            picture pic;
            orb.draw(pic, po[++i].jumps);
            add(scale(9)*pic, Z);
        }
    }
    shipout("orb12demo", bbox(0.5cm));
}
```





# 4

## SOURCE ORBITAL.ASY

---

The prototype of the nonprivate interface of *Orbital*.

```
struct Orbital {  
    Orbital Orbital(int mag);  
  
    void settransform(string option);  
    void setlabel(string option);  
    void setplot(string option);  
    void setdisplay(string option);  
    void sethome(string option);  
    void setorbpen(pen orbcolor, pen transcolor);  
    void setfillpen(pen innercolor, pen outercolor);  
    void setarrow(bool option);  
  
    void draw(picture dest=currentpicture, int[] jump);  
    void draw(picture dest=currentpicture, int[] j1, int[] j2);  
}
```

SOURCE ORBITALGENERATOR.ASY

---

The prototype of the nonprivate interface of *OrbitalGenerator*, *PrimeOrbital* and *PrimeOrbitalGenerator*.

```
typedef void VISIT(int[]);

struct OrbitalGenerator
{
    OrbitalGenerator OrbitalGenerator(int len, VISIT hook = null);
    int generate();
}

struct PrimeOrbital
{
    restricted int[] jumps;
    restricted int numer, denom;
    restricted real balance;

    PrimeOrbital PrimeOrbital(int[] jumps);
    void write();
}

struct PrimeOrbitalGenerator
{
    PrimeOrbitalGenerator PrimeOrbitalGenerator(int len);
    PrimeOrbital[] generate();
    void write();
}
```

---

ORBITAL – *An Asymptote package for drawing orbitals.*

© Peter H. N. Luschny

Web Edition. Printing winter 2008.

This temporary version will expire on December 31th, 2008.

For updates see: <http://www.luschny.de/math/swing/orbital>

---