Extending MathMorphs with Function Plotting

by Andrés Valloud

Buenos Aires University

Introduction

This chapter describes how to plot mathematical functions in Squeak. It covers and shows the objects involved and how to present the results in Morphic using the MorphicWrappers. It is aimed at Squeakers who desire to develop objects with rich graphic representations.

History notes

The history of the function plotters is interesting. They started with the MathMorphs project at UBA. The MathMorphs project is led by Leandro Caniglia, Ph.D. in Mathematics, and has the goal of introducing mathematical objects in the computer together with their definitions. The result is that mathematical objects come alive on the screen, and are much more than a pile of coefficients thrown somewhere in a more or less arbitrary way. The project has grown and expanded itself into areas other than mathematics and computer science, including physics and biology.

Within the focus of MathMorphs, a group of students, including myself, attended the course "Objetos Matemáticos en Smalltalk" (Mathematical Objects in Smalltalk) at UBA. Among other ideas we studied the Sturm theorem, which regards the isolation of real roots of polynomials. By means of an implementation of this theorem, we were able to represent algebraic numbers in a computer with infinite precision. Algebraic numbers are a field composed by the roots of polynomials with integer coefficients. This set of numbers includes the integers, rationals, and their n-th roots. The Sturm theorem was interesting by itself, and I decided to make a plotter to see how the theorem reacted to different polynomials. The theorem associates each polynomial with a chain of polynomials which, when evaluated, tells us where the roots of the original polynomial are. The plotter was a plotter for the chain.

After the Sturm plotter was completed, I constructed other simple plotters, especially for Munsell's HSV color system. After that, there was the need for a general function plotter. A compression project I was working on needed the ability to plot histograms. The Sturm plotter was quite simple. It just built the image and pasted it on the screen, using a big, complicated and inelegant main loop. The Munsell plotters were a bit more advanced in the way the plot was drawn, but their architecture was pretty much the one found in the Sturm plotter. This approach lacked enough generality to provide a framework on its own. Therefore, it was necessary to create it first.

The development of the function plotters

In order to build a graph we need a grid to plot it on, the functions we will plot, and some range in which we will evaluate the functions. We also need a procedure to draw our plot on the grid. We will solve each of these problems one by one. We will also leave room for a control entity to come forward, the plotter itself.

An introduction to grid plotting

Our first problem is the grid. If we were to plot on a grid, what color scheme would we like? A grid resembling a blackboard or a notebook's sheet of paper could be interesting. Almost all plots have thicker lines for the axes and thinner lines for the grid, if any grid is present.

Form and ColorForm

Given we are requested a grid of a certain image size, we need a piece of paper of that size in order to draw the grid on it. In Squeak, and in other Smalltalks too, the objects that represent such pieces of paper are instances of **Form**. These objects can be created in many ways, but for most purposes this will be enough:

```
Form extent: aPoint depth: anInteger
```

As a result we obtain a **form** that has a size of **aPoint**, and that uses **anInteger** bits at each pixel for color determination purposes. A **640@480** size form with a depth of **16** is a high color form consisting of **640** horizontal pixels by **480** vertical pixels.

Note that Squeak provides both **Form** and **ColorForm** pieces of paper. The first uses a given amount of bits per pixel, its depth, to determine the pixel's color immediately using the same amount of bits for each color component in the RGB color space. Frequent amounts of bits per color component are 3, 4, 5 and 8. These bits give colors in a fixed color space of 512 (9 bits deep), 4,096 (12 bits deep), 32,768 (15 bits deep) and 16,777,216 colors (24 bits deep), respectively. Note that 16 bit deep forms use 15 bits per pixel to store colors. The extra bit is currently unused, and is there to pad the 15 bits into 16 to make handling easier.

On the other hand, instances of **ColorForm** hold a color pallete and the values at each pixel position are indices to the palette. The maximum size of the palette is 256 colors, hence instances of **ColorForm** can use up to 8 bits per pixel. The advantage is that if we need an image in true color that uses 256 colors or less, we can get an exact copy in **aColorForm** and thus reduce each pixel entry size from 24 to 8 bits. As a result, the form's memory requirement is divided by approximately 3.

Color and TranslucentColor

Instances of **Color** hold 10 bits per RGB channel. This avoids propagation of round-off errors when adding, subtracting or mixing colors together. In addition, instances of **TranslucentColor** hold 8 additional bits describing its translucency coefficient. When this coefficient is zero, the color is transparent. If it is maximum, then the color is opaque. The **Color** and **TranslucentColor** interfaces use floating point numbers instead of bit chunks. All the value ranges are normalized to **1.0**, therefore, the RGB and alpha values can be anything from **0.0** to **1.0**.

Colors can be added and subtracted together, which is component-wise addition and subtraction. They can also be component-wise multiplied by **aNumber**. The result of each individual component after these operations is checked for bounds, and forced into **[0.0, 1.0]**. For instance,

```
Color gray - Color white = Color black
```

evaluates to **true**. When painting an area with **aTranslucentColor** that has an alpha value of (for example) **0.7**, the resulting color will be:

```
(theBackgroundColor * 0.3) + (aTranslucentColor * 0.7)
```

The component overflow check is not necessary for this expression. It also shows that regular colors are translucent colors with a translucency index of **1.0**. Colors in Squeak also support the HSV color system, also known as the Munsell color system. The initials HSV stand for **hue**, **saturation** and **value**. It is possible to ask **aColor** about these values, and also about its **luminance** and **brightness.**

FormCanvas

When we draw on a piece of paper, a drawing board mimicked by instances of **FormCanvas** can be useful. They are created in the same way as a **Form**, and they hold **aForm** inside them. This form can be requested by sending the message **form**.

Although it is possible to do things with instances of **Form** alone, instances of **FormCanvas** provide a more suitable interface for our purposes. Several geometrical shapes can be drawn very easily using **aFormCanvas**, including lines, circles, rectangles and ellipses. This separation between the actual graphic and the algorithms that draw things on this data is important. It allows us to change and enhance our algorithms without enlarging the base data class. It also allows very complicated procedures to be encapsulated in their own class without spreading instance variables. Moreover, it encourages polymorphism since then the same algorithm can be run in different objects without needing to rewrite code.

Let's go back to our problem. We could draw the grid by first filling **aFormCanvas** with a single background color. Then, we could draw lines on top of the background color. In this regard, the protocol of **aFormCanvas** includes:

line: startPoint to: endPoint width: anInteger color: aColor

fillColor: aColor

fillRectangle: aRectangle color: aColor

The first message instructs **aFormCanvas** to draw a line of color **aColor**, from **startPoint** to **endPoint**. Each dot drawn will be a square of **anInteger** by **anInteger** pixels. The second message tells the **formCanvas** to fill itself with **aColor**.

The third message fills **aRectangle** inside the **formCanvas** with **aColor**. This deserves particular attention. First, to build an instance of **Rectangle** we can send either the **corner: aPoint** or the **extent: aPoint** message to another point. The message **corner: b** sent to a point named **a** creates **aRectangle** whose corners are **a** and **b**. On the other hand, the message **extent: b** creates **aRectangle** whose corners are **a** and **a+b**.

Filling rectangles in aFormCanvas

Back to **FormCanvas** and **fillRectangle:**. The fact that a rectangle is filled with **aColor** means that to paint a single horizontal line at the y position **yStart**, we need to use:

aFormCanvas fillRectangle:

(xStart @ yStart extent: xEnd @ yStart + 1)

If we did not add **+ 1** at the end, the rectangle would have an area of zero, and when filled with paint this would indicate we need no paint, and then nothing would be painted at all. This means that if we have **aFormCanvas** with a size of **320@200**, and we wanted to paint its last horizontal line, we would have to fill the rectangle resulting from:

0@198 extent: 320@1

Nothing would be painted if the rectangle started at **0@199**, because then the rectangle would fall outside **aFormCanvas**. However, we could also try this rectangle with the desired effect:

0@199 corner: 320@198

Ok, now, to draw a grid with thicker x and y axes, we need to know where the axes are. But that information is available only after we evaluate all the functions in their given domains. So, before we build the grid, we need to evaluate the functions.

Function evaluation

The function evaluation goals are to provide a bound for the functions' images, and to calculate the plot's actual points. This problem is different for every plotter. It is not the same to evaluate a function in cartesian as in polar coordinates. Let's take a look at the cartesian case first.

Evaluation in cartesian coordinates

Function evaluation is a two-step process. First, the functions are evaluated in their own domain to get their image bound. Then, the calculated points are scaled into points that can be drawn directly over our grid, once it is drawn using the information collected from the first step above.

For our purposes, we will not do the job in this order. We will first process all the function points, and then we will draw the grid. To avoid unnecessary complications, we will do all function evaluation and manipulation inside instances of the kind of **FunctionPlotterFunction**.

Plotter function protocol

To create a plotter function, we will use **aPlotterFunctionClass new: aFunction**. Here, **aFunction** is essentially any object that understands the message **valueAt:**. Although disguised blocks can serve as functions, objects from a mathematical function hierarchy should be used instead. The basic protocol of plotter functions is:

domain

domain: aRegion

evaluate: anInteger

evaluate: anInteger timesIn: aRegion

imageBound

invalidatePointCache

scaleTo: anAmbient

scaled

The **domain** accessors provide access to the function's domain. The **imageBound** message requests the **imageBound** for the function. If the function has not been evaluated yet, the answer is **nil**. To evaluate the function, the **evaluate:** messages are used. Here, **anInteger** is the amount of samples to take within the domain. If **evaluate:timesIn:** is sent, **aRegion** becomes the domain and then the function is evaluated **anInteger** times. The **scaleTo: anAmbient** message tells the function to take the values given by the evaluation process and translate them to

actual plotting points over the grid, according to **anAmbient**. In practice, **anAmbient** will be the grid plotter we still have to describe. After point scaling, the answer to the **scaled** message will be **true**. Finally, plotter functions will hold their scaled points until told to invalidate their point cache.

In the case of cartesian function plotting, the concrete subclass of **FunctionPlotterFunction** used will be **XYPlotterFunction**. The instance variables of these objects include the **domain**, the **function**, the **yBound**, and the **valueCache**.

Regions

Domains, bounds and intervals in general will be represented by instances of **ClosedInterval**. They are created by evaluating

```
ClosedInterval from: anObject to: anotherObject
```

Their ends are accessed by the **start** and **stop** messages. They also implement the **size** message, which is implemented by answering **stop – start**. The protocol for instances of **ClosedInterval** includes mutator messages which are very useful for progressive enclosures. For instance, in order to find the **ClosedInterval** that best encloses a set of intervals, we can take the copy of any of them and then send a **do:** to the set.

```
answer ← (aSet detect: [:each | true]) copy.

aSet do: [:some | answer growSoThatEncloses: some]
```

In the first line, **answer** becomes the copy of any interval in the set of intervals. In the second line, it is told to grow so that it encloses every other interval. We will do this to find the image and domain bounds. The numerical version of this message is **growSoThatIncludes: aNumber**. Now, we will go into the evaluation details.

First step of cartesian evaluation

We will assume that we request a plot that has an extent of **640@480**, and the functions we want to plot have to take values in the range **[a, b]**. Each function may have its own particular domain, with the restriction that the closed interval **[a, b]** encloses all particular domains exactly. Our strategy will be to take one sample per horizontal pixel requested within the range that encloses all function domains. In this case, we could then sample the interval **[a, b]** 640 times. This can be tricky, because if we take steps of **b – a / 640**, the final sample will be at **a - b / 640 + b**, which is not **b**! We could then start taking samples at **a**, incrementing the probing value by **b – a / 639.** This causes problems when the plot size has a width of 1 pixel, because then we will divide by zero. Hence, we will use **b – a / 640** steps, but evaluate 641 times instead. This produces a harmless extra

point, and it also ensures we will have at least two points to plot, which will be useful later.

This process also allows us to determine a bound for the image of the function. Moreover, it accommodates for each function to have its own domain. Once we have the functions' image bound, we can determine where the x and the y axes are. Here is the method **evaluate:**

```
evaluate: anInteger
| deltaX currentX currentY |
deltaX ← domain size / anInteger.
valueCache ← (OrderedCollection new: anInteger + 1).
"We start with an 'empty' interval"
yBound ← ClosedInterval
        from: (function valueAt: domain start)
        to: (function valueAt: domain start).
0 to: anInteger do:
[:each |
    currentX ← deltaX * each + domain start.
    "We enlarge the interval so that it includes every point"
    yBound growSoThatIncludes:
        (currentY ← function valueAt: currentX).
    valueCache add: currentX @ currentY]
```

Special care is taken in the **currentX** assignment to avoid floating point addition problems. This happens when **deltaX** is not large enough on its own to make **domain start** change, or when addition results in an error of increasing size in **currentX**.

Second step of cartesian evaluation

There is a second step in function evaluation process. Once the image bounds are determined, we know what our plot will represent. Namely, the rectangle of the cartesian plane which has a horizontal span corresponding to the domain bound of all the functions' domains, and a vertical span of the image bound for all the functions' images. The task we now face is to map our evaluation space into the plotting space, an instance of **Form**. In this case, this will be done by the "ambient", the grid plotter. In order to do this, functions provide the message **scaleTo: anAmbient**. Ambients, on the other hand, provide scaling messages. Here are the ones we will use for this stage:

includes: aPoint

pointFor: aPoint

spanFor: aPoint

transformSpanToGraph: aPoint

transformSpanYToGraph: aValue

yForXAxis

The message **includes:** is answered with **true** when the evaluation space of the grid includes **aPoint**. When a grid plotter knows the evaluation space and receives **yForXAxis**, it can answer the y position in the plot where the x axis is. Furthermore, the grid plotters respond to the message **transformSpanYToGraph:** by answering the y position in the plot that corresponds to a y value in the evaluation space.

The message **spanFor:** is answered with a point in our evaluation space that corresponds to **aPoint** in the plot. The message **pointFor: aPoint**, on the other hand, gives as a response a point in the plot that corresponds to **aPoint** in the evaluation space (we will also refer to the evaluation space as the span). A mutator version of the message **pointFor: aPoint** is **transformSpanToGraph: aPoint**, which changes **aPoint** into **pointFor: aPoint** but without creation of new **Point** instances. This can be extremely useful when dealing with a lot of points and functions, because this environment promotes the creation of large amounts of points that will become useless quickly. Regarding this issue, creation of **Point** instances has been a problem in the past. A few simple modifications like the ones described above and below allowed a performance increase of up to 72%.

Nonreferenced (dead) objects must be detected so they do not take up memory, and this is the job of the garbage collector. Garbage collection can be pretty fast in Squeak, but that does not mean we are entitled to load it with tons of work because it is fast anyway. The best way to deal with garbage collection time is to avoid creating garbage in the first place. In addition, if we do not create unnecesary objects, we also avoid the cost of such creation, which is also expensive. We will come back to these issues later.

The CartesianGridPlotter as an ambient

Let's review the **transformSpanToGraph:** implementation. In order to do this, we must take a look at the **CartesianGridPlotter**. This object will provide the grid on which to plot, plus the transformation services between the plotting space and the function evaluation space. For the purposes of the plotter, the evaluation space will be stored as an instance of **Rectangle**, in the instance variable called **span**.

The problem now is to map points in the **span** into points in the **grid**, thus scaling. The **span** will be generated from the domain and image bounds calculated using the method **growSoThatEncloses:**, and fed to the grid plotter when the function evaluation process is completed. Each function will then be sent **scaleTo: aCartesianGridPlotter**. The implementation of this message is shown below:

```
scaleTo: aPlottingGrid

    valueCache do: [:each | aPlottingGrid transformSpanToGraph: each].

    self scaled: true
```

Note that points are mutated inside the **valueCache**, also avoiding the use of **at:** and **at:put:**. This is not recommended as a general rule. However, tricks like this help improve the performance so much, that their carefully and properly controlled utilization can be considered as a valid alternative for the implementation of critical sections. In this particular case, this is done because point scaling can be extremely time consuming because of creation of **Point** instances, and the further management of the created points. Actually, it is interesting to compare how many points are created by this and other more "orthodox" procedures.

Now let's check the calculations necessary to transform **aPoint** in the **span** to a point in the **grid**. We will consider the x coordinate first. In this case, the **span** and **grid extent** have, or will usually have, different **origin x** coordinates. This means that we must shift all numbers before doing the calculations involved in scaling, then proceed with zero based number crunching, and then shift the results as a last step. The first step of this process it to take **aPoint x** and subtract **span origin x** from it. Second, we must translate the distance from **aPoint x** to **span origin x** into an equivalent distance from **aPoint x** to the origin point of the **grid**. This is done by the expression

```
aPoint x - span origin x * graphSize x / span width
```

This expression is correct, but the problem with it is that **graphSize x / span width** never changes. So, the plotter will cache this value in the instance variable named **spanWgSizeX** when the **span** is given. This factor is the ratio between the **grid**'s **width** and the **span**'s **width**.

The vertical scaling is a bit tricky, since in graphs higher values of y mean higher position in the graph, whereas in instances of **Form** higher values of y mean lower position in the graph. We start then with this expression instead:

```
span corner y - aValue * graphSize y / span height
```

Again, **graphSize y / span height** does not change and so the plotter will cache it into **spanHgSizeY**. In order to mutate **aPoint**, we will use the private method **setX: anObject setY: anotherObject**. Again, the

reason behind this is to avoid unnecessary point creation. Here is the point mutator method in **CartesianGridPlotter** that translates from span space to graph space:

```
transformSpanToGraph: aPoint
    aPoint
        setX: (aPoint x - span origin x * spanWgSizeX) rounded
        setY: (span corner y - aPoint y * spanHgSizeY) rounded
```

We round the new values to create integer coordinate points.

So, we have now evaluated functions in cartesian coordinates. But what about polar coordinates?

Evaluation in polar coordinates

Evaluation in polar coordinates is different because the goal is to evaluate a function that, given an angle in radians, answers the distance from the origin to a certain point. This point is in the image of the function, at the given angle. It is very much like being in charge of a cannon, and the function, given the direction we aim the cannon, tells us how far to shoot.

Of course, polar functions do not differ a lot from cartesian functions. Instead of a number we provide an angle; and instead of picking up the oriented distance to the x axis, we pick up the distance to the origin. Thus, both functions behave in the same way, because they take an amount and answer another amount. The interesting thing is to evaluate the functions in the polar space, and map the result into the cartesian plane. This can be extremely handy. For example, in comparison with polar coordinates, it is irritating and cumbersome to describe a semi-circle in cartesian coordinates. On the other hand, in polar coordinates, a circle becomes a constant, since a circle is a set of points that are at the same given distance from a given point, namely its center. The similarity between cartesian coordinate and polar coordinate functions, encourages the implementation of polar coordinate functions by subclassing the classes modeling cartesian coordinate functions. Accordingly, the plotter functions used will be instances of the class **ThetaRhoPlotterFunction**, which will be a subclass of **XYPlotterFunction**.

First step of evaluation in polar coordinates

By far the trickiest thing will be to properly evaluate a function in polar coordinates. This is not because evaluation is hard by itself, but because we are planning to map it into a cartesian plane.

Given a function to plot, how many times should we evaluate it in its domain? If the amount of points is too few, the function could look like a polygon instead of a curve. But if the amount of points is too large, then we generate too many useless points. These useless points are very

expensive in terms of execution time. They involve creation, evaluation, coordinate system mapping, scaling and then plotting. The worst is that they may turn out to be the same after mapping and scaling. This is especially true if the function takes low values almost all the time, except for a few spikes which alter the scaling in the ambient. The problem is that we do not know the behavior of the function before evaluating it, so we cannot determine a reasonable amount of samples to take until it is too late.

Our solution will be to evaluate a safe-and-sound number of times based on the size of the domain, and then to eliminate useless points during the scaling process, taking proper care in determining what useless means. Here, it will mean consecutive evaluation points that are equal, or almost equal, after scaling. For instance, the scaling process should leave just two scaled equal points for the constant function zero. We will deal with scaling later.

What is that safe-and-sound number of times? It depends on the function being evaluated. As we do not know, we will use a fixed value to multiply the **domain size**, namely: **graphSize x / domainBound size * (domainBound size max: 2 * Float pi).** Although it looks quite complicated, it just scales the amount of points for the size of each function's domain. Now, evaluation is different from cartesian evaluation because we need to map one coordinate space into another.

When mapping from polar coordinates into cartesian coordinates, we need the horizontal value bound. We would like a constant function to show a circle touching the horizontal and vertical edges of the graph. That means we will have to scale with respect to x, and to do that, we need the bound of the values of x. In cartesian coordinates, the bound was provided by the interval enclosing all the function domains. In polar coordinates, we will have to build that ourselves. Here is the evaluation method for the **ThetaRhoPlotterFunction** instances:

```
evaluate: anInteger

| deltaTheta currentTheta rhoTrans thetaTrans cRho |

valueCache ← (OrderedCollection new: anInteger + 1).

deltaTheta ← domain size / anInteger. “Increment between samples”

currentTheta ← domain start. “Samples begin here”

cRho ← function valueAt: currentTheta. “This is our first sample”

“And we translate it to cartesian coordinates”

thetaTrans ← cRho * currentTheta cos.

rhoTrans ← cRho * currentTheta sin.


“We now need bounds for y AND x, since we find out about x after translation”
```

```
xBound ← ClosedInterval from: thetaTrans to: thetaTrans.
yBound ← ClosedInterval from: rhoTrans to: rhoTrans.

valueCache add: thetaTrans @ rhoTrans. “We get our first sample in”
1 to: anInteger do: [:each |
        currentTheta ← deltaTheta * each + domain start. “New value in domain”
        cRho ← function valueAt: currentTheta. “New sample”
        “We grow our bounds with translated points”
        xBound growSoThatIncludes: (thetaTrans ← cRho * currentTheta cos).
        yBound growSoThatIncludes: (rhoTrans ← cRho * currentTheta sin).
        “And we add the translated point to the value cache”
        valueCache add: thetaTrans @ rhoTrans]
```

Note the care taken to initialize the bounds. The points are created from translated coordinates, and by the time the process ends, the functions originally in the polar coordinate system now can pretend to be functions in the cartesian coordinate system. There is one more step involved, the scaling. Of course, the plotter will first request the **xBound** from all the polar coordinate functions, then build its domain bound, send it to the grid plotter, which then will initialize its mapping capabilities, and then the plotter will be able to start the scaling process.

Second step of evaluation in polar coordinates

Scaling will be done here by means of **transformSpanToGraph: aPoint**. The scaling method in the plotter functions will get rid of the useless points. A point will be considered useless when the sum of the absolute values of the differences of the coordinates of this point and the last point scaled, is less than or equal to **1**. The scaling process will begin by mapping all the points into plot space, and then a second filtering pass will be applied. To reduce the burden when the domain size is large, and consequently the amount of points scaled is very large, only one collection of points will be used. Here is the source code for the polar coordinate scaling method:

```
scaleTo: aPlottingGrid
| lastPosition lastPoint currentPoint |
“First we map points in the span to points in the graph”
valueCache do: [:each | aPlottingGrid transformSpanToGraph: each].
“We will look for similar points, so we initialize some variables”
lastPosition ← 1. lastPoint ← valueCache at: lastPosition.
2 to: valueCache size do: [:each |
```

```
                currentPoint ← valueCache at: each.
                "If the last point we added is similar to the current one…"
                (lastPoint x - currentPoint x) abs +
                (lastPoint y - currentPoint y) abs > 1 ifTrue:
                        [lastPosition ← lastPosition + 1. "Then we move it back"
                        valueCache at: lastPosition put: currentPoint.
                        lastPoint ← currentPoint]].
        "So we are 'compacting' the value cache by eliminating useless points.  Also, we
        may have to process the last point in case we did not add it"
        (lastPoint = currentPoint and: [lastPosition > 1]) ifFalse:
                [lastPosition ← lastPosition + 1.
                valueCache at: lastPosition put: lastPoint].
        "We discard the top portion of the value cache"
        valueCache ← valueCache copyFrom: 1 to: lastPosition.
        self scaled: true
```

This completes the scaling process. We are now ready to plot the grid.

Grid plotting

Our attention will now go to the **CartesianGridPlotter** class. As we already know, it can translate between points in the span and points in the plot. This knowledge now enables it to determine where the axes are and how to center the grid. It also gives the aspect ratio of the span compared to the aspect ratio of the plot. This is very nice to know, because then, if the aspect ratio is 1, the grid will be composed of squares; whereas if the aspect ratio was not 1, the grid would be composed of rectangles.

Aspect ratio

The aspect ratio of a rectangle is defined as its width over its height. Hence, **aRectangle** that has an **extent** of **640@480** will have an aspect ratio of **4/3**. The idea behind this is that the grid will show how the graph is distorted in the requested plot size. For instance, a circle in a **640@480** plot will look like an ellipse. Accordingly, the grid's units should be rectangles of a **4/3** aspect ratio, because the aspect ratio of a circle's **span** is **1**. And if the aspect ratio of the **grid** is **plotAR**, and the **span**'s aspect ratio is **spanAR**, the combined aspect ratio of the graph inside the plot will be **plotAR * spanAR**. Now we know what the aspect ratio of the graph is, and so we can draw the grid and the axes properly.

Color schemes

We will need three colors to plot a grid, namely the background color, the main axes color, and the grid color. Changing these three colors, we will be able to mimic sheets of notebook paper, blackboards, and blueprint designs. Their names are Default, Arte, RecRoll, UBABlack, UBABlackGrid and UBAGreen. The default color preset can be set by sending **resetColors** to the grid plotter. The rest can be set by appending their names to **colorPreset**. For instance, the color preset Arte is set by sending **colorPresetArte**. Their names deserve some explanation. The preset Arte mimics the paper sheets of the Arte brand notebooks. The RecRoll preset next imitates a brand of recycled paper notebooks called RecRoll. I used those notebooks at UBA. The preset UBABlack models UBA's not-so-black blackboards with chalky axes and grids. The preset UBABlackGrid is a variation with black grids and axes, and it is my favorite. The last preset, UBAGreen, models UBA's green blackboards with chalky grids and axes. These colors may be accessed individually within the plotter by sending the messages **backgroundColor**, **axisColor** and **gridColor**.

Filling the background

The first thing we will do in the **CartesianGridPlotter** will be to prepare our **FormCanvas**. It is much easier to draw the axes and the grid on top of the background than filling the rectangles left between the grid and the axes. Let's simply do:

```
grid ← FormCanvas extent: graphSize depth: 32
```

But why a depth of 32? That means we will use true color with full support for alpha blending capabilities. It is possible to use alpha blending with less color depth, but we will choose to do our plots in a 32 bit deep form. In any case, we can send the message **asFormOfDepth:** to the form, or to do something a bit more elaborate such as the Heckbert median cut color reduction algorithm. Once we have our grid, it is time to fill it:

```
grid fillWith: self backgroundColor
```

This completes our filling of the grid. What should we draw next? If we draw the main axes first, then we will have to avoid them when drawing the grid. On the other hand, if we draw the grid first, we can safely draw the axes over it. Then, our next step is to draw the grid.

Drawing the grid

This part is tricky too. The behavior of the grid is dictated by numerous factors. First, the size of the rectangles drawn depends on the aspect ratio. Their position depends on both axes and the size of the plot. Let's examine this carefully.

The influence of the aspect ratio on the grid

To distort an initial square of the grid, first we need to know how big it is. For our purposes, we will use squares of **anInteger** pixels long sides. But which **anInteger**? We will start with a **baseCellSize** of **48**, and we will let the following procedure adjust this value so that there is a healthy and nice-looking number of grid rectangles.

calculateBaseCellSize

```
baseCellSize ← ((graphSize x max: graphSize y) / 10) rounded max: 8.
self aspectRatio > 1 ifTrue:
        [baseCellSize ← (graphSize y / self aspectRatio / 6) ceiling
                min: baseCellSize max: 4].
self aspectRatio < 1 ifTrue:
        [baseCellSize ← (graphSize x * self aspectRatio / 6) ceiling
                min: baseCellSize max: 4]
```

This method adjusts the cell size so that there are at least **6** horizontal and vertical grid lines, and avoids the basic cell side falling below **4** pixels.

If the aspect ratio of the plot is **1** then the cells, now **baseCellSize** high and wide, should remain the same. When the aspect ratio is greater than **1**, we should have rectangles with an extent of **baseCellSize * self aspectRatio @ baseCellSize**. Or, the other way around, vertical lines of the grid should be separated by **baseCellSize * self aspectRatio** pixels. A similar reasoning applies when the aspect ratio is less than **1**. Here are the methods that control how far apart horizontal and vertical lines of the grid should be:

gridXInterleave

```
"Answer the space between x axis guide lines"
self calculateBaseCellSize.
self aspectRatio > 1 ifTrue: [↑(baseCellSize * self aspectRatio) rounded].
↑baseCellSize
```

gridYInterleave

```
"Answer the space between y axis guide lines"
self calculateBaseCellSize.
self aspectRatio < 1 ifTrue: [↑(baseCellSize / self aspectRatio) rounded].
↑baseCellSize
```

The influence of the axes' position on the grid

At this point, we already have the functions evaluated and scaled. We also know the span and the plotting space. Then, we can certainly verify if the main x and y axes are included or not. These two cases will be handled differently.

If there are no axes in the span, then where should we draw the subgrid? We could follow the 0,0 coordinates and start from there, but then the subgrid could end up not centered in the plot. With no thicker axes to see, this looks odd. So, when there are no axes, we will follow the plot borders and center the subgrid with respect to them. But if the axes are in the plot, we would then like the subgrid to be centered with respect to the axes.

That is what we will do for each axis. If an axis is present, then the correspondent vertical or horizontal subgrid is centered at the axis, otherwise it is centered from the plot borders. Here is the main grid plotter method:

```
plotGrid
"Answer the grid generated by the current settings"
| drawX drawY |
grid ← FormCanvas extent: graphSize depth: 32.
grid fillColor: self backgroundColor.
“We first determine what kind of subgrid we need to plot”
(drawX ← self xInterval includes: 0)
        ifTrue: [self generateXZGridOn: grid using: self xInterval]
        ifFalse: [self generateXCGridOn: grid].
(drawY ← self yInterval includes: 0)
        ifTrue: [self generateYZGridOn: grid using: self yInterval]
        ifFalse: [self generateYCGridOn: grid].
“And then we plot the main axes”
drawX ifTrue: [self drawXAxisOn: grid using: self xInterval].
drawY ifTrue: [self drawYAxisOn: grid using: self yInterval].
↑self grid
```

The axes and grid lines are drawn using the rectangle filling methods we already saw in the protocol of **FormCanvas**.

Some of the selector names deserve an explanation. For each coordinate, x and y, there is an axis and a subgrid. The subgrid is a set of lines parallel to the given main axes. What we just discussed means that we have two different ways of drawing the subgrids, either centered around the axes or centered on the plot. Here, these centering methods are

referred to by the Z (centering around the axes or around zero) and C letters (plain centering on the plot). For instance, the method name **generateYCGridOn:** selector means to generate the y subgrid, centered on the plot. Finally, the axes are drawn after the subgrid is drawn. This is done to avoid the subgrid overwriting the main axes, which is not esthetically good.

Introduction to the plot engine

So far in our problem we have evaluated the functions, scaled them, and we have just drawn the grid. It is now time to draw the functions. We have seen that the plotter functions, when scaled, hold a **valueCache** that contains all the points to be drawn. Actually, these points give us the points of a polygon which we will draw on the grid. The idea is that we play connect the dots with such points, and that is why it is important to have at least two points.

Yet, for certain applications, it would be much nicer if we were able to apply some effects to our polygon. For instance, students of calculus know that one interpretation of the value of the integral of a function is a measurement of the area between the function and the x axis. Students of statistics find this very useful when plotting histograms and probability distributions, and they can usually derive a lot of information from those graphs. Students of multivariate calculus are often interested in the contour of certain three dimensional objects such as cylinders, cones, paraboloids and so on. Pie charts and bar graphs, with their variations, would be a great enhancement to our drawn polygon. And hey! We should also keep function colors in mind!

Function colors and the Munsell color system

Computers usually follow the RGB color space, in which the red, green and blue coordinates may take values between 0 and some fixed value like 31, 63 or 255. Each color is then represented by a triplet of those values, one value per color coordinate. The whole RGB color space has the shape of a cube.

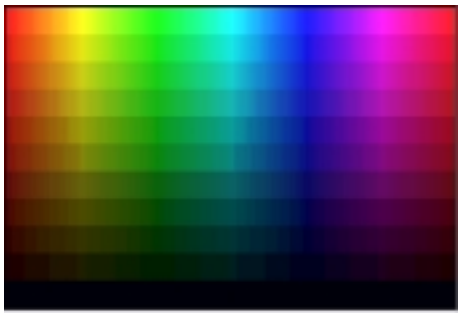
Here is a neat little problem. Choose colors such that they are "most" different. How do we do that? Let's get more detailed. We would like colors of the same brightness, yet, as different as possible.. But to do that in the RGB cube is not trivial! Things can get messy very quickly because of recursivity in the algorithms. To make things more complicated, the RGB cube does not allow an order relationship between colors as we can find for, say, the real numbers (this can prove to be a very tough problem in connection with the hash value of a color and to color quantization), so we encounter difficulty trying to choose colors sequentially.

Fortunately, it is very easy to solve this problem if we use another coordinate system. Instead of working in the RGB cube, we will work in the HSV color system. Let's examine it.

The Munsell color system space looks like a cylinder. Of course, we will use cylindrical coordinates to describe it. Cylindrical coordinates are an extension of polar coordinates. In polar coordinates, we choose an origin, and for each angle the function provides the distance to the origin where we should plot a dot. For instance, a circle in polar coordinates is a constant. In general, it is easier to describe circle-like figures in polar coordinates than in any other coordinate system.

But we need to describe a cylinder and not a circle. Thus we say well, the cylinder is the collection of all the parallel, same-radius circles that have their origins in a segment that is perpendicular to all those circles. Then, we can use a height shift value that lets us move in the segment to choose any particular circle, and then we can use polar coordinates within the circle to reach any point in the cylinder. Cylindrical coordinates are polar coordinates plus a shift axis.

In the Munsell color system cylinder, the segment goes from black to white, and it is referred to as the value component of any given color. Let's get in a circle in particular. The colors are arranged in such a way that all possible colors of the same apparent brightness are together in the circle. Evidently, all the colors in each circle are as bright as the value of the circle in question. Now, to get any color, we use polar coordinates. The angle part is called the hue, and by changing it we sweep all possible colors. As we get farther away from the center, colors are said to become more saturated, or more colorful so to speak. At the outer perimeter of the circle, we find pure colors. Getting closer to the center mixes each pure hue with the gray color at the center. In a sense, it is doing alpha blending between any given shade of gray and the pure color at the same brightness. Because each color can be described by their hue, saturation and value, this color system is also known as HSV. Here you can see the skin of the Munsell cylinder, at saturation **0.9** and with **11** different values, from zero to ten. Hues run from left to right, and values run from bottom to top. The left corresponds to a hue of **0**, while the bottom corresponds to a value of **0**. It was plotted by the **MunsellTree** plotter.



Back to our problem. The HSV color system has six familiar hues around its outer perimeter. We could fix the value and saturation, and then choose those basic six hues first. After those run out, we then could

choose the hues between each consecutive hue chosen before, and so on. This is exactly what the instances of **ColorStream** do. They are also the grounds upon which the **RainbowMorph** is based. The **RainbowMorph** changes its color over time, by means of the **step** method. Each time it steps, it will change its color to **aColorStream next**. At first, it will change coarsely, but as time goes by, the color stream will choose closer and closer colors. After a few minutes, it will smoothly fade from one shade to the next. At all times though, colors chosen will be as far apart as possible from all colors selected previously.

We want exactly this behavior for the function color assignment, i.e., to choose colors as far apart from each other as possible. It is also desirable to choose colors with the same brightness, that is, with the same saturation and value, because if not that would show when the plots are drawn and shown. If we allowed different saturations and values, we could end up with very bright colors together with pale ones. Hence, we will assign each function the color given by **aColorStream next**.

```
next
| newH |
newH ← colorStep * colorDelta + colorShift.
newH >= 1 ifTrue:
[colorStep ← 0. colorShift = 0
        ifTrue: [colorShift ← colorDelta / 2]
        ifFalse: [colorShift ← colorShift / 2. colorDelta ←colorDelta / 2].
        ↑self next] ifFalse: [colorStep ← colorStep + 1].
↑Color h: (h ← newH * 360) s: s v: v
```

The initialization method of **ColorStream** makes **colorDelta** to be **1/6**, and **colorShift** and **colorStep** to be **0**. Each time this method is executed, it goes around the outer perimeter of the saturation and value circle chosen in the HSV color system. When the turn is completed, the shift and the delta are updated so that new colors fall between colors already chosen.

Functions and alpha blending colors

Furthermore, functions will get the colors coming out from a **ColorStream** with a specific alpha blending value. This allows functions to overlap the grid and other functions, preventing them from overwriting the already existing graphics. Currently, the alpha blending plot value is **0.08**. Other effects, such as area filling, will receive other alpha blending values, such as **0.02** and even **0.005**, to differentiate the effect from the plot itself. These values will be held by the plot engine.

The plot engine

We referred to a few things that would enhance our simple polygon plot. Different ways to draw a function will be referred to as plot modes. Each plot mode will draw the polygon and enhance it in some way as it is being drawn. The object that will implement these plot modes is the plot engine.

The plot engine is an object that, taken an ambient for reference and a function to plot, will output the plot to a certain amount of plot targets. The ambient will be a grid plotter and it will provide information about the position of the axes. The plotter function will tell the plotter what color and plot mode to use. Plotter functions have a very flexible mechanism to tell the plot engine things. They have attributes that can be set and retrieved by name. Some of them are so important that they have specific accessors, such as the **plotMode**, the width of the plot, called **dotSize**, and the **color**. These are all considered to be attributes of the plotter function.

Plot targets

About the output, it is very desirable to be able to output the plot to more than one form canvas simultaneously. The first target will be a cache of the plot. We would not like to replot each time the plotter is moved in a Morphic world. On the other hand, it would be nice if we could see the plot being generated in real time. This implies that besides drawing on our cache, we will have to draw directly on the screen. To allow this, we will wrap each form canvas to be drawn on inside an instance of the class **PlotTarget.**

There is an additional benefit arising from using a cache. The plotter will be working in 32 bits, and so will be the cache. If the screen is set to something less than 32 bits, though, each draw operation will have its colors truncated. As a result, what is displayed on the screen will be the result of several truncated color operations. At then end, though, we may draw the whole cache and so we will display the plot with just one color truncation operation. The difference between these is quite noticeable.

Morphic worlds are drawn on a form canvas, but evidently the plotter may not be the only thing present in the display. To allow drawing directly over them as the plot engine works, plot targets will provide an offset to their form canvas. This is done so the plot engine only sees a form canvas on which it has to draw starting at **0@0**. Because the plot target will take care of drawing on the form canvas, it will implement a few methods to allow skewing the coordinates by the corresponding offset. For instance, if the function plotter is at **100@100** in the Morphic world, and its grid has an extent of **640@480**, the plot target will redirect the rectangle **0@0 corner: 640@480** to **100@100 corner: 740@580**. Incidentally, being able to display progress in realtime is also why we will

concentrate on synchronous enhancement of the polygon on the fly. Special effects look great when they appear on the screen as they are being drawn. Plot targets are given to the plot engine by using the method **addTarget: aPlotTarget**. The drawing methods implemented by instances of **PlotTarget** are:

line: startPoint to: endPoint width: aWidth color: aColor

line: startPoint to: endPoint width: aWidth color: aColor

withFirstPoint: aBoolean

These are very similar. What they do is to draw a line in the form canvas from **startPoint + offset** to **endPoint + offset**, with a dot size of **aWidth**, and with color **aColor**. Furthermore, the first point of the line can be skipped while drawing. This produces better quality plots. Since we will play connect the dots, we do not need to plot those dots twice (once when we arrive, and once when we proceed to the next one).

The plot engine's plot modes

We will describe the plot modes now, together with some examples of them in action. Some of the illustrations include a few additions to make them more clear. It is a thrilling experience to watch the function plotters draw these pictures on the fly.

The **PlotEngine** class currently has one concrete subclass, **XYPlotEngine**. Because the instances of **ThetaRhoPlotterFunction** translate the points into cartesian coordinates, it is not necessary to have a dedicated **ThetaRhoPlotEngine** class. The **XYPlotEngine** provides the following plot modes:

AlphaToOrigin

AlphaToXAxis

DiscreteDerivative

DownVolumeCylinder

DownRightVolumeCone

DownRightVolumeCylinder

OddConical

Standard

We will now describe these eight plot modes.

The standard plot mode

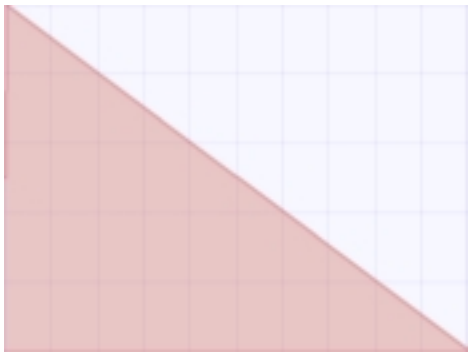
This mode takes the points from the plotter function's **valueCache** and simply draws a polygon on the plot targets. Here is the implementation:

```
plotStandard
"Produce a standard plot on the targets"
| last current |
last ← toPlot at: 1.
2 to: toPlot size do:
[:each |
        current ← toPlot at: each.
        targets do: [:some | some
                line: current to: last
                width: dotSize color: plotColor
                withFirstPoint: each = toPlot size].
        last ← current]
```

When this method is executed, the plot engine has the function's **valueCache** stored in **toPlot**, its plot width in **dotSize**, and its color in **plotColor**. The alpha value of **plotColor** is set to **0.08** by the plot engine. We can see here how each little line of the plot is drawn backwards, so that the **withFirstPoint:** plot method takes care of plotting the extra point only when necessary.

Examination of random sequences

A portion of a previous work regarding compression had to do with the distribution of the absolute values of the difference between pairs of consecutive elements taken from a stream. If the stream is generating numbers at random in a given range, the distribution of this particular amount can be proven to have a triangular shape like the one in the illustration below. In this case, the domain is **[0, 255]**. The high peak close to zero is **510**, and at zero there are **256** hits. As the difference increases by **1**, the hits decrease by **2**. The hit average for a range of width *n* is *(n+1)/3*.



Let's suppose now for a moment that various common compressors produce a random sequence of bytes (or whatever). We will see how well they perform at their task. For our tests, we will use the zip and rar compression algorithms. Both use the popular Lempel Ziv algorithm for string matching. After LZ, zip uses

Huffman, while rar uses a proprietary encoding mechanism. Rar also has dedicated “multimedia” algorithms. The triangular distribution for a random sequence will be left as a reference. We will examine both compressors working on a wav file. The file is an 8 bits, mono, 22khz sample rate, 666,108 bytes long file. The three plots here show the distribution of the uncompressed file, of the zip file (237,902 bytes), and of the rar file (234,000 bytes), left to right. Rar may choose to use its multimedia algorithms.

The histograms immediately below are normalized to a maximum hit value of 510. In this case, rar’s behavior is closer to random than zip’s. Note how the first histogram shows that the absolute values of the differences of consecutive values in this particular wav file are usually small. This behavior is quite common for sound files, regardless of bit depth and channels (even when not de-interleaved). If the histogram was of the difference of consecutive values alone, it would have two spikes at the left and right ends, with a big valley in between.

The area filling plot modes

Some plot modes will fill an area between the function's graph and the axes, or between the function's graph and a certain fixed point. This is the case of the **AlphaToOrigin** and **AlphaToXAxis** plot modes. In the first case, a line is drawn from each point of the function's graph to a fixed point. In the second case, a line is drawn from each point of the function's graph to its x axis projection. To solve these cases in general, the plot engine implements two private methods called **plotFillTo: aBlock** and **plotFillToPoint: aPoint**. Here is the implementation of the first. The other can be obtained by replacing the reference to **aBlock value: last** by **aPoint**.

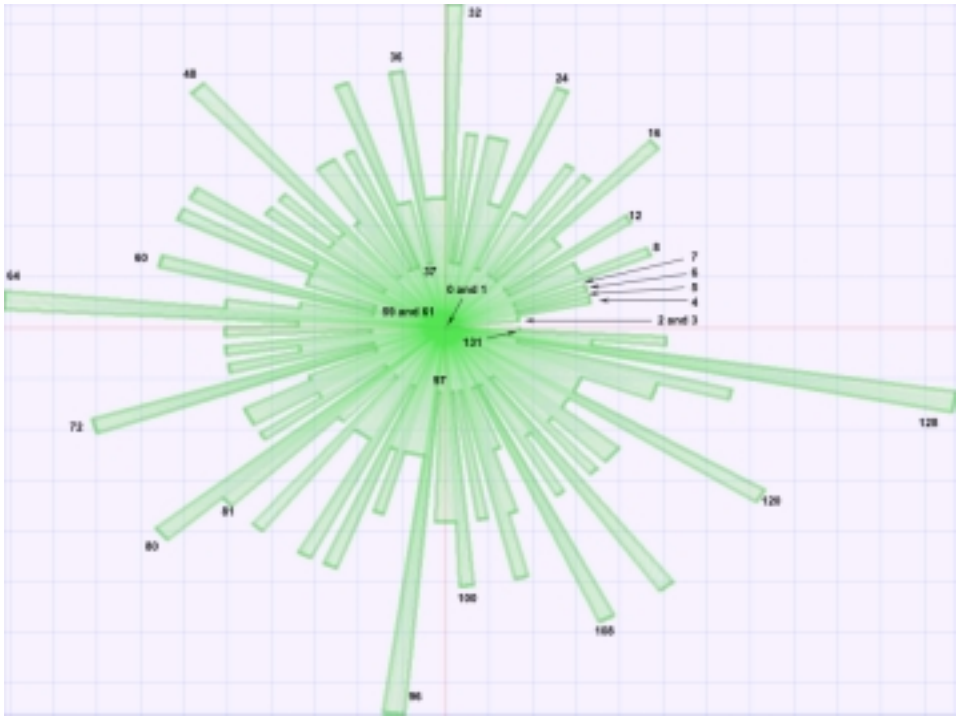
```
plotFillTo: aBlock
"Produce a standard plot on the targets, and for each point plotted fill the line
connecting the point plotted with aBlock value: plotted point"
| last current |
last ← toPlot at: 1.
“Effect for the first point”
targets do: [:other | other
        line: last to: (aBlock value: last) width: dotSize color: fillColor].
2 to: toPlot size do: [:each |
        current ← toPlot at: each.
        “Standard plot”
        targets do: [:some | some line: current to: last width: dotSize
                color: plotColor withFirstPoint: each = toPlot size].
```

```
"Effect for each point plotted"
targets do: [:more | more line: current to: (aBlock value: current)
        width: dotSize color: fillColor withFirstPoint: false].
last ← current]
```

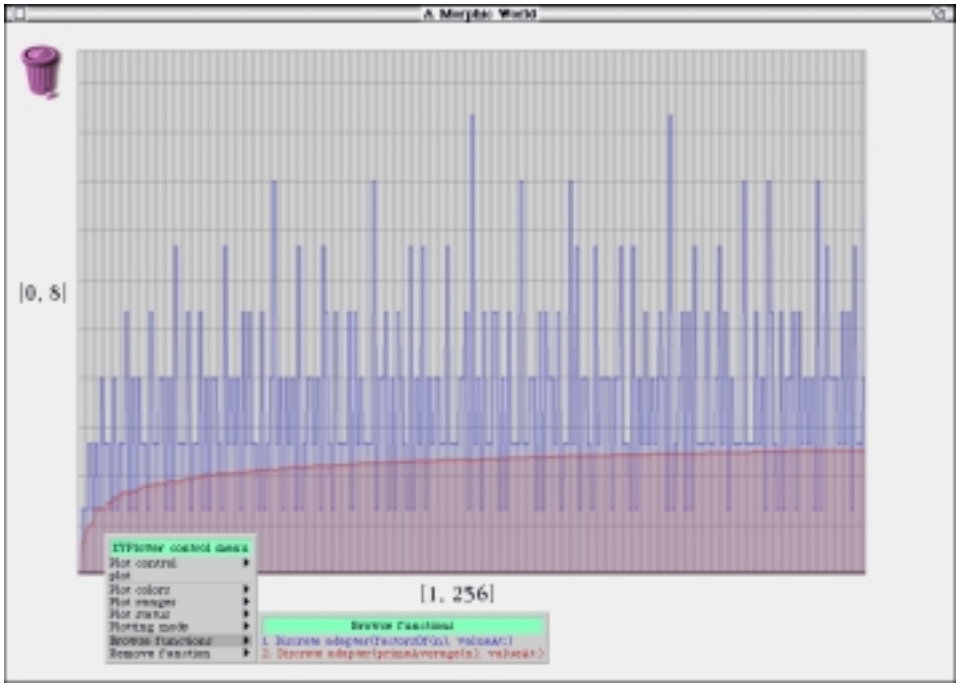
Here, **aBlock** is set to: **[:each | each x @ xAxisPosition]**. The variable **xAxisPosition** comes from the context in which the block is created. Its value is **ambient yForXAxis**. Note the care taken to draw the enhancement from the point referenced by **last**. The variable **fillColor** contains the function's color with an alpha value of **0.02**.

An application of the ThetaRhoPlotter in number theory

Imagine we took a function that, given an integer, answered the amount of prime factors in the given integer. With that function, we could also find the average prime factors per integer up to a given integer, and get another



function. Here are two plots on this topic. Incidentally, the average primes per integer up to *n* is asymptotically close to *log log n + M*, where *M* is the Mertens' constant which value is close to 0.57. In this first graph, note how the integers arrange themselves in rings. The innermost ring is the prime ring.



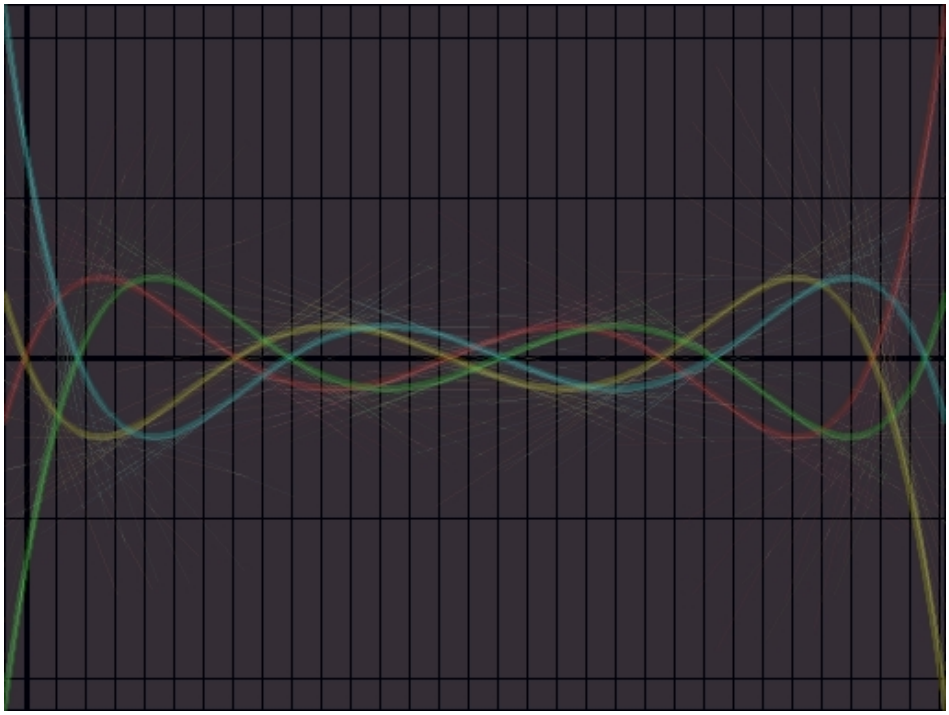
In the second plot, we can see how the prime average, in red, gets flat almost instantly. The blue function here is the green function from the first illustration. The first illustration is using the **AlphaToOrigin** plot mode, while the second illustration is using the **AlphaToXAxis** plot mode.

The discrete derivative plot mode

This plot mode will add small tangent lines to the graph. It is especially designed to draw such lines only when there has been a considerable variation in the slope of the curve being plotted. The color of the derivative lines is the function's color with an alpha value of **0.07**. Here is an example of this plot mode with four polynomials. The colors in this plot show the **ColorStream** in action (also note the tangent lines).

The graph dragging plot modes

The plot modes remaining take the graph and drag it on the targets, leaving some sort of trace while they do so. The idea of these methods came to me by accident. I was trying to get the x axis area filling mode to work, but while I was at it I made several mistakes. Those mistakes



showed that a simple process would make a simple plot into something much better. Even more, these effects could be designed such that the plotter would appear to be three dimensional.

In the **AlphaToOrigin** plot mode, area is filled between the graph and the origin. But it could also be thought of as if the points that form the graph were taken from the center after soaking them with ink. As they move toward their destination, they leave some ink on the way, thus filling the area between the graph and the origin. Graph dragging is a generalization of this thought. Points will be dragged by **aPoint**, which

will be sometimes fixed, sometimes variable. Again, in the plot engine this is implemented by two methods, one for fixed values and the other for variable values. Their names are **plotDraggedBy: aPoint** and **plotDraggedTo: aBlock**. Here is one of them:

```
plotDraggedBy: aPoint
"Produce a standard plot on the targets, and for each point plotted drag from that
point by aPoint"
| last current |
last ← toPlot at: 1.
“Effect for first point”
targets do: [:other | other line: last to: last + aPoint width: dotSize
                        color: dragColor].
2 to: toPlot size do: [:each |
        current ← toPlot at: each.
        “Standard plot”
        targets do: [:some | some line: current to: last width: dotSize
                color: plotColor withFirstPoint: each = toPlot size].
        “Effect for every plotted point”
        targets do: [:more | more line: current to: current + aPoint
                width: dotSize color: dragColor].
        last ← current]
```

In this case, the alpha blending value for **dragColor** is **0.005**, making the drag plot modes work best with dark grids.

There are four drag modes. Two of them, the cylindrical ones, use fixed drag points. The other two, the conical ones, use a variable drag value. The **downVolumeCylinder** plot mode drags the graph by **0 @ (self plotSize y * 2 / 3) rounded**, and the **downRightVolumeCylinder** drags by **self plotSize x / 10) rounded @ (self plotSize y / 3) rounded**. The **downRightVolumeCone** plot mode works with this block, taken from the method **XYPlotEngine>>plot:**

```
dragX ← self plotSize x / 14. dragY ← self plotSize y / 3.
↑self plotDraggedTo: [:each | (each x / 7 - dragX) rounded @
                        dragY rounded]
```

Finally, the **oddConical** plot mode works with this other block, also taken from **XYPlotEngine>>plot:**

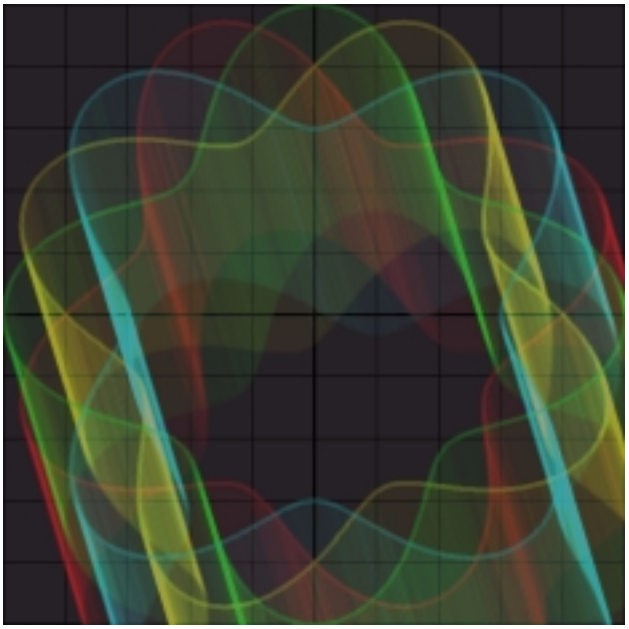
```
dragY ← self plotSize y / 3.
```

```
↑self plotDraggedTo: [:each | (each x / 5) rounded @
                              (dragY + each) rounded]
```

Because the drag modes usually create a circular shape, they do their best in polar coordinates.

An example of the graph dragging plot modes with the ThetaRhoPlotter

The functions drawn here are shifted sine functions. The plotter was instructed to use the **downRightVolumeCylinder** plot mode. The impression obtained is that the plotter is drawing in 3D!



Pending issues

Alas, so much alpha blending could be improved. Right now, and as you can find out after an examination of the illustrations involved, area filling plot modes suffer from an artifact. This artifact happens when a single pixel suffers several applications of some alpha blending color mixes. This causes color saturation especially in the **alphaToOrigin** plot mode, and "holes" in the plot drag modes.

Another artifact happens when several functions force different colors to be alpha blended with one another. Because alpha blending makes the first color drawn less and less important, it simply fades away. Right now, the alpha blending values are correct mainly because the default dot size is set to 5.

These problems would be fixed if each function was drawn on its own layer form. We would start drawing the functions in their layers, then the

effects added to them in other layers but without using alpha blending. So, we would have two forms per function. Then, we would add the effects to the function layer using an alpha blending mask, hence eliminating color saturation and holes. The final phase of this process would be to add the resulting function layers using regular color addition. To make this fast, we would use a very special object, **BitBlt**. More on **BitBlt** in the next sections!

Sometimes, though, the artifacts can make the plots look prettier. This is one of the reasons for the artifacts to remain there.

The function plotter itself

Now that we have described the processes by which functions are plotted, we need a controlling entity that will coordinate these processes. These entities will be instances of the **FunctionPlotter** class. Each plotter will have a grid plotter, a plot engine, and a collection of plotter functions. The most interesting method in a function plotter is its **plot** method. Here is the **plot** method found in **XYPlotter**:

```
plot

"Answer the plot"

| xBound yBound |

self functions isEmpty ifTrue: [↑self plotEmptyGrid].

xBound ← self domainBound.

xBound isNil ifTrue: [↑self errorOnMissingRegion].
```

This check is to avoid trying to plot inside a **nil** region. The error is **self notify: 'Missing region'**. If there is a valid region to plot in, then functions are evaluated if their point cache is invalid.

```
functions do: [:each | each valueCache isNil ifTrue:

        [each evaluate: (each domain size / xBound size * self plotSize x) floor]].
```

Once they are evaluated, we get the image bound and tell the grid plotter what the **span** is.

```
yBound ← self imageBound.

grid span: (xBound start @ yBound start corner: xBound stop @ yBound stop).

answer ← FormCanvas on: grid plotGrid.

self updateMorph.
```

The **updateMorph** mechanism implemented by the MorphicWrappers shows the empty grid first if the plotter has its Morphic counterpart.

```
functions do: [:each | each scaled ifFalse: [each scaleTo: grid]].
```

Then, functions are scaled…

```
plotEngine ambient: grid.
plotEngine removeAllTargets.
plotEngine addTarget: answer shift: 0@0.
self directDrawEnable ifTrue:
    [plotEngine
        addTarget: self morphicWrapper
        directDrawTarget shift: self morphicWrapper position].
```

Direct draw is a procedure by which the graph is drawn in Morphic in real time. In order to do that, one of the targets for the plot engine becomes the Morphic world's display, which is an instance of **FormCanvas**. An appropriate shift is given by the morph's position.

Now, the plot engine is told to plot each function on all the targets:

```
functions do: [:each | plotEngine plot: each].
```

And finally, if there is a wrapper morph for the plotter, then it is updated. Otherwise, the form is answered.

```
↑self hasMorphicWrapper ifTrue: [self updateMorph] ifFalse: [↑self answer]
```

The last **updateMorph** message is sent for a very special reason. The plot is drawn in true color. Yet, if the plot engine is drawing in Morphic, when the display depth is not 32, then things get drawn in special ways that are faster but that also lose some quality. Hence, the last update copies the form drawn in true color to the screen one last time, so that color reduction is applied only once for each pixel of the plot.

The function plotter in Morphic

Each plotter has its Morphic counterpart, which are instances of subclasses of the **SimplePlotterMorph** class. This morph provides basic functionality, such as its extensive double click menu. The menu controls the addition and removal of functions, the dot sizes, the colors, the plot modes, the plot sizes, the aspect ratio, the invalidation of point caches, etc.

The addition and removal of functions is done by adding submenus to the main menu, named Browse Functions and Remove Functions. These submenus have a list of the functions, each one showing in its current color for easy identification. Accordingly, there is a **ColoredMenuMorph** class, subclass of **MenuMorph**, to allow for colorful entries.

Most of the parameters for the plotter functions can also be accessed from the double click menu and its submenues. Plotter properties also have their place. The graph size can be changed, and also the aspect ratio of such sizes can be changed. Suggested plot sizes in the menu are based

on several widths, and the heights are calculated using the aspect ratio selected. The basic widths are 320, 400, 512, 640, 720, 800, 896, 960 and 1024, usual widths for standard video modes. Aspect ratios provided in the menu are 1, 6/5, 4/3 and phi. Personally, I do not like the phi aspect ratio. I like the 6/5 one much better, and I think it has to do with the aspect ratio of the human field of view.

Other plotter parameters accesible from the double click menu include the standard values for newly added functions, the color presets, the domain and image bounds, and the status of the direct draw procedure.

By implementing the **click:** method, the plotters are also able to catch clicks on them. A click is an event, and it also has a position. We as plotters can then ask our ambient where the point we have been clicked on is in the span, and then we can give the result to the cursor for everyone to see.

Finally, there is a GIF snapshot facility. This takes the form generated by the plotter and saves a file to the disk. Color reduction is usually needed, because our forms are in true color. The necessary devices for nice color reduction are described in the next section.

Color Reduction

When we described the difference between **Form** and **ColorForm** we mentioned that if we had a true color instance of **Form** with up to 256 colors, we could put those colors in a color array and generate a **ColorForm** that would take one-third of the space required by the **Form**, roughly speaking (we divide by 3 the bits per pixel required, but we also add a color table). Nevertheless, if we had **aForm** with 257 colors we would not be able to do that unless we reduced the amount of colors used by the form.

By the way this **ColorForm** would be useful for many other things. The GIF graphical format allows just 256 colors per image, for instance. Although this format is being replaced by PNG (which is in the public domain, compresses more yet is lossless, and supports more than 256 colors per image), it serves as an example of how color reduction can be useful. Furthermore, there is a **GIFReadWriter** class in Squeak, so we can use the color reduction to get a nice copy of our image and save a GIF image instead.

There are many ways to reduce the amount of colors used by an image. The simplest and quickest, but by far the most inelegant, is to take the bits used to represent each color component in the RGB system, and truncate them to a lower amount of bits. If we truncate enough so that our color space has just 256 colors, we win. This can be done by sending the message **asFormOfDepth: desiredBitsPerPixel** to any form. But in

this way we also lose a lot of density in our color space! We will usually dislike the 256 color space version of a true color image obtained by this method, in comparison to the original. However, we can also use **asFormOfDepth:** to increase the amount of bits per pixel used by a form.

A nice way to reduce the colors used by an image is to implement the Heckbert median cut color reduction algorithm. It is implemented and included in the function plotters package to provide GIF format snapshots of plots.

The Heckbert color quantization algorithm

Paul Heckbert's color quantization median cut algorithm (median cut algorithm) is described in a small paragraph of Heckbert's graduation thesis. The idea is simple, and the only factor that makes the algorithm difficult is the structure of the RGB cube, because it does not allow an order relationship between colors that is so nicely behaved as the order relationship in the real numbers.

Boxes and the minimization algorithm

We will define a box in the RGB cube to be a sub cube of the RGB color space. Boxes may contain a collection of colors. Such colors should be inside the box, in terms of the RGB color system. For instance, the hypothetical box **Color gray corner: Color white** should not contain **Color black**. Moreover, we will require that boxes containing colors are minimized, in the sense that a box containing colors must be the smallest box that contains such colors. If not, we see that a procedure similar to the one described for **ClosedInterval>>growSoThatIncludes:** done on the colors for each of the colors' RGB coordinates gives us the minimal box that contains such colors.

Let's think geometrically inside the RGB cube for a minute. Once boxes are minimized, it is natural to ask the boxes about their center. This will be the average between their **start** and **stop** colors, and it turns out that we will call this color the representative color for the box. It is also natural to ask boxes for the cube's axis upon which they take more space. We will call this dimension the dominant dimension for the box. If some axes tie, we will choose any of them.

The median and the splitting algorithm

The median of **aSortedCollection** is:

```
| pivot |
pivot ← aSortedCollection size // 2.
↑aSortedCollection size odd
```

```
ifTrue: [aSortedCollection at: pivot + 1]
ifFalse:[(aSortedCollection at: pivot) +
        (aSortedCollection at: pivot + 1) // 2]
```

When colors inside a box are sorted by its dominant dimension, we will define the box' median to be the component corresponding to the dominant dimension of the median of the box' sorted colors.

Now, if the box has more than one color, it is possible to split it by its median. Boxes can be thought of as being determined by a **start** and a **stop** color. All colors between **start** and **stop** are inside the box. The splitting algorithm generates two boxes from one, and those boxes are:

```
start corner: (stop copy at: dominantDimension put: self median).
(start copy at: dominantDimension put: self median) corner: stop.
```

Note that, strictly speaking, this process can generate boxes, rectangles, segments and points. We will consider them all to be boxes. The splitting algorithm also cuts the sorted collection of the box' colors at the median. The first half of this sorted collection goes into the first box, and the second half goes into the second box. It is important to avoid splitting boxes with method like **includes:**, because colors may be unevenly distributed since boxes may share portions of the RGB cube.

Incidentally, this process implies asking **aColor** for its **red**, **green**, and **blue** components a very large number of times. If we check the implementation for these messages, we will see that the methods imply doing some bit manipulation. This means colors end up doing a lot of bit shifting which will give the same results over and over again. This is a bottleneck, which is solved by implementing a **ColorProxy** object, that holds a color inside and caches its components.

Color quantization

Now we can do color quantization, based on the pieces we already have. First, we get one box with all the colors we want to quantize, and we minimize it. Then, if we need to get *n* quantized colors, we apply the splitting algorithm *n-1* times and we take the representatives from the boxes. Finally, we find the closest representative for each original color, and we are done.

Color mapping after quantization

We are done with color quantization, but as you will see, that is the easiest part. Now we have to take all the colors in the form and replace them by their corresponding representatives. That means that we will have to query a mapping from one set of colors to the other a very large number of times. For instance, there are 307,200 pixels in a **640@480** form. Unfortunately, when quantizing from true color there is no other

alternative. The complete process should take around 15 seconds. That could be acceptable, but what if we need something faster?

Color mapping à la BitBlt

If we decide to trade true color accuracy for speed, there is another way suggested by Dan Ingalls. There is a very special object in every Smalltalk called **BitBlt**. Its name comes from BIT BLock Transfer. Note that the actual movement of bits is not the main purpose of **BitBlt**, rather, it models the transfer itself. Movement of bits happens all the time when something changes on the screen, or when something is moved from one buffer to another.

Most transfers are in regard to operations on instances of **Form**. This transfer can be done in a multitude of ways. Each of these ways is called a combination rule. There is a combination rule that does what we need to do, albeit only in high color. Note that **BitBlt** is quite bit level operation oriented. This rule replaces colors using their raw bit values as indices for a replacement table. For instance, the index for **Color white** would be 32767 (15 bits, 5-5-5). The replacement table is an instance of **Bitmap,** a subclass of **ArrayedCollection**. Hence, **aBitmap** behaves like **anArray**. The difference is that the objects stored inside **aBitmap** are small integers. We can get new instances of **Bitmap** by sending the message **new: desiredSize**. When created, instances of **Bitmap** are filled with zeros.

For our color quantization purposes, we need to get a color replacement table. This table will be accessed with indices resulting from the raw bits of the colors to be replaced. At such indexed positions, it should contain the raw bits of the corresponding replacement color. Here we see why we have to go to high color: a true color replacement table would need too much memory! Keep in mind that we need to use these replacement tables because we want to use **BitBlt**. We could do with say a **Dictionary**, but that brings the problems of repeated querying and the hash of **aColor**.

In order to build our replacement table, we first truncate the colors in the original form we are given into high color, if needed. This loss of color information is barely noticeable in most cases, if noticeable at all. After truncation, our form has a depth of 16 bits per pixel at most. Then, we need to build the color replacement table so that we can replace colors by their corresponding representatives. To do that, we need to know which colors are used in the form. We already have those colors from the quantization process, where we asked **aForm** for its **colorsUsed**. We might guess that all colors inside a box are best matched by the box' representative, but this is not always true. Hence, we could try to find the best matching representative.

If we are inclined toward the second option, we can use the three dimensional Pythagorean theorem inside the RGB cube. The Pythagorean theorem needs a square root, but as it is not needed to determine whether a color is closer or farther, we can do with distances squared. Also, the interface for color components found in **Color** is based on floating point numbers generated from the bits stored inside colors. We will do with those bits instead, and, together with other considerations, we end up with a distance measurement ranging from **0**, color equality, to **3139587**, total color disparity between black and white.

By implementing the messages we need in **Color** itself, we avoid asking **aColor** about its components. The methods below are somewhat terse. My current (slow) machine needs **90** nanoseconds to perform an object assignment. Because of the enormous number of times these methods are executed, the toll of extra assignments that would make the code clearer is quite significant.

```
distanceTo: aColor
"Answer the distance to aColor, ranging from 0 to 3139587 in the RGB cube.
This is like || self - aColor ||↑2."
| aRGB blueSummand greenSummand redSummand |
aRGB ← aColor privateRGB.
redSummand ← (rgb bitShift: -20) - (aRGB bitShift: -20).
greenSummand ← ((rgb bitShift: -10) bitAnd: 16r3FF) –
                    ((aRGB bitShift: -10) bitAnd: 16r3FF).
blueSummand ← (rgb bitAnd: 16r3FF) - (aRGB bitAnd: 16r3FF).
↑(redSummand * redSummand) + (greenSummand * greenSummand) +
    (blueSummand * blueSummand)
```

The method above computes the distance between two colors.

It is also useful to compute the distance between a color and a 15 bit integer representation of a color. This is done by the following method:

```
distanceTo5bit: anInteger
"Answer the distance to anInteger, ranging from 0 to 3139587 in the RGB cube.
This is like || self - aColor ||↑2. anInteger is [5 bits red][5 bits green][5 bits blue]"
| blueSummand greenSummand redSummand |
redSummand ← (rgb bitShift: -20) - ((anInteger bitAnd: 16r7C00) bitShift: -5).
greenSummand ← ((rgb bitShift: -10) bitAnd: 16r3FF) –
                    (anInteger bitAnd: 16r3E0).
blueSummand ← (rgb bitAnd: 16r3FF) - ((anInteger bitAnd: 16r1F) bitShift: 5).
↑(redSummand * redSummand) + (greenSummand * greenSummand) +
```

```
(blueSummand * blueSummand)
```

Once we have our color replacement table, we need to perform a **BitBlt** operation. The combination rule used will be **Form paint**, which overwrites the destination with the source. What is written is the corresponding replacement of the color at the source with the color taken from the replacement table. We collect **BitBlt**'s result in the destination form. We could do as follows to quantize the colors in our source form down to 256 colors:

```
destination ← ColorForm extent: source extent depth: 8.

aBitBlt ← BitBlt toForm: destination.

aBitBlt sourceForm: source; combinationRule: Form paint; colorMap: colorMap;

    sourceOrigin: 0@0; destOrigin: 0@0; destRect: source boundingBox;

    sourceRect: source boundingBox; copyBits
```

In the code above, **source** is our form, **colorMap** is our bitmap, the origin points are an indication of where **BitBlt** should start to work, the source and destination rectangles are for clipping purposes, and finally the message **copyBits** starts the process. This is about 25 times faster than the form peeking and poking method.

Acknowledgements

Every piece of work done is based on previous work by other people. I would like to thank the Squeak Central for creating Squeak in 1996. Without their hard work, the whole MathMorphs project would not be a reality today.

The Squeak Central is Alan Kay, Dan Ingalls, Ted Kaehler, Scott Wallace, John Maloney, Andreas Raab, Kim Rose and Pat Brecker. Kim and Mark Guzdial had the idea of making a book about Squeak that would also have the spirit of the original "Smalltalk-80: The Language" book. Mark and Kim have been very helpful in the complex process of writing this present publication. It is a great honor to participate in this book.

I would also thank Dan Ingalls for his help and suggestions on **BitBlt** so that it would help the Heckbert color quantization algorithm, and so that it could be used to fix the plotter's artifacts described here. Dan has also been especially supportive about writing this chapter.

My sister, Florencia Valloud, introduced the Munsell tree and color system to me. She helped in the construction of the Munsell plotters and the **ColorStream** class.

I would like to thank Leandro Caniglia and the members of the MathMorphs project for creating it and for keeping it a healthy environment in which to work. Many thanks go to Luciano Notarfrancesco, Pablo Malavolta, Pablo Shmerkin, Francisco Garau, Gerardo Richarte, Alejandro Weil, Ariel Schwartzman,Valeria Murgia, Eric Rodríguez Guevara, and Ariel Pacetti. I am also grateful toward the Universidad de Mar del Plata (Mar del Plata University), where a MathMorphs presentation was given. It was the first time I gave a lecture to a general audience, and the first time I showed the function plotters in public.

Finally, I would like to thank Catana Lucero. Her support and human qualities have been instrumental for this work, both during the months of writing this chapter and also while I was building the function plotters. Catana also read this chapter several times giving valuable suggestions, and her work truly is on every page. I am very glad I met her by means of a random chat program, October 22nd, 1998. There is no doubt that luck has been on our side.

Table of contents

Index of contents