Rapid prototyping in vision algorithms

Prof.dr.ir. Bart M. ter Haar Romeny

Eindhoven University of Technology
Department of Biomedical Engineering
Biomedical Image Analysis
B.M.terHaarRomeny@tue.nl

Vision Solutions 2003

Introduction

In an imaging chain, where a customer specifies a task to be carried out on images, we often encounter the following processes:

The task has to be coded into an algorithm, e.g. 'find the locations of corners on buildings', or 'find a spiral in a meteorological image', 'find text with digits on a chip'. This algorithm has to run fast in a realtime application. Whether it works well has to be established by the expert, who gives advise to tune the task specification if the performance is not as expected.
The focus is often too early on the fast implementation. Vision algorithms are characterized by an enormous variability. Complete fields exist, that perform 'geometric reasoning', 'mathematical morphology', 'semantic reasoning', perceptual grouping'. They boil down to translating a task, specified by a human in his language, to a mathematical formulation that can be programmed.

In this paper we focus on the translation of the task into the algorithm. Machines are nowadays so powerful, that it is appropriate to leave the low level functions to the computer, invisible to the programmer, and focus on the high level tasks of the design.

The design of the algorithm is often the most difficult task in the whole chain, and takes a substantial time to develop. It is not uncommon to study weeks on finding the best algorithm for the task. The most suitable language to parse the task into an algorithm is mathematics. Shapes are well described by edges, ridges, curvatures etc. These can be found from the data by taking (high order) derivatives. To learn from large databases, useful tools are implemented with singular value decomposition and Eigenvalues from linear algebra, or methods from statistics and neural networks.

Therefore:

- we don't like to worry about the internal representation of the data, or to manipulate many copies of the same routine for all data formats (bytes, short integer, complex etc.);
- we like to design the algorithm step by step, correcting a step if necessary: we want an interpreter;
- we like a system, that has a complete set of mathematical building blocks on board;
- we like to do symbolic and fast numeric calculations in a single system.

**Mathematica**

### Integrated symbolic and numeric capability

*Mathematica* is a programming language (www.wolfram.com) that fulfills our needs. It is an interpreter. It can do symbolic manipulations:

\[
\int \sqrt{x} \sqrt{a-x} \, dx
\]

\[
\text{Out}[2] = \sqrt{a} - a \left( \frac{a \sqrt{x}}{4} + \frac{x^{3/2}}{2} \right) - \frac{1}{4} a^2 \text{ArcTan} \left[ \frac{\sqrt{a-x} \sqrt{x}}{a-x} \right]
\]

It is fast in numerical calculations (version 5 is for many routines faster than Matlab):

\[
\text{In}[3]= \text{matrix} = \text{Table}[\text{Random[]}, \{100\}, \{100\}];
\text{Timing}[\text{Eigenvalues[\text{matrix}]}];
\]

\text{Out}[4]= \{0.031 \text{ Second}, \text{Null}\}

\[
\text{In}[5]= \text{matrix} = \text{Table}[\text{Random[]}, \{512\}, \{512\}];
\text{Timing}[\text{Fourier[\text{matrix}]}];
\]

\text{Out}[6]= \{0.266 \text{ Second}, \text{Null}\}
It has a powerful document processor, capable of displaying text, mathematical notation, graphics, animations and code all in one integrated document. This paper is completely written in Mathematica. This makes documents interactive and templatable. The full code of very topic is given with the discussion, and can be easily adapted.

---

**Full range of 2D, 3D, 3D-t graphics**

It has powerful graphics:

```
In[7]:=
ListPlot3D[Import["mr128.gif"][[1, 1]], Mesh -> False];
```

```
In[8]:=
<<Graphics``;
pts = {{0, 0}, {1, 2}, {-1, 3}, {0, 1}, {3, 0}};
Show[Graphics[{Hue[0], Line[pts], GrayLevel[0], Spline[pts, Cubic]}], PlotRange -> All];
```

```
In[11]:=
<<RealTime3D``
```
```
In[12]:=
PlotVectorField3D[{y, -x, 0} / z, {x, -1, 1}, {y, -1, 1}, {z, 1, 3}, VectorHeads -> True];
```

```
In[13]:=
<<Default3D``
Flexible and complete I/O

It can easily incorporate external programs, e.g. written in C++, by 'installing' them into the kernel. This reads a DICOM image:

\[
\text{In}[14]:= \text{Install["c:/tmp/DicomImport.exe"]};
\]

\[
\text{In}[15]:= \text{Show[im = LoadDicom["C:/tmp/ct016.dcm"][[1]]];}
\]

\[
\text{In}[16]:= \text{Dimensions[im[[1, 1]]]}
\]

\[
\text{Out}[16]= \{768, 768\}
\]

Functional programming style and pattern matching

Programming in mathematica is done in a functional style. It typically lead to short, compact code. It has powerful and fast command to map functions on arrays, and for pattern matching:

\[
\text{In}[17]:= \text{data = ReadList["c:/tmp/dictionary.dat", String];}
\text{Length[data]}
\]

\[
\text{Out}[18]= 118617
\]

\[
\text{In}[19]:= \text{Take[data, 20]}
\]

\[
\text{Out}[19]= \{\text{aardvark, aardvarks, aaronic, abaca, abaci, aback, abacterial, abacus, abacuses, abaft, abalienate, abalienated, abalienation, abalone, abalones, abandon, abandoned, abandonedly, abandonnee}\}
\]

\[
\text{In}[20]:= \text{Select[data, (# == StringReverse[#] && StringLength[#] > 2) &]}
\]

\[
\text{Out}[20]= \{\text{adinida, aha, ana, ana, anna, bib, bob, boob, civic, dad, deed, deified, deled, did, dud, eke, ene, ere, ese, esse, eve, eye, gag, gig, hah, huh, kayak, kook, level, madam, malayalam, minim, mom, mum, nisin, non, noon, nun, pap, peep, pep, pip, poop, pop, pup, radar, redder, refer, reviver, rotator, rotor, sagas, sees, sexes, shahs, sis, solos, sos, stats, succus, suus, tat, tenet, tit, tnt, toot, tot, wow}\}
\]

\[
\text{In}[21]:= \text{Map[f, \{a, b, c\}]}
\]

\[
\text{Out}[21]= \{f[a], f[b], f[c]\}
In[68]:= wordLengths = Map[StringLength, data];
   ListPlot[Table[Count[wordLengths, i], {i, 1, Max[wordLengths]}],
   PlotJoined -> False, PlotStyle -> PointSize[.02]];

Distribution of wordlengths in an English dictionary of 118617 words.

Functions are n-dimensional

Most functions are n-dimensional. E.g. to take the Fourier transform of a 4D dataset of over a million voxels:

In[24]:= im5D = Table[Random[], {32}, {32}, {32}, {32}];

In[25]:= Timing[Fourier[im5D]] // First
Out[25]= 1.344 Second

Shape, texture, motion: derivatives of images, differential geometry

Derivatives of images tell something about local changes: the first derivative detects intensity changes, and acts as an edge detector.

Derivatives of discrete data have to be calculated in a so-called regularized manner: by convolution of a Gaussian derivative function. This important result follows from a first principles approach, which will not be explained here. For details see [ter Haar Romeny 2003].

The one-dimensional Gaussian function is given by

\[
\text{gauss}[x, \sigma] := \frac{1}{\sigma \sqrt{2\pi}} e^{-\frac{x^2}{2\sigma^2}};
\]

In 2D we may multiply two Gaussian functions for \( x \) and \( y \):
This is the function \texttt{gD} to take a derivative of an image:

\begin{verbatim}
In[30]:= gD[im_, nx_, ny_, σ_] := Module[{x, y, kx, ky, mid, tmp},
  kx = N[Table[Evaluate[D[gauss[x, σ], {x, nx}]], {x, -6 σ, 6 σ}]];
  ky = If[nx == ny, kx, N[Table[Evaluate[D[gauss[y, σ], {y, ny}]],
    {y, -6 σ, 6 σ}]]];
  mid = Ceiling[Length[#1] / 2] & ;
  tmp = Transpose[ListConvolve[{kx}, im, {{1, mid[kx]}, {1, mid[kx]}}]];,
  Transpose[ListConvolve[{ky}, tmp, {{1, mid[ky]}, {1, mid[ky]}}]]];
\end{verbatim}

Let us calculate the edge magnitude $\sqrt{L_x^2 + L_y^2}$ of an MRI image of $256^2$ pixels:

\begin{verbatim}
In[31]:= SetDirectory["C:\\tmp"];
  im = Import["mx256.gif"][[1, 1]];
  Timing[grad = \sqrt{gD[im, 1, 0, 1]^2 + gD[im, 0, 1, 1]^2};]
\end{verbatim}

\begin{verbatim}
Out[33]= {0.109 Second, Null}
\end{verbatim}
In[34]:= << Graphics`; SetOptions[ListDensityPlot, Mesh -> False];
    DisplayTogetherArray[
    {ListDensityPlot[im], ListDensityPlot[grad]}];

We can also show the zero crossings of the Laplacian $L_{xx} + L_{yy}$:

In[36]:= \[\sigma\] = 1; ListContourPlot[\[gD\] im, 2, \[\sigma\]] + \[gD\] im, 0, 2, \[\sigma\]],
   Contours -> \{0\}, ContourShading -> False, ImageSize -> 200];

Or at a larger (coarser) scale:

In[37]:= \[\sigma\] = 2; ListContourPlot[\[gD\] im, 2, \[\sigma\]] + \[gD\] im, 0, 2, \[\sigma\]],
   Contours -> \{0\}, ContourShading -> False, ImageSize -> 200];


## Corner detection

It turns out that reasoning in an *intrinsic* coordinate system is attractive. Here in every pixel a coordinate system is made that has one axis (w) perpendicular to the isophote (line of constant intensity in an image), and the other (v) tangential to the isophote. We call these *gauge coordinates*. Here is the definition:

\[
\text{In}[1]:= \text{gauge2D}[f_, \text{nv}_-, \text{nw}_-; \text{nv} \geq 0, \text{nw} \geq 0] := \\
\quad \text{Module}[[\text{Lx}, \text{Ly}, v, w], w = \frac{\text{Lx}, \text{Ly}}{\sqrt{\text{Lx}^2 + \text{Ly}^2}}; v = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}.w; \\
\quad \text{Simplify}[\text{Nest}[[v.(\text{D}[\#1, x], \text{D}[\#1, y]) & ], \text{Nest}[[w.(\text{D}[\#1, x], \text{D}[\#1, y]) & ], f, \text{nw}], \text{nv}]/. \\
\quad \{\text{Lx} \to \text{D}[f, x], \text{Ly} \to \text{D}[f, y]]];
\]

This is the second order derivative to \(v\), a good ridge detector:

\[
\text{In}[2]:= \text{gauge2D}[\text{L}[x, y], 2, 0]
\]

\[
\text{Out}[2]= \frac{\text{L}^{[0,2]}[x, y] \text{L}^{[1,0]}[x, y]^2 - 2 \text{L}^{[0,1]}[x, y] \text{L}^{[1,1]}[x, y] \text{L}^{[0,0]}[x, y] + \text{L}^{[1,1]}[x, y]^2 \text{L}^{[2,0]}[x, y]}{\text{L}^{[0,1]}[x, y]^2 + \text{L}^{[1,0]}[x, y]^2}
\]

Again, a long formula with derivatives. Here is the more readable form, using *Mathematica*’s pattern matching:

\[
\text{In}[3]:= \text{shortnotation}[\text{expr}_, \text{nx}_-, \text{my}_-] := \\
\quad \text{Module}[[\text{nx}, \text{my}], \text{DisplayForm}[\text{Simplify}[\text{expr} /. \text{Derivative}[\text{nx}_-, \text{my}_-][\text{x}_-, \text{y}_-] \to \\
\quad \text{Subscript}[^{\text{L}}, \text{StringJoin}[\text{Table}[^{\text{x}}, \{\text{nx}\}], \text{Table}[^{\text{y}}, \{\text{my}\}]]]]]]; \\
\text{In}[4]:= \text{gauge2D}[\text{L}[x, y], 2, 0] /\text{shortnotation}
\]

\[
\text{Out}[4] /\text{DisplayForm}=
\frac{-2 \text{L}_x \text{L}_y \text{L}_y + \text{L}_{xx} \text{L}_y^2 + \text{L}_x^2 \text{L}_{yy}}{\text{L}_x^2 + \text{L}_y^2}
\]

The following function writes the numerical code by replacing all derivatives by the numerical function \(gD\):

\[
\text{In}[42]:= \text{gauge2D}[\text{im}_-, \text{nv}_-, \text{nw}_-, \sigma_-; \sigma > 0] := \text{Module}[[\text{im0}], \text{gauge2D}[\text{L}[x, y], \text{nv}, \text{nw}]/. \\
\quad \text{Derivative}[\text{nx}_-, \text{ny}_-][\text{x}_-, \text{y}_-] \to gD[\text{im0}, \text{nx}, \text{ny}, \sigma] / . \text{im0} \to \text{im}]; \\
\text{In}[43]:= \text{im} = \text{Import}["\text{hands.gif}"][1, 1]; \\
\quad \text{Lvv} = \text{gauge2D}[\text{im}, 2, 0, 3]; \\
\quad \text{DisplayTogetherArray}[[\text{ListDensityPlot} /\emptyset[\text{im}, \text{Lvv}], \text{ImageSize} \to 450]]
\]
And $L_{vv} L_{w}^2$ is a good corner detector:

```mathematica
In[46]:= im = Import["Utrecht256.gif"][[1, 1]];  
corners = gauge2DN[im, 2, 0, 3] gauge2DN[im, 0, 1, 3]^2;  
DisplayTogetherArray[ListDensityPlot /@ {im, corners}, ImageSize -> 450];
```

**Edge preserving smoothing**

We can get rid of noise by *blurring* the image. But then we also blur the edges, so we have to pay a price. This can be solved by not blurring at the edges. How do we do this?

Blurring is described by the diffusion equation:

$$\frac{\partial L}{\partial t} = \nabla \cdot \nabla L$$

Perona and Malik [1991] introduced a *conductivity coefficient* ($c$) in the diffusion equation to make the diffusion adaptive to local image structure:

$$\frac{\partial L}{\partial t} = \nabla \cdot c \nabla L$$

where the function $c = c(L, \frac{\partial L}{\partial x}, \frac{\partial^2 L}{\partial x^2}, \ldots)$ is a function of local image differential structure, i.e. depends on local partial derivatives. When $c$ is large, we blur a lot, when $c$ is small, we have a small kernel, and blur just a little. We make $c$ so that it is only small at edges:

$$c = e^{-\frac{\nabla^2 f}{\epsilon^2}} = e^{-\frac{(\frac{\partial L}{\partial x})^2 (\frac{\partial L}{\partial y})^2}{\epsilon^2}}$$

To work on discrete images, we replace (with the `Replace` operator, short notation `/`) every occurrence of a spatial derivative in the right-hand side of the P&M equation (`pmc1` resp. `pmc2`) with the scaled Gaussian derivative operator `gD`

We solve the equation symbolically:
In[48]:= Clear[im, σ, k]; c = \[ExponentialE]^\((x^2 + y^2)/k^2\); formula = \[PartialD]_x (c \[PartialD]_x L[x, y]) + \[PartialD]_y (c \[PartialD]_y L[x, y]) // Simplify

Out[49]= \[ExponentialE]^\((x^2 + y^2)/k^2\) \((k^2 - 2 L[x, y] + (k^2 - 2 L[x, y]^2) L[2, 0][x, y])\)

Lots of derivatives. But the numerical program is written in a single line by replacing (with \.) all derivatives with the gD function:

In[50]:= pm[im_, σ_, k_] = formula /. Derivative[n__, m__][L][x__, y__] \[Function] gD[im, n, m, σ]

Out[50]= \[ExponentialE]^\((x^2 + y^2)/k^2\) \((k^2 - 2 gD[im, 1, 0, σ] gD[im, 1, 0, σ]) gD[im, 1, 1, σ] - 4 gD[im, 0, 2, σ] - 4 gD[im, 0, 1, σ] gD[im, 1, 0, σ] + (k^2 - 2 gD[im, 0, 1, σ]) gD[im, 2, 0, σ]\)

We calculate the variable conductance diffusion first on a simple small (64x64) noisy test image of a black disk (minimum: 0, maximum: 255):

In[51]:= imdisk = Table[If[(x - 32)^2 + (y - 32)^2 < 300, 0, 255], {y, 64}, {x, 64}];
nnoise = Table[100 Random[], {y, 64}, {x, 64}];
im = imdisk + noise; ListDensityPlot[im, ImageSize -> 120];

Figure 2. Simple 64x64 test image of a black disk on a white background (intensities 0 resp. 255) with additive uniform noise (amplitude=100).

A forward-Euler approximation scheme now becomes particularly simple:

In[54]:= personalamilkc1[im_, δs_, σ_, k_, niter_] := Module[{}, evolved = im;
Do[evolved += δs (pm[evolved, σ, k]), {niter}]; evolved];

where im is the input image, δs is the time step, σ is the scale of the differential operator, k is the conductivity control parameter and niter is the number of iterations. Here is an example of its performance:
In[55]:=  
line = {Red, Line[{{0, 32}, {64, 32}}]}; DisplayTogetherArray[
  {ListDensityPlot[#, Epilog -> line] & /@ {im, imp = Peronamlikcl[im, .1, .7, 25, 20]},
    ListPlot /@ {im[[32]], imp[[32]]}}, ImageSize -> 380];

Figure 3. Top left: input image; top right: same image after variable conductance diffusion, with operator scale \( \sigma = 0.8 \) pixels, timestep \( \delta s = 0.1 \), \( k = 200 \), nr. of iterations = 10. Bottom row: intensity profile of middle row of pixels for both images. The edge steepness is well preserved, while the noise is substantially reduced.

Other schemes are possible, like the Euclidean shortening flow \( \frac{\partial L}{\partial s} = L_{yy} L_r^2 - 2 L_x L_{xy} L_y + L_{xx} L_y^2 \), where the blurkernel is flattened along the edge:

In[56]:=  
es[im_, steps_, \( \sigma_\), range_] := Module[{\( \delta s \), int}, \( \delta s = range / steps; \)
  int = im; Do[int += \( \delta s (gD[int, 0, 2, \( \sigma \)] gD[int, 1, 0, \( \sigma \)]^2 - 2 gD[int, 0, 1, \( \sigma \)]
    gD[int, 1, 0, \( \sigma \)] gD[int, 1, 1, \( \sigma \)] + gD[int, 0, 1, \( \sigma \)]^2 gD[int, 2, 0, \( \sigma \)]) /\n    (gD[int, 0, 1, \( \sigma \)]^2 + gD[int, 1, 0, \( \sigma \)]^2), {steps}]; int];

This is an example for an ultrasound image with its particular speckle pattern:

In[57]:=  
us = Import["us.gif"][[1, 1]];
  DisplayTogetherArray[{ListDensityPlot[us], ListDensityPlot[es[us, 6, .8, 9]]}];

Figure 4. Euclidean shortening flow on a 260 x 345 pixel ultrasound image (source: www.atl.com).
A rapid prototyping toolkit

It is important to shorten the design cycle, in order to speed up the development of the full vision system. At the Department of Biomedical Engineering we have focused our prototyping development on Mathematica. So far we have experienced very steep learning curves with students, which showed in the development of a series of library functions (each project took 5.5 weeks, starting from scratch with little programming experience, but a good background in technical and biomedical sciences). Some examples:

- Adaptive histogram equalization
- Edge preserving smoothing
- Active shape segmentation
- Face recognition by Eigen-image analysis
- CT-thorax tissue classification by multi-scale texture analysis
- Deblurring Gaussian blur
- Color thickness coding on 3D volume visualizations
- Deformable contours ('snakes' and 'levelsets')
- Watershed segmentation
- Image matching by minimization of mutual information
- Background subtraction in MRI by entropy minimization
**MathLink**: connection to Java, parallel functions, OpenGL, C++, Matlab, ...

The following function is a free OpenGL viewer in *Mathematica*, written by J.P. Kuska (http://phong.informatik.uni-leipzig.de/~kuska/mathgl3d/v3/):

```math
In[58]:= Get["MathGL3d`OpenGLViewer`"];
In[59]:= d1[x_List, t_] := D[#1, t] & @ x;
d2[x_List, t_] := D[#1, t, t] & @ x;
norm[x_List] := x/Sqrt[x.x];
FrenetFrame[x_List, t_, phi_, r_] :=
  Module[{xp, xpp, tangent, binormal, normal},
    xp = d1[x, t]; xpp = d2[x, t];
    tangent = norm[xp];
    binormal = norm[Cross[xp, xpp]];
    normal = Cross[binormal, tangent];
    x + normal*r*Cos[phi] + r*binormal*Sin[phi]]

In[63]:= blankknot = ParametricPlot3D[
  Evaluate[FrenetFrame[{Cos[t] + 2 Cos[3 t], Sin[t] + 2 Sin[3 t], Sin[2 t]}, t, phi, 0.5]],
  {phi, 0, 2 Pi}, {t, 0, 2 Pi}, PlotPoints -> {18, 100}, DisplayFunction -> Identity];
In[64]:= MVClear[]; MVShow3D[blankknot,
  MVTexture -> RasterArray[Map[GrayLevel, {{1, 0}, {0, 1}}, {2}}],
  MVTextureMapType -> MVMeshUVMapping,
  MVScaleTexture -> (6, 40), MVGrayBackground -> True];
In[65]:= MVPasteGraphics[];
```
Conclusion

Our experiences with the use of a high level programming language for the rapid prototyping of vision routines are highly encouraging. The speed of the internal routines has come to a level that development cycles can be made short and flexible.

The final realtime application, which often has to be applied on huge datasets, can be made in a program of choice, like C/C++, Fortran, or even ASIC or graphics hardware. Such a step however is most appropriately taken when the design of the algorithm is finalized.

Computer vision is a science, and a little bit an art. Pixel manipulations are being replaced by differential geometric reasonings, learning from huge databases, and semantic reasoning about the questions raised about the image. Rapid prototyping is also a science, and a little bit of art. This short paper may have pointed to a new and exciting direction in vision solutions' efficiency.

Further references


- http://www.wolfram.com/services/

- http://library.wolfram.com/infocenter