Idea Cheat Sheet

Conventions (Not Enforced)

A – Capital letters are parameters
x – Lower case letters are streams
Variables beginning with f, n, r, c are ranges variables

Basic Data Types

Basic C Types: char, short, int, float, double, unsigned char, unsigned short, unsigned int
Complex Types: complex, dcomplex
64 bit integer type: dint
Address sized integer: INT

Index Notation

range n = 32; Takes on values 0..31
#n INT having size of range n (#n == 32)
y[n] = x(n); Scalar to vector. Form 1 vector token from #n scalar tokens.
y[n] = x(n-Ovrl); Scalar to vector with overlap of Ovrl. Form 1 vector token from #n scalar tokens, saving Ovrl for the next vector.
y[n] = x(n-#n/2); Scalar to vector with overlap size floor(#n/2)
y(n) = x[n]; Vector to scalar. Form #n scalar tokens from 1 vector token.
[n]y = x(n); Demultiplex x into #n streams.
y = x + x(-2); Add x with x delayed by 2;
y = x(-3); x delayed by 3;
y += x(-n); Sum tokens x(0) + x(-1) + … x(-(#n-1))
y += x(-n)/#n; y is average of last n samples of x
[f]y[r] = x[r] + f; Family of #f streams, e.g., [2]y[r] = x[r]+2.
range c = random(2)%32+32; Variable range
y[c] = x(c); Variable sized array
[r]y[c] = x[r][c]; Family of size r from the rows of x.

Defining Parameters

X = 3; Define variable x to be a parameter value of 3.
X[] = {1,2,3}; Set X to the vector with values 1,2,3
range n = 3; X[n] = {1,2,3}; same as above
A[][] = {{1,2,3},{4,5,6},[7,8,9}}; set A to a 3x3 matrix
complex Z = {cos(1.0,sin(1.0)};

Typing Rules

Expressions may be implicitly typed:
x = 1; x is an int
y = x+2; y is an int
y2 = z.re + 3; y2 is float
y3 = 10*z; y3 is complex if z is
Explicit typing can be used to cast types:
stream complex z = 1.0;
int x = z.re+z.im;
Circular feedback expressions must have explicit type:
int x = x(-1) + 1;

Arithmetic and Functions of Numbers

All C syntax arithmetic expressions permitted
Arithmetic expressions extended to include complex type.
complex I = {0,1}; Set I to sqrt(-1)
y[r] = x[r]*3+4 +5*I;
z(r) = x[r]/(y[r]+1);
Exponentiation:
y[r] = x[r]`2;
y[r] = x[r]`(8+2*I);
Standard C math library functions
y = sqrt(x);
y = sqrt(-x + 3*I); Compute the square root of –x + 3i.
y = exp(x/12); Compute e'(x/12).
Math library works on parameters or streams
A = log(3); A = log10(100);
A = abs(-5 +3*I); Compute the magnitude |–5 + 3i|.
A = sin(5.0/3.0); Compute the sine of 5/3.

Collapsing Operators

y += x[c]; Sum of elements of x.
y += x[10+n]; Sum the #n elements of x beginning at element 10.
y[c] += x[r][c]; Sum of columns of x.
y[r] += x[r][c]; Sum of rows of x.
y[r] += [f]z[r]; Family sum of vectors.
y *=x[c]; Product of elements of x.
y >?= x[c]; Max of elements of x.
y <?= abs(x[c]); Min magnitude of elements of x.
y &= (x[c] != 0); All elements of x[c] do not equal 0.
y |= (x[c] != 0); Some element of x[c] is not equal to 0.
z[r][c] += x[r][k] * y[k][c]; Matrix multiplication.
z[r] += x[r][k]*y[k]; Matrix vector multiplication.
z += x(-i)*C[i]; FIR filter.
Given: x[r][c] = …; range r3 = 3; range c3 = 3; range rb = #r-#r3+1; range cb = #c-#c3+1;
y[rb][cb] += x[rb+r3][cb+c3]*K[r3][c3]; 3x3 image filter.
int ia[r][c] = x[r][c] != 0; Create binary image.
ib[rb][cb] &= ia[rb+r3][cb+c3]; 3x3 erosion filter.
ic[rb][cb] |= ia[rb+r3][cb+c3]; 3x3 dilation filter.

Stream Source Generating Functions

stream x = 3; Define x to be a stream of constant int value.
x = uniform(3); Stream of uniformly distributed floating point values between 0 and 1. Initial seed is 3.
float x = normal(2); Stream of normal distributed values (mean 0, stddev 1).
complex x = x_normal(2);
float x = osc(3.0,2.0,1.0); Oscillator with frequency 3.0 radians per sample, amplitude 2.0 and initial phase of 1.0.
complex x = x_osc(3.0,1.0,0.0); Complex oscillator

Constructing a Few Simple Matrices

Given range n = 100;
y[r][c] = x[c](r); Matrix of size #r *#c.
y[n][r][c] = x[r][c](n); 3d array of size #n*#r*#c.
y[r][c] = x(r*#c+c); Matrix of #r*#c values.
Eye[r][c] = r==c; Identity matrix when #r == #c.
Zeros[r][c] = 0;
Linspace[n] = n*(4.7-1.2)/(#n-1) + 1.2;
Vector of 100 equally-spaced numbers from 1.2 to 4.7.
Rowvec[n] = 3+n; Vector of values 3,4,5,..,101, 102.
y[r][c] = r==c ? x[r][c] : 0.0;
Matrix whose diagonal is the diagonal entries of x.

Portions of Matrices and Vectors

Given range n = 100;
y[n] = x[2+n]; The 2nd to 101st element of x.
y[n] = x[2*n + 1]; Elements 1,3,5,…199 of x.
Given x is defined as x[r][c] = … then:
y[c] = x[5][c]; Elements in the 5th row of x.
y[n] = x[5][n]; First #n elements in the 5th row of x.
y[n] = x[n][5]; First #n elements in the 5th column of m.
y[r1][c1] = x[r1+5][c1+6];
Submatrix of size #r1*#c1 beginning at location 5,6.
y[r1][c1] = x[r1+i][c1+j];
Submatrix of x of size #r1*#c1 beginning at varying location i, j.

Setting Subsections of a Matrix

Given x[r][c] = …; y[c] = …; z[r] = … then:
a[n] = set(y,10.0,3); a = y with 3rd element replaced by 10.
a[r][c] = rset(x,y,10); a = x with 10th row replace by y.
a[r][c] = cset(x.z,10); a = x with 10th column replaced by z.
a[r][c] = set(x,4,5,10.0); a = x with 4th row and 5th column set to 10.

Partitioning and Concatenation

Given: v[r] = …; m[r][c] = …
[f]va[rf] = part_fam(v); Partition vector into family of vectors.
v2[r] = concat_fam([f]va);
[f]ma[rf][c] = rpart_fam(m); Family row part.
[f]mb[r][cf] = cpart_fam(m); Family column part.
m2[r][c] = rconcat_fam([f]ma);
m3[r][c] = cconcat_fam([f]mb);
vs[rs] = part_strm(v,Nmax); Partition v into multiple tokens that are vectors of size at most Nmax.
v3[r] = concat_strm(vs,Nmax,#r); Collect vectors partitioned by part_strm into vector of original size.
mrs[rs][c] = rpart_strm(m,Rmax);
mcs[r][cs] = cpart_strm(m,Cmax);
m4[r][c] = rconcat_strm(m,Rmax,#r);
m5[r][c] = cconcat_strm(m,Cmax,#c);

Solving Linear Equations

range c = #r; complex a[r][c] = …; complex x[c] = …
y[r] = solve(a,x); Solve for y in equation x[c] += a[r][c]*y[r];
b[r][c] = inv(a); Inverse of square matrix a.
b[r][c],p[c] = lu(a); LU factorization.
q[r1][c],r[c][c1] = qr(a); QR factorization.
v[r][c],d[c] = eig(a); The columns of v are the eigenvectors of a, and the values of d are the eigenvalues of a

Plotting

Given stream x = …, y[c] = …, th[c], z[r][c] = …
by = bar(y);
iz = image(z);
py = plot(y); Line plot of index (range c) vs. y.
py = plot(#c-c,y); Line plot of #c-c vs. y.
py = polar(y); Polar display of complex data.
py = polar(y,th); Polar display of real radius (y) and angle (th) data.
sy = scatter(y,sqrt(y)); Scatter plot of y vs. sqrt(y).
sx = scope(x); One input plot of x.
sx = scope(x,sqrt(x)); Two input plot of x and sqrt(x).
sy = spectrogram(y);
sz = surf(z); 3-d surface plot.

Transposes and Dot Products

B[c][r] = A[r][c]; Transpose of parameter.
y[c][r] = m[r][c]; Transpose of stream.
C[c][r] = conj(A[r][c]); Conjugate transpose.
d += x[c]*y[c]; Dot product.
e[r][c] = x[c]*z[r]; Outer product.

Finding and Gathering

Given v[r], m[r][c]:
indx[n] = find(v); Index of all nonzero values of v.
indx[n] = find(v,MaxN); Index of first MaxN nonzero values.
value[n] = v[indx[n]]; Gather.
value[n] = gather(v,indx); Gather using function.
rows[n],cols[n] = find(m); Row and column indices of non zer0 values of m.
rows[n],cols[n] = find(m,MaxN); Row and column indices of at most MaxN non-zero values of m.
values[n] = m[rows[n]][cols[n]]; Gather.
values[n] = gather(m,rows,cols); Gather using function.

Conditional Processing

```
Given c = random(4)%3; y = uniform(1);
if (c>2) {  x = y+1; } else {  x = y-1; }
if (c) {  w = y;  } Conditional data production.

switch(c) {
case 0: { z = y+1; } // Each case has an implicit break.
case 1: { z = y-1; }
case 2: { z = y; }
default: { z = 0; } // Default if c is not 0, 1, or 2
}
```

Iterative Processing

```
z = normal(1);  y = uniform(2);
do (x1=1,x2=1) {
  pop y;  // get new value of y for each iteration
  x3=x1+x2+y;
  x1_new=x2+z; // value of z is held
  x1=x1_new;
  x2=x3;
  push x2;  // output new value of x2 for each iteration
  p1 = print(x1);  // prints previous value of x1
  p4 = print(x1_new); // prints new value of x1
} while(x1 < 1000);
// final value of all variables besides x2 are available
outside loop

while (y1=z+1; x1=1; x1<100) {
  x1 = x1+1;
  x2 = sqrt(x1);
  y1 = func(x1,y1);
}
// only initialized variables y1 and x1 are visible on output

for (y1=0, x1=1; x1<100; x1 = x1+1) {
  x2 = sqrt(x1);
  y1 = func(x2,y1);
}
// only initialized variables x1 and y1 are visible on output
```