Appendix 1 Mathematical Computing

not exhaustive (or exhausting)
  pi benchmark
  thirteen orders of magnitude performance
  open-source options

A1.1 LANGUAGES

C
  C++ object-oriented combine data and methods
  types
  binary, hex
  int,long range
  signed, unsigned
  32 bit -2,147,483,648 to 2,147,483,647
  64 bit -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
  addressing memory, large data sets, simulations
  float, double precision
  matissa, exponent
  single (32 bit) approx 7 decimal digits
  double (64 bit) approx 15 decimal digits
  compilers
  GCC https://gcc.gnu.org/
  LLVM https://llvm.org/

```
/*
* pi.c
* Neil Gershenfeld 2/6/11
* pi calculation benchmark
* pi = 3.14159265358979323846
*/

#include <stdio.h>
#include <time.h>
```

```
#define NPTS 1000000000

void main() {
   int i;
   double a,b,c,pi,dt,mflops;
   struct timespec tstart,tend;
   clock_gettime(CLOCK_REALTIME,&tstart);
   a = 0.5;
   b = 0.75;
   c = 0.25;
   pi = 0;
   for (i = 1; i <= NPTS; ++i)
      pi += a/((i-b)*(i-c));
   clock_gettime(CLOCK_REALTIME,&tend);
   dt = (tend.tv_sec+tend.tv_nsec/1e9)-(tstart.tv_sec+tstart.tv_nsec/1e9);
   mflops = NPTS*5.0/(dt*1e6);
   printf("NPTS = %d, pi = %f\n",NPTS,pi);
   printf("time = %f, estimated MFlops = %f\n",dt,mflops);
   }
```

....

Intel 8088 0.000001 GFlop
Intel i7-8700T
gcc pi.c -o pi -lm
0.57 GFlop
gcc pi.c -o pi -lm -O3 -ffast-math
9.37 GFlop
Rust https://www.rust-lang.org/
Go https://go.dev/
memory safety, garbage collection, race conditions
Python https://www.python.org/
aspects of APL (most favorite), Lisp (least favorite)
Anaconda distribution, package manager https://www.anaconda.com/products/distribution
interpreter overhead

....

```
#
# pi.py
# Neil Gershenfeld 1/23/17
# calculation of pi by a scalar sum
# pi = 3.14159265358979323846
#

import time

NPTS = 100000000
a = 0.5
b = 0.75
```

```
c = 0.25
pi = 0
start_time = time.time()
for i in range(1,(NPTS+1)):
   pi += a/((i-b)*(i-c))
end_time = time.time()
mflops = NPTS*5.0/(1.0e6*(end_time-start_time))
print("NPTS = %d, pi = %f"%(NPTS,pi))
print("time = %f, estimated MFlops = %f"%(end_time-start_time,mflops))
```

Intel i7-8700T 0.029 GFlop
NumPy https://numpy.org/
overload operators

```
#
# numpi.py
# Neil Gershenfeld 1/23/17
# calculation of pi by a numpy sum
# pi = 3.14159265358979323846
#

from numpy import *
import time

NPTS = 100000000
a = 0.5
b = 0.75
c = 0.25
start_time = time.time()
i = arange(1,(NPTS+1),dtype=float64)
pi = sum(0.5/((i-0.75)*(i-.25)))
end_time = time.time()
mflops = NPTS*5.0/(1.0e6*(end_time-start_time))
print("NPTS = %d, pi = %f"%(NPTS,pi))
print("time = %f, estimated MFlops = %f"%(end_time-start_time,mflops))
```

Intel i7-8700T 0.47 GFlop approx 10x faster
Numba https://numba.pydata.org/

```
#
# numbapi.py
# Neil Gershenfeld 2/6/20
# calculation of pi by a Numba sum
# pi = 3.14159265358979323846
#
```

```
import time
from numba import jit

NPTS = 100000000
@jit(nopython=True)
def calc():
   a = 0.5
   b = 0.75
   c = 0.25
   pi = 0
   for i in range(1,(NPTS+1)):
      pi += a/((i-b)*(i-c))
   return pi
pi = calc() # first call to compile the function
start_time = time.time()
pi = calc() # second call uses the cached compilation
end_time = time.time()
mflops = NPTS*5.0/(1.0e6*(end_time-start_time))
print("NPTS = %d, pi = %f"%(NPTS,pi))
print("time = %f, estimated MFlops = %f"%(end_time-start_time,mflops))
```

Intel i7-8700T 4.87 GFlop approx 10x faster

JavaScript ECMAScript https://www.ecma-international.org/publications-and-standards/standards/ecma-262/

JIT compilation

```
<html>
<head>
<script>

//
// pi.html
// Neil Gershenfeld 1/24/17
// pi calculation benchmark
// pi = 3.14159265358979323846
//

function serial_benchmark() {
   var points = parseInt(document.getElementById('serial_points').value)
   var a = 0.5
   var b = 0.75
   var c = 0.25
   var pi = 0
   var tstart = Date.now()/1000
   for (var i = 1; i <= points; ++i)
```

```
      pi += a/((i-b)*(i-c))
   var tend = Date.now()/1000
   var mflops = points*5.0*1e-6/(tend-tstart)
   document.getElementById('div_pi_serial').innerHTML = 'pi: '+pi
   document.getElementById('div_time_serial').innerHTML = 'time: '+(tend-tstart).toFixed(1)+'s'
   document.getElementById('div_flop_serial').innerHTML = 'estimated MFlops: '+mflops.toFixed(1)
   }

function reduce_benchmark() {
   var points = parseInt(document.getElementById('reduce_points').value)
   var a = 0.5
   var b = 0.75
   var c = 0.25
   var tstart = Date.now()/1000
   var array = new Float64Array(points)
   var pi = array.reduce(function(sum,val,i,arr){return sum+a/(((i+1)-b)*((i+1)-c))},0)
   var tend = Date.now()/1000
   var mflops = points*5.0*1e-6/(tend-tstart)
   document.getElementById('div_pi_reduce').innerHTML = 'pi: '+pi
   document.getElementById('div_time_reduce').innerHTML = 'time: '+(tend-tstart).toFixed(1)+'s'
   document.getElementById('div_flop_reduce').innerHTML = 'estimated MFlops: '+mflops.toFixed(1)
   }

function parallel_benchmark() {
   document.getElementById('div_flop_parallel').innerHTML = 'estimated MFlops: calculating ...'
   var threads = parseInt(document.getElementById('parallel_threads').value)
   var points = parseInt(document.getElementById('parallel_points').value)
   var results = []
   var workers = new Array(threads)
   var blob = new Blob(['('+parallel_worker.toString()+'())'])
   var url = window.URL.createObjectURL(blob)
   var tstart = Date.now()/1000
   for (var t = 0; t < threads; ++t) {
      workers[t] = new Worker(url)
      workers[t].addEventListener('message',function(evt) {
         results.push(evt.data.sum)
         workers[evt.data.index].terminate()
         if (results.length == threads) {
            var tend = Date.now()/1000
            var mflops = (threads*points)*5.0*1e-6/(tend-tstart)
            var pi = results.reduce(function(x,y){return x+y},0)
            document.getElementById('div_pi_parallel').innerHTML = 'pi: '+pi
            document.getElementById('div_time_parallel').innerHTML = 'time: '+(tend-tstart).toFixed(
            document.getElementById('div_flop_parallel').innerHTML = 'estimated MFlops: '+mflops.toF
            }
         })
      workers[t].postMessage({points:points,index:t})
      }
   window.URL.revokeObjectURL(url)
   }

function parallel_worker() {
   self.addEventListener('message',function(evt) {
      var points = evt.data.points
      var index = evt.data.index
      var a = 0.5
      var b = 0.75
      var c = 0.25
      var sum = 0
      var istart = 1+points*index
      var iend = points*(index+1)
      for (var i = istart; i <= iend; ++i)
         sum += a/((i-b)*(i-c))
      self.postMessage({sum:sum,index:index})
      })
   }

</script>
</head>
<body>

<button onclick='serial_benchmark()'>calculate pi serial</button><br>
number of points: <input type='text' id='serial_points' value='1000000000' size=10>
<div id='div_pi_serial'>pi:</div>
<div id='div_time_serial'>time:</div>
<div id='div_flop_serial'>estimated MFlops:</div>

<br>

<button onclick='reduce_benchmark()'>calculate pi reduce</button><br>
number of points: <input type='text' id='reduce_points' value='10000000' size=10>
<div id='div_pi_reduce'>pi:</div>
<div id='div_time_reduce'>time:</div>
<div id='div_flop_reduce'>estimated MFlops:</div>

<br>

<button onclick='parallel_benchmark()'>calculate pi parallel</button><br>
number of workers: <input type='text' id='parallel_threads' value='4' size=3><br>
points per thread: <input type='text' id='parallel_points' value='1000000000' size=10>
<div id='div_pi_parallel'>pi:</div>
<div id='div_time_parallel'>time:</div>
<div id='div_flop_parallel'>estimated MFlops:</div>
```

....

Intel i7-8700T 3.73 GFlop

A1.2 PARALLEL COMPUTING

cores, threads

```
//
// threadpi.cpp
// Neil Gershenfeld 3/1/20
// calculation of pi by a C++ thread sum
// pi = 3.14159265358979323846
//
#include <iostream>
#include <chrono>
#include <thread>
#include <vector>
unsigned int npts = 2e7;
unsigned int nthreads = std::thread::hardware_concurrency();
std::vector<double> results(nthreads);
void sum(int index) {
   unsigned int start = npts*index+1;
   unsigned int end = npts*(index+1)+1;
   double result = 0;
   for (unsigned int i = start; i < end; ++i)
      result += 0.5/((i-0.75)*(i-0.25));
   results[index] = result;
   }
int main(void) {
   double pi = 0;
   std::thread threads[nthreads];
   auto tstart = std::chrono::high_resolution_clock::now();
   for (int i = 0; i < nthreads; ++i)
      threads[i] = std::thread(sum,i);
   for (int i = 0; i < nthreads; ++i) {
      threads[i].join();
      pi += results[i];
      }
   auto tend = std::chrono::high_resolution_clock::now();
auto dt = std::chrono::duration_cast<std::chrono::microseconds>(tend-tstart).count();
   auto mflops = npts*nthreads*5.0/dt;
   std::cout << "npts: " << npts << " nthreads: " << nthreads << " pi: " << pi << '\n';
   std::cout << "time: " << 1e-6*dt << " estimated MFlops: " << mflops << '\n';
   return 0;
   }
```

Intel Xeon Platinum 8175M 2x 24 x 2 cores, 96 threads 267 GFlop

```
#
# numbapip.py
# Neil Gershenfeld 2/6/20
# calculation of pi by a Numba parallel sum
# pi = 3.14159265358979323846
#

import time
from numba import njit,prange

NPTS = 10000000000
@njit(parallel=True,fastmath=True)
def calc():
   a = 0.5
   b = 0.75
   c = 0.25
   pi = 0
   for i in prange(1,(NPTS+1)):
      pi += a/((i-b)*(i-c))
   return pi
pi = calc() # first call to compile the function
start_time = time.time()
pi = calc() # second call uses the cached compilation
end_time = time.time()
mflops = NPTS*5.0/(1.0e6*(end_time-start_time))
print("NPTS = %d, pi = %f"%(NPTS,pi))
print("time = %f, estimated MFlops = %f"%(end_time-start_time,mflops))
```

Intel 2x Xeon Platinum 8175M 315 GFlop
Web Workers https://developer.mozilla.org
pi.html
Intel 2x Xeon Platinum 8175M 152 GFlop
GPUs
NVIDIA CUDA https://developer.nvidia.com
log reduction

```
//
// cudapi.cu
// Neil Gershenfeld 3/1/20
// calculation of pi by a CUDA sum
// pi = 3.14159265358979323846
//
#include <iostream>
#include <chrono>
```

```
#include <string>
using namespace std;
uint64_t blocks = 1024;
uint64_t threads = 1024;
uint64_t nloop = 1000000;
uint64_t npts = blocks*threads;
void cudaCheck(string msg) {
   cudaError err;
   err = cudaGetLastError();
   if (cudaSuccess != err)
   cerr << msg << ": " << cudaGetErrorString(err) << endl;
   }
__global__ void init(double *arr,uint64_t nloop) {
   uint64_t i = blockIdx.x*blockDim.x+threadIdx.x;
   uint64_t start = nloop*i+1;
   uint64_t end = nloop*(i+1)+1;
   arr[i] = 0;
   for (uint64_t j = start; j < end; ++j)
      arr[i] += 0.5/((j-0.75)*(j-0.25));
   }
__global__ void reduce_sum(double *arr,uint64_t len) {
   uint64_t i = blockIdx.x*blockDim.x+threadIdx.x;
   if (i < len)
      arr[i] += arr[i+len];
   }
void reduce(double *arr) {
   uint64_t len = npts >> 1;
   while (1) {
      reduce_sum<<<blocks,threads>>>(arr,len);
      cudaCheck("reduce_sum");
      len = len >> 1;
      if (len == 0)
         return;
      }
   }
int main(void) {
   double harr[1],*darr;
   cudaMalloc(&darr,npts*sizeof(double));
   cudaCheck("cudaMalloc");
   auto tstart = std::chrono::high_resolution_clock::now();
   init<<<blocks,threads>>>(darr,nloop);
   cudaCheck("init");
   reduce(darr);
   cudaDeviceSynchronize();
   cudaCheck("cudaDeviceSynchronize");
   auto tend = std::chrono::high_resolution_clock::now();
auto dt = std::chrono::duration_cast<std::chrono::microseconds>(tend-tstart).count();
```

```
   auto mflops = npts*nloop*5.0/dt;
   cudaMemcpy(harr,darr,8,cudaMemcpyDeviceToHost);
   cudaCheck("cudaMemcpy");
   printf("npts = %ld, nloop = %ld, pi = %lf\n",npts,nloop,harr[0]);
   printf("time = %f, estimated MFlops = %f\n",1e-6*dt,mflops);
   cudaFree(darr);
   return 0;
   }
```

NVIDIA A100 6192 cores 2,102 GFlop
Numba+CUDA

```
#
# numbapig.py
# Neil Gershenfeld 3/1/20
# calculation of pi by a Numba CUDA sum
# pi = 3.14159265358979323846
#
from numba import cuda
import numpy as np
import time
block_size = 1024
grid_size = 1024
nloop = 1000000
npts = grid_size*block_size
@cuda.jit
def init(arr,nloop):
   i = cuda.blockIdx.x*cuda.blockDim.x+cuda.threadIdx.x;
   start = nloop*i+1;
   end = nloop*(i+1)+1;
   arr[i] = 0;
   for j in range(start,end):
      arr[i] += 0.5/((j-0.75)*(j-0.25));
@cuda.jit
def reduce_sum(arr,len):
   i = cuda.blockIdx.x*cuda.blockDim.x+cuda.threadIdx.x;
   if (i < len):
      arr[i] += arr[i+len]
def reduce(arr,npts):
   len = npts >> 1
   while (1):
      reduce_sum[grid_size,block_size](arr,len)
      len = len >> 1
      if (len == 0):
         return
darr = cuda.device_array(npts,np.float64)
```

```
init[grid_size,block_size](darr,nloop) # compile kernel
reduce(darr,npts) # compile kernel
start_time = time.time()
init[grid_size,block_size](darr,nloop)
reduce(darr,npts)
cuda.synchronize()
end_time = time.time()
mflops = npts*nloop*5.0/(1.0e6*(end_time-start_time))
harr = darr.copy_to_host()
print("npts = %d, nloop = %d, pi = %f"%(npts,nloop,harr[0]))
print("time = %f, estimated MFlops = %f"%(end_time-start_time,mflops))
```

multi-GPU

```
//
// cudapit.cu
// Neil Gershenfeld 3/1/20
// calculation of pi by a CUDA multi-GPU thread sum
// pi = 3.14159265358979323846
//
#include <iostream>
#include <chrono>
#include <thread>
#include <vector>
#include <cstdint>
uint64_t blocks = 1024;
uint64_t threads = 1024;
uint64_t nloop = 10000000;
uint64_t npts = blocks*threads;
std::vector<double> results;
__global__ void init(double *arr,uint64_t nloop,uint64_t npts,int index) {
   uint64_t i = blockIdx.x*blockDim.x+threadIdx.x;
   uint64_t start = nloop*i+npts*nloop*index+1;
   uint64_t end = nloop*(i+1)+npts*nloop*index+1;
   arr[i] = 0;
   for (uint64_t j = start; j < end; ++j)
      arr[i] += 0.5/((j-0.75)*(j-0.25));
   }
__global__ void reduce_sum(double *arr,uint64_t len) {
   uint64_t i = blockIdx.x*blockDim.x+threadIdx.x;
   if (i < len)
      arr[i] += arr[i+len];
   }
void reduce(double *arr) {
   uint64_t len = npts >> 1;
   while (1) {
      reduce_sum<<<blocks,threads>>>(arr,len);
      len = len >> 1;
      if (len == 0)
         return;
      }
   }
void sum(int index) {
   cudaSetDevice(index);
   double harr[1],*darr;
   cudaMalloc(&darr,npts*sizeof(double));
   init<<<blocks,threads>>>(darr,nloop,npts,index);
   reduce(darr);
   cudaDeviceSynchronize();
   cudaMemcpy(harr,darr,8,cudaMemcpyDeviceToHost);
   results[index] = harr[0];
   cudaFree(darr);
   }
int main(void) {
   int ngpus;
   cudaGetDeviceCount(&ngpus);
   std::thread threads[ngpus];
   double pi = 0;
   auto tstart = std::chrono::high_resolution_clock::now();
   for (int i = 0; i < ngpus; ++i) {
      results.push_back(0);
      threads[i] = std::thread(sum,i);
      }
   for (int i = 0; i < ngpus; ++i) {
      threads[i].join();
      pi += results[i];
      }
   auto tend = std::chrono::high_resolution_clock::now();
auto dt = std::chrono::duration_cast<std::chrono::microseconds>(tend-tstart).count();
   auto gflops = npts*nloop*ngpus*5.0/dt/1e3;
   std::cout << "npts: " << npts << " nloop: " << nloop << " ngpus: " << ngpus << " pi: " << pi << '\n';
   std::cout << "time: " << 1e-6*dt << " estimated GFlops: " << gflops << '\n';
   return 0;
   }
```

8 A100 16,239 GFlop
OpenMP https://www.openmp.org
SYCL https://www.khronos.org/sycl/
WebGPU https://www.w3.org/TR/webgpu/
CPU, GPU, TPU
JAX https://github.com/google/jax
CuPy https://cupy.dev/

Taichi https://github.com/taichi-dev/taichi
example
MPI https://www.open-mpi.org/

```
//
// cudampipi.cu
// Neil Gershenfeld 10/14/20
// calculation of pi by a CUDA+MPI sum
// assumes one GPU per MPI rank
// pi = 3.14159265358979323846
//
#include <iostream>
#include <cstdint>
#include <mpi.h>
//
using namespace std;
//
uint64_t blocks = 1024;
uint64_t threads = 1024;
uint64_t npts = 1000000;
uint64_t nthreads = blocks*threads;
int nloop = 10;
//
__global__ void init(double *arr,uint64_t npts,uint64_t nthreads,int rank) {
   uint64_t index = blockIdx.x*blockDim.x+threadIdx.x;
   uint64_t start = npts*index+npts*nthreads*rank+1;
   uint64_t end = npts*(index+1)+npts*nthreads*rank+1;
   arr[index] = 0;
   for (uint64_t j = start; j < end; ++j)
      arr[index] += 0.5/((j-0.75)*(j-0.25));
   }
//
__global__ void reduce_sum(double *arr,uint64_t len) {
   uint64_t index = blockIdx.x*blockDim.x+threadIdx.x;
   if (index < len)
      arr[index] += arr[index+len];
   }
//
void reduce(double *arr) {
   uint64_t len = nthreads >> 1;
   while (1) {
      reduce_sum<<<blocks,threads>>>(arr,len);
      len = len >> 1;
      if (len == 0)
         return;
      }
   }
```

```
//
int main(int argc, char** argv) {
   int rank,nranks;
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   MPI_Comm_size(MPI_COMM_WORLD,&nranks);
   double *arr,result,pi;
   cudaMalloc(&arr,nthreads*sizeof(double));
   if (rank == 0) {
      for (int i = 0; i < nloop; ++i) {
         MPI_Barrier(MPI_COMM_WORLD);
      double tstart = MPI_Wtime();
         init<<<blocks,threads>>>(arr,npts,nthreads,rank);
         reduce(arr);
         cudaDeviceSynchronize();
         cudaMemcpy(&result,arr,8,cudaMemcpyDeviceToHost);
         MPI_Reduce(&result,&pi,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
      double tend = MPI_Wtime();
         double dt = tend-tstart;
         double gflops = npts*nthreads*nranks*5.0/dt/1e9;
         printf("npts = %ld, nthreads = %ld, nranks = %d, pi = %lf\n",npts,nthreads,nranks,pi);
         printf("time = %f, estimated GFlops = %f\n",dt,gflops);
         }
      }
   else {
      for (int i = 0; i < nloop; ++i) {
         MPI_Barrier(MPI_COMM_WORLD);
         init<<<blocks,threads>>>(arr,npts,nthreads,rank);
         reduce(arr);
         cudaDeviceSynchronize();
         cudaMemcpy(&result,arr,8,cudaMemcpyDeviceToHost);
         MPI_Reduce(&result,&pi,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
         }
      }
   cudaFree(arr);
   MPI_Finalize();
   return 0;
   }
```

2048 nodes, 12228 GPUs (Summit) 17,340,800 GFlop

FPGA

Verilog https://www.verilog.com/

VHDL https://ieeexplore.ieee.org/document/5981354

A1.3

libraries
- interactive, programming
- graphing
- input, output
- LINPACK https://netlib.org/linpack/
- LAPACK https://netlib.org/lapack/
- BLAS https://netlib.org/blas/
- Numerical Recipes http://numerical.recipes/
- MATLAB https://www.mathworks.com
- scripting history
- SciPy https://scipy.org/
- Matplotlib https://matplotlib.org/

```
#
# line.py
# Neil Gershenfeld  2/3/11
# line plot example
#

import matplotlib.pyplot as plt
from numpy import *

l = 15.5
x = arange(-l,l,.2)
y = sin(x)/x
plt.plot(x,y)
plt.show()
```

animate

```
#
# lines.py
# Neil Gershenfeld 2/4/17
# animated line plot example
#

import matplotlib.pyplot as plt
from numpy import *
import time

l = 15.5
x = arange(-l,l,.2)
y = sin(x)/x
plt.ion()
```

```
fig,ax = plt.subplots()
line, = ax.plot(x,y)
plt.show()
nloop = 25
loop = 1
step = 1
while 1:
   time.sleep(0.01)
   y = sin(0.1*loop*x)/(0.1*loop*x)
   line.set_ydata(y)
   fig.canvas.draw()
   loop += step
   if ((loop == 1) | (loop == nloop)):
      step = -step
```

2D

```
#
# image.py
# Neil Gershenfeld  2/3/11
# image plot example
#

import matplotlib.pyplot as plt
from matplotlib import cm
from numpy import *

l = 15.5
x = arange(-l,l,0.2)
y = arange(-l,l,0.2)
(x,y) = meshgrid(x,y)
r = sqrt(x**2+y**2)
z = sin(r)/r
plt.imshow(z,extent=[-l,l,-l,l],cmap=cm.gray)
plt.show()
```

3D

```
#
# surface.py
# Neil Gershenfeld  2/3/11
# surface plot example
#
from pylab import *
from numpy import *
from mpl_toolkits import mplot3d
l = 15.5
x = arange(-l,l,0.2)
y = arange(-l,l,0.2)
(x,y) = meshgrid(x,y)
r = sqrt(x**2+y**2)
z = sin(r)/r
fig = plt.figure(figsize =(8,8))
ax = plt.axes(projection ='3d')
ax.plot_surface(x,y,z)
plt.show()
```

Web
MathML https://www.w3.org/Math/
notebooks
Jupyter https://jupyter.org/
Colab https://colab.research.google.com/
Plotly https://plotly.com/

```
<html>
<head>
<meta charset="utf-8"/>
<script src="plotly-latest.min.js"></script>
</head>
<body>
<div id='plot'>
<script>
var l = 10.5
var step = .1
var data = {x:[],y:[],type:'scatter'};
for (var x = -l; x <= l; x+= step) {
   data.x.push(x)
   data.y.push(Math.sin(x)/x)
   }
Plotly.newPlot('plot',[data]);
var rmin = 1
var rmax = 5
var dr = .1
var r = rmin
function update() {
   r += dr
   if ((r > rmax) || (r < rmin))
      dr = -dr
   for (var i = 0; i < data.x.length; ++i) {
      var d = data.x[i]*r
      data.y[i] = Math.sin(d)/d
      }
   Plotly.redraw('plot')
   setTimeout(update,10)
   }
setTimeout(update,10)
</script>
</div>
</body>
</html>
```

D3 https://d3js.org/
R https://www.r-project.org/
ML
Tensorflow https://www.tensorflow.org/
PyTorch https://pytorch.org/
Flax https://github.com/google/flax

A1.4 GRAPHICS

interactive overhead
canvas https://www.w3.org/html/wiki/Elements/canvas

```
<html>
<head>
<title>canvas line test</title>
</head>
<body>
<div id="div" style="position:absolute;top:0%;height:100%;left:0%;width:100%;text-align:center;">
<canvas id="canvas"></canvas>
</div>
<script type="text/javascript">
//
// canvasline.html
// Neil Gershenfeld 1/28/17
// demonstrates drawing canvas lines by animating sin(k*x)/k*x
//
var div = document.getElementById("div")
var height = div.clientHeight
var width = height
var canvas = document.getElementById("canvas")
canvas.width = width
canvas.height = height
var npts = 500
var nloop = 100
var loop = 1
var step = 1
var linewidth = 0.005
var dtms = 10
var ctx = canvas.getContext("2d")
ctx.lineWidth = linewidth*width
window.setInterval("animate()",dtms);
function animate() {
   var x,y,r
   ctx.clearRect(0,0,width,height)
   ctx.beginPath()
   point = 0
   r = 100*(2*point-npts)/(npts*nloop)
   x = width*point/npts
   y = height*(1-Math.sin(loop*r)/(loop*r))/2
   ctx.moveTo(x,y)
   for (var point = 1; point < npts; ++point) {
      r = 100*(2*point-npts)/(npts*nloop)
      x = width*point/npts
      y = height*(1-Math.sin(loop*r)/(loop*r))/2
      ctx.lineTo(x,y)
      }
   ctx.stroke()
   loop += step
   if ((loop == 1) || (loop == nloop))
      step = -step
   }
</script>
</body>
</html>
```

SVG https://www.w3.org/Graphics/SVG/

```
<html>
<head>
<title>svg line test</title>
</head>
<body>
<div id="div" style="position:absolute;top:0%;height:100%;left:0%;width:100%;text-align:center;">
<svg id="svg" xmlns="http://www.w3.org/2000/svg" version="1.1"
   viewBox="0 0 1 1">
</svg>
</div>
<script type="application/javascript">
//
// svgline.html
// Neil Gershenfeld 1/28/17
// demonstrates drawing SVG lines by animating sin(k*x)/k*x
//
var div = document.getElementById("div")
var height = div.clientHeight
var width = height
var svgns = "http://www.w3.org/2000/svg"
svg = document.getElementById("svg")
svg.setAttribute("width",width)
svg.setAttribute("height",height)
var npts = 1000
var nloop = 100
var loop = 1
var step = 1
var linewidth = 0.005
var dtms = 10
window.setInterval("animate()",dtms)
line = document.createElementNS(svgns,"polyline")
line.setAttribute("stroke-width",linewidth)
line.setAttribute("stroke","black")
line.setAttribute("fill","none")
svg.appendChild(line)
function animate() {
   var x1,y1,x2,y2,r1,r2
   var points = ""
   for (var point= 0; point < npts; ++point) {
      r1 = 100*(2*point-(npts+1))/(npts*nloop)
      x1 = point/npts
      y1 = (1-Math.sin(loop*r1)/(loop*r1))/2
      r2 = 100*(2*(point+1)-(npts+1))/(npts*nloop)
      x2 = (point+1)/npts
      y2 = (1-Math.sin(loop*r2)/(loop*r2))/2
      points += x1+','+y1+' '+x2+','+y2+' '
      }
   line.setAttribute("points",points)
   loop += step
   if ((loop == 1) || (loop == nloop))
      step = -step
   }
</script>
</body>
</html>
```

https://threejs.org/

```
<html>
```

```
<head>
<title>three.js surface test</title>
<script src="three.min.js"></script>
</head>
<body>
<div id="div" style="position:absolute;top:0%;height:100%;left:0%;width:100%;text-align:center;">
</div>
<script type="text/javascript">
//
// threejssurf.html
// Neil Gershenfeld 2/3/17
// demonstrates drawing three.js surfaces by animating sin(k*x)/k*x
//
var div = document.getElementById("div")
var height = div.clientHeight
var width = height
var npts = 100
var nloop = 100
var loop = 1
var step = 1
var scene = new THREE.Scene()
var camera = new THREE.PerspectiveCamera(45,
   window.innerWidth/window.innerHeight,0.1,1000)
camera.position.set(0,-2.75,1.5);
camera.rotation.x = 65*(Math.PI/180)
var geometry = new THREE.PlaneGeometry(1,1,npts,npts)
for (var y = 0; y <= npts; ++y)
   for (var x = 0; x <= npts; ++x) {
      geometry.vertices[x*(npts+1)+y].x = 2*(x-npts/2)/npts
      geometry.vertices[x*(npts+1)+y].y = 2*(y-npts/2)/npts
      }
var material = new THREE.MeshPhongMaterial({color:0xc5b358})
var mesh = new THREE.Mesh(geometry,material)
scene.add(mesh)
var ambientLight = new THREE.AmbientLight(0xffffff)
scene.add(ambientLight)
light = new THREE.SpotLight(0xffffff,0.3)
light.position.set(0,-10,1)
scene.add(light)
var renderer = new THREE.WebGLRenderer({antialias:true})
renderer.setClearColor(new THREE.Color(0xffffff))
renderer.setSize(width,height)
document.body.appendChild(renderer.domElement)
animate()
function animate() {
   var r,z
   for (var y = 0; y <= npts; ++y) {
      for (var x = 0; x <= npts; ++x) {
         r = 30*Math.sqrt((0.5+y-npts/2)*(0.5+y-npts/2)+
            (0.5+x-npts/2)*(0.5+x-npts/2))/(nloop*npts/2)
         z = (1+Math.sin(loop*r)/(loop*r))/2
         mesh.geometry.vertices[x*(npts+1)+y].z = z
         }
      }
   loop += step
   if ((loop == 1) || (loop == nloop))
      step = -step
   mesh.rotation.z += 1/1000
   mesh.geometry.computeFaceNormals()
   mesh.geometry.computeVertexNormals()
   mesh.geometry.verticesNeedUpdate = true
   renderer.render(scene,camera);
   requestAnimationFrame(animate)
}
</script>
</body>
</html>
```

WebGL https://www.khronos.org/webgl/
OpenGL https://www.opengl.org/
fixed, programmable pipelines

```
/*
* glsurf.c
* (c) Neil Gershenfeld  2/4/03
* example of GL and GLUT, drawing sin(kr)/kr surface
*/

#include <GL/glut.h>
#include <math.h>
#include <unistd.h>

void normal(GLfloat,GLfloat,GLfloat,GLfloat,GLfloat,GLfloat,
GLfloat,GLfloat,GLfloat,GLfloat*,GLfloat*,GLfloat*);

#define NGRID 100
#define KMIN 0.0
#define KMAX 20.0
float k=0.0, dk=0.2;

#define r(x,y) (k*sqrt(x*x+y*y))
#define height(x,y) (sin(r(x,y))/r(x,y))
```

```
GLfloat x[NGRID][NGRID],y[NGRID][NGRID],z[NGRID][NGRID];

void display(void) {
   int i,j;
   GLfloat nx,ny,nz;
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glBegin(GL_QUADS);
   for (i = 0; i < (NGRID-1); ++i) {
    for (j = 0; j < (NGRID-1); ++j) {
         normal(x[i][j],y[i][j],z[i][j],
    x[i+1][j],y[i+1][j],z[i+1][j],
x[i][j+1],y[i][j+1],z[i][j+1],
&nx,&ny,&nz);
         glNormal3f(nx,ny,nz);
 glVertex3f(x[i][j],y[i][j],z[i][j]);
 glVertex3f(x[i+1][j],y[i+1][j],z[i+1][j]);
 glVertex3f(x[i+1][j+1],y[i+1][j+1],z[i+1][j+1]);
 glVertex3f(x[i][j+1],y[i][j+1],z[i][j+1]);
}
}
  glEnd();
  glFlush();
  }

void idle(void) {
GLfloat rx,ry,rz;
int i,j;
if ((k > KMAX) | (k < KMIN))
dk = -dk;
k += dk;
   for (i = 0; i < NGRID; ++i)
  for (j = 0; j < NGRID; ++j) {
     x[i][j] = 2.0*((float) j + 0.5)/NGRID - 1.0;
         y[i][j] = 2.0*((float) i + 0.5)/NGRID - 1.0;
 z[i][j] = height(x[i][j],y[i][j]);
  }
glMatrixMode(GL_MODELVIEW);
rx = rand();
ry = rand();
rz = rand();
glRotatef(1.0,rz,ry,rz);
  glutSwapBuffers();
glutPostRedisplay();
usleep(100);
}

void mouse(int button, int state, int x, int y) {
```

```
exit(0);
}

void normal(GLfloat x1, GLfloat y1, GLfloat z1,
GLfloat x2, GLfloat y2, GLfloat z2,
GLfloat x3, GLfloat y3, GLfloat z3,
GLfloat *xn, GLfloat *yn, GLfloat *zn) {
   *xn = (y2-y1)*(z3-z1) - (z2-z1)*(y3-y1);
   *yn = (z2-z1)*(x3-x1) - (x2-x1)*(z3-z1);
   *zn = (x2-x1)*(y3-y1) - (y2-y1)*(x3-x1);
}

main(int argc, char **argv) {
   GLfloat matl_ambient[] = {.25, .22, .06, 1.0};
   GLfloat matl_diffuse[] = {.35, .31, .09, 1.0};
   GLfloat matl_specular[] = {.80, .72, .21, 1.0};
   GLfloat light_ambient[] = {0.9, 0.9, 0.9, 1.0};
   GLfloat light_diffuse[] = {0.8, 0.8, 0.8, 1.0};
   GLfloat light_specular[] = {1.0, 1.0, 1.0, 1.0};
   GLfloat light_position[] = {0,0,1, 1.0};

   glutInit(&argc,argv);
   glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
   glutInitWindowSize(500, 500);
   glutCreateWindow("GLUT sin(kr)/kr example");
   glutDisplayFunc(display);
   glutMouseFunc(mouse);
   glutIdleFunc(idle);
   glMaterialfv(GL_FRONT_AND_BACK,GL_AMBIENT,matl_ambient);
   glMaterialfv(GL_FRONT_AND_BACK,GL_DIFFUSE,matl_diffuse);
   glMaterialfv(GL_FRONT_AND_BACK,GL_SPECULAR,matl_specular);
   glMaterialf(GL_FRONT_AND_BACK,GL_SHININESS, 95.0);
   glEnable(GL_LIGHTING);
   glLightModelf(GL_LIGHT_MODEL_TWO_SIDE, 1.0);
   glLightModelfv(GL_LIGHT_MODEL_AMBIENT,light_ambient);
   glLightfv(GL_LIGHT0,GL_AMBIENT,light_ambient);
   glLightfv(GL_LIGHT0,GL_DIFFUSE,light_diffuse);
   glLightfv(GL_LIGHT0,GL_SPECULAR,light_specular);
   glLightfv(GL_LIGHT0,GL_POSITION,light_position);
   glEnable(GL_LIGHT0);
   glEnable(GL_DEPTH_TEST);
   glEnable(GL_NORMALIZE);
   glClearColor(1.0,1.0,1.0,1.0);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   glOrtho(-1.5,1.5,-1.5,1.5,-1.5,1.5);
   glutMainLoop();
```

```
}
```

ModernGL https://github.com/moderngl
WebGPU https://www.w3.org/TR/webgpu/
pygfx https://github.com/pygfx/pygfx

```
#
# pygfxsurf.py
# Neil Gershenfeld 2/18/23
# sin(r)/r surface plot example
#
import pygfx as gfx
import numpy as np
#
l = 15.5
x = np.arange(-l,l,0.2)
y = np.arange(-l,l,0.2)
N = x.size
(x,y) = np.meshgrid(x,y)
r = np.sqrt(x**2+y**2)
r = np.reshape(r,N*N)
#
d = 100
scene = gfx.Scene()
surface = gfx.Mesh(
   gfx.plane_geometry(2*d,2*d,N-1,N-1),
   gfx.MeshStandardMaterial(color="#DDBB22",roughness=0.5,metalness=0.5,flat_shading=True),)
surface.rotation.multiply(
   gfx.linalg.Quaternion().set_from_euler(
   gfx.linalg.Euler(-np.pi/2,0,0)))
scene.add(surface)
scene.add(gfx.AmbientLight(intensity=1))
light = gfx.objects.PointLight()
light.position.set(d,d,d)
scene.add(light)
#
k = 0
dk = 0.01
kmax = 1
dt = 0.01
def animate():
   global k,dk
   k += dk
   if ((k > kmax) | (k < 0)):
      dk = -dk
   surface.rotation.multiply(
      gfx.linalg.Quaternion().set_from_euler(
      gfx.linalg.Euler(0,0,dt)))
   surface.geometry.positions.data[:,2] = d*np.sin(k*r)/(k*r)
   surface.geometry.positions.update_range(0,N*N)
#
gfx.show(scene,before_render=animate)
```

Taichi https://github.com/taichi-dev/taichi

A1.5 SYMBOLIC MATH

Maxima https://maxima.sourceforge.io/
Mathematica https://www.wolfram.com/mathematica/
Sage https://doc.sagemath.org/
SymPy https://www.sympy.org/en/index.html

```
#
# sympi.py
# Neil Gershenfeld 2/5/23
# pi symbolic calculation
#
from sympy import *
x,i= symbols('x i')
f = 4*atan(x)
print('Taylor series terms of 4*atan(x) at x=1:')
for n in range(1,12):
   print(pow(x,n-1).subs(x,1)*f.subs(x,0)/factorial(n-1),' ',end='')
   f = diff(f,x)
print('\nPartial fraction combination of pairs of Taylor series terms:')
f = 4/(4*i-3)-4/(4*i-1)
print(together(f))
```

```
Taylor series terms of 4*atan(x) at x=1:
0  4  0  -4/3  0  4/5  0  -4/7  0  4/9  0
Partial fraction combination of pairs of Taylor series terms:
8/((4*i - 3)*(4*i - 1))
```

JAX https://github.com/google/jax
automatic differentiation

A1.6 *ICATION

sonification

HTML5 audio

....

```
<html>
<body>
<button id="button">play sound</button>
<br>
<canvas id="canvas"></canvas>
<script>
//
// audioline.html
// Neil Gershenfeld 2/5/17
// demonstrates playing sin(x)/x as a sound
//
var audioCtx = new (window.AudioContext || window.webkitAudioContext)()
var channels = 1
var channel = 0
var duration = 2
var ramp = 0.1*audioCtx.sampleRate
var frequency = 2*Math.PI*100
var frames = duration*audioCtx.sampleRate
var myArrayBuffer = audioCtx.createBuffer(
   channels,frames,audioCtx.sampleRate)
var button = document.getElementById('button')
button.addEventListener('click',function() {
   var buffer = myArrayBuffer.getChannelData(channel)
   for (var i = 0; i < frames; i++) {
      var time = (i+0.5-frames/2)/audioCtx.sampleRate
      buffer[i] = (1-Math.sin(frequency*time)/(25*time))/2
      }
   for (var i = 0; i < ramp; i++)
      buffer[i] = buffer[i]*i/(ramp-1)
   for (var i = (frames-ramp); i < frames; i++)
      buffer[i] = buffer[i]*(frames-i-1)/(ramp-1)
   var source = audioCtx.createBufferSource()
   source.buffer = myArrayBuffer
   source.connect(audioCtx.destination)
   source.start()
   })
var height = document.body.clientHeight
var width = height
var canvas = document.getElementById("canvas")
canvas.width = width
canvas.height = height
var linewidth = 0.001
var ctx = canvas.getContext("2d")
ctx.lineWidth = linewidth*width
ctx.clearRect(0,0,width,height)
ctx.beginPath()
var i = 0
var time = (i+0.5-frames/2)/audioCtx.sampleRate
x = width*i/frames
y = height*(1-Math.sin(frequency*time)/(0.1*frequency*time))/2
ctx.moveTo(x,y)
for (var i = 1; i < frames; i++) {
   var time = (i+0.5-frames/2)/audioCtx.sampleRate
   x = width*i/frames
   y = height*(1-Math.sin(frequency*time)/(25*time))/2
   ctx.lineTo(x,y)
   }
ctx.stroke()
</script>
</body>
</html>
```

fabrication
STL

A1.7 MACHINE LEARNING

XOR
define model, loss, optimizer, backprop
TensorFlow https://www.tensorflow.org/

```
#
# xortf.py
# Neil Gershenfeld 4/27/23
# XOR TensorFlow example
#
import tensorflow as tf
import numpy as np
#
input = np.array([[0, 0],[0, 1],[1, 0],[1, 1]],dtype=np.float32)
output = np.array([[0],[1],[1],[0]],dtype=np.float32)
#
model = tf.keras.models.Sequential()
model.add(tf.keras.Input(shape=(2,)))
model.add(tf.keras.layers.Dense(2,
   activation=tf.keras.activations.tanh,
   kernel_initializer=tf.initializers.RandomNormal))
```

```
model.add(tf.keras.layers.Dense(1,
   kernel_initializer=tf.initializers.RandomNormal))
#
model.compile(
   optimizer=tf.keras.optimizers.Adam(learning_rate=0.01),
   loss=tf.keras.losses.MeanSquaredError())
#
model.fit(input,output,epochs=1000,verbose=1)
#
predictions = model.predict_on_batch(input)
print('predictions:\n',predictions)
```

output

```
Epoch 1/1000
1/1 [==============================] - 0s 331ms/step - loss: 0.4976
Epoch 2/1000
1/1 [==============================] - 0s 12ms/step - loss: 0.4847
Epoch 3/1000
1/1 [==============================] - 0s 4ms/step - loss: 0.4718
...
Epoch 998/1000
1/1 [==============================] - 0s 3ms/step - loss: 4.1078e-15
Epoch 999/1000
1/1 [==============================] - 0s 3ms/step - loss: 4.1078e-15
Epoch 1000/1000
1/1 [==============================] - 0s 3ms/step - loss: 5.5511e-16
predictions:
 [[4.4703484e-08]
 [1.0000000e+00]
 [1.0000001e+00]
 [1.0430813e-07]]
```

PyTorch https://pytorch.org/

```
#
# xorpt.py
# Neil Gershenfeld 4/27/23
# XOR PyTorch example
#
import torch
import torch.nn as nn
#
x = torch.tensor([[0, 0],[0, 1],[1, 0],[1, 1]],dtype=torch.float)
y = torch.tensor([0, 1, 1, 0],dtype=torch.float).view(-1, 1)
dataset = torch.utils.data.TensorDataset(x,y)
dataloader = torch.utils.data.DataLoader(dataset,batch_size=16)
```

```
#
model = torch.nn.Sequential(
   nn.Linear(2,2),
   nn.Tanh(),
   nn.Linear(2,1),
   )
#
def init(layer):
   if type(layer) == nn.Linear:
      nn.init.normal_(layer.weight)
model.apply(init)
#
loss = nn.MSELoss(reduction='sum')
optimizer = torch.optim.Adam(model.parameters(),lr=0.01)
#
for epoch in range(1000):
   for x,y in dataloader:
      ypred = model(x)
      error = loss(ypred,y)
      if epoch % 100 == 0:
          print(epoch,error.item())
      optimizer.zero_grad()
      error.backward()
      optimizer.step()
#
ypred = model(x)
print(ypred.detach().numpy())
```

output

```
0 1.8111252784729004
100 0.6191580891609192
200 0.1057029515504837
300 0.0007048047264106572
400 6.663513829607837e-08
500 3.538502824085299e-12
600 2.0037305148434825e-12
700 1.4068746168049984e-12
800 1.5063505998114124e-12
900 1.0516032489249483e-12
[[-3.5762787e-07]
 [ 1.0000005e+00]
 [ 1.0000006e+00]
 [-3.5762787e-07]]
```

A1.8 PROBLEMS

(A1.1) Visualize bouncing balls using as many different programming languages and graphical environments as you can.

(A1.2) Use your bouncing ball simulation to model and characterize a Maxwell Demon [Leff & Rex, 1990]

Appendix 2 Benchmarking

Benchmarking is a subject that receives both too little and too much attention. Too little, because knowing the relative speeds of machines, languages, and algorithms can have an enormous impact on your ability to obtain timely results. Too much, because tests that may have little bearing on practical problems can dominate manufacturers' advertising and users' purchase decisions.

Amid all of the hype, a simple recurring truth is that the best benchmark is a problem that you are interested in. An early standard was the *Linpack* set of subroutines, which have been run on an enormous range of machines. Results for the *TOP500* systems are listed at `https://www.top500.org`, and since the largest ones use as much power as a whole city the *Green500* list (`https://www.top500.org/lists/green500`) tracks their relative efficiency.

Because there is a great deal of specialized structure in these routines, aggressive compilers used switches that recognized them and used carefully hand-tuned code to appear faster on this benchmark. To prevent that, as well as to cover a much broader range of applications, an industry-wide group defined a suite of test problems called the *SPEC* benchmark (`https://www.spec.org`). This is a comprehensive set of programs covering many types of numerical algorithms. Where it's available, it's a reliable guide to machine speed. However, the suite may not be available for a particular system that you're interested in, and it is not freely accessible. For this reason it's useful to have a simple test program that can provide a rough order-of-magnitude estimate of speed.

I've found it convenient to use a series expansion of π,

$$\begin{aligned}\pi &= 4\tan^{-1}(1)\\ &\approx \sum_{i=1}^{N} \frac{0.5}{(i-0.75)(i-0.25)} \quad .\end{aligned}$$

Summing this series requires five floating-point operations per step (ignoring the overhead for iterating the loop), providing an estimate of the computational speed by measuring the time taken to sum it. This is usually reported in the number floating-point operations per second, called *flops*.

This is a classic parallel computation, a *scatter-gather* that can distribute terms in the sum to multiple processors and then collect the result. In particular, it is a type of *map-reduce*, in which each node calculates a value based on its index, and these are then centrally summed. There are more efficient ways to calculate pi [Bailey *et al.*, 1996], and because this is limited to the precision of the numbers used summing beyond

Table A2.1. *Selected execution speeds to sum a series expansion of π.*

speed (Gflops)	*system*	*version*
17,340,800	IBM AC922 (Summit)	C++, MPI, CUDA, 2048 nodes, 12228 GPUs
88,333	Cray XC40 (Theta)	C, MPI, OpenMP, 1024 nodes, 64 cores/node
16,239	NVIDIA A100	C++, CUDA, 8 GPUs
2,117	Intel 8175M	C, MPI, 10 nodes, 96 cores/node
2,102	Intel 8175M	Python, Numba, MPI, 10 nodes, 96 cores/node
2,052	NVIDIA A100	C++, CUDA, 6192 cores
1,595	IBM Blue Gene/P	C, MPI, 4096 processes
1,090	NVIDIA V100	Python, Numba, CUDA, 5120 cores
811	Cray XT4	C, MPI, 2048 processes
315	Intel 8175M	Python, Numba, 96 cores
267	Intel 8175M	C++, 96 threads
152	Intel 8175M	JavaScript, 96 workers
44.6	Intel i7-8700T	C, 6 threads
9.37	Intel i7-8700T	C, optimized
1.78	Raspberry Pi 4	C, 4 threads
0.85	Connection Machine CM-2	C, 32k processors
0.57	Intel i7-8700T	C, unoptimized
0.47	Intel i7-8700T	Python, NumPy
0.148	IBM ES/9000	C
0.134	Intel Pentium III	C
0.118	Cray Y-MP4	C, vector
0.074	Raspberry Pi Zero	C
0.029	Intel i7-8700T	Python
0.017	SAMD51J20A	C
0.013	Intel Pentium Pro	C
0.010	Cray Y-MP4	C, scalar
0.003	RP2040	Arduino
0.001	Sun SPARCStation 1	C
0.001	DEC VAX 8650	C
0.0007	Intel 486	C
0.0003	RP2040	MicroPython
0.0001	ATtiny1614	Arduino
0.00003	Sun 3/60	C
0.00003	Intel 286	C
0.000001	Intel 8088	C

that is wasted in round-off, but the calculation is simple to port to new systems, and the validity of the result is easy to check. Table A2.1 shows some sample speeds for machines, languages, and options. It is NOT in any way a thorough characterization of these systems, but it is an easily-generated estimate that is typically surprisingly close to much more careful benchmarks.

The single most remarkable feature of this table is that it spans thirteen orders of magnitude! That's the difference between an algorithm taking the duration of recorded history and a fraction of a second. For some big problems, literally the fastest way to solve them was to wait for a faster computer to be developed.