Accuracy/Precision/General Error Concepts

CSS 455, Winter 2012

Scientific Computing

Errata: chapter 1 of turner

- P3, 2nd equation: last term should be $a_{-m}\beta^{-m}$ rather than $a_{-m}\beta^{m}$
- P3, Eq 1.3, last term should be $b_N\beta^{-N}$ rather than $b_N\beta^{N}$
- P11, Exercise 3. third term in cosh(x) should be $(x^4/4!)$ rather that $(x^4/41)$
- P282, Section 1.4, last answer (1.67618) is incorrect.

Precision

- **Precision** refers to the number of significant figures or to the repeatability of the measurement. *Precision may be improved by larger data sets.*
- (For example, 6.022×10^{23} is a more precise measurement of Avogadro's Number than is 6.02×10^{23}.)

Accuracy

- **Accuracy** is an indication of how close the measurement is to the true value. It may include systematic instrument error.
- For example, 6.0 x10^{23} is a more accurate value of Avogadro's Number than is 5.885646x10^{23}.

Computational Error

- Algorithmic Error:
  Truncation or discretization. Some terms may be omitted. For example, Taylor Series for a function:

$$f(x+\delta) = f(x) + f'(x)(\delta) + \frac{f''(x)}{2!}(\delta^2) + \frac{f'''(x)}{3!}(\delta^3) + \cdots$$

We may truncate after just a few terms for small δ

Computational Error

- Data representation
  Rounding. Computer representation of real numbers is generally inexact.
  - (The number 1/3 will be rounded to 0.3333333 in some floating point representations.)
- Error Propogation.
  Calculations are often done in steps. The later steps depend upon the results (and errors) of the earlier ones.

Error Analysis

- Absolute: approximate - true.
  - *Units are the same as the measured values.*
- Relative: Absolute error divided by true value.

$$\frac{approx - true}{true}$$

Unitless, expressed as fraction or percent.

Floating-Point Numbers

- Many scientific calculations are done with approximately 64 bits used for floating point representation: real*8, double precision (most workstations)
- The division of these bits between exponent and mantissa fields varies from machine to machine. The precision is about 14-16 *decimal* digits and the exponent range is about $10^{\pm 200}$ - $10^{\pm 500}$

Double Precision is default on Matlab. Single precision and integer representation must be selected.

```
DU>> format long e
EDU>> n=2/3
n =
   6.666666666666666e-001
```

Floating Point Representation

$$x = \pm b_0.b_1b_2b_3 \ldots b_N \times \beta^E \text{ , with } L \le E \le U$$

- β is the base (β=2 in binary system)
- b_0 is implicit digit (defined by convention) ($b_0=1$)
- $b_1b_2 \ldots b_N$ is the mantissa field of N-digits
- E is the exponent field

$$x = \pm\left\{b_0 + \frac{b_1}{\beta} + \frac{b_2}{\beta^2} + \frac{b_3}{\beta^3} + \ldots + \frac{b_N}{\beta^N}\right\}\beta^E$$

Normalized Binary Representation

$$(19)_{10} = 1(2^4) + 0(2^3) + 0(2^2) + 1(2^1) + 1(2^0) = 16 + 2 + 1$$

$$10011. = 1.0011 \times 2^4$$

Implicit bit

Normalized Binary Representation

$$\left(\frac{1}{5}\right)_{10} = 0\left(\tfrac{1}{2}\right) + 0\left(\tfrac{1}{2}\right)^2 + 1\left(\tfrac{1}{2}\right)^3 + 1\left(\tfrac{1}{2}\right)^4 + 0\left(\tfrac{1}{2}\right)^5 + 0\left(\tfrac{1}{2}\right)^6 + 1\left(\tfrac{1}{2}\right)^7 + 1\left(\tfrac{1}{2}\right)^8 + \ldots$$

$$\left(\frac{1}{5}\right)_{10} = \frac{1}{8} + \frac{1}{16} + \frac{1}{128} + \frac{1}{256} + \ldots = 0.00110011001100\ldots$$

$$\left(\frac{1}{5}\right)_{10} = 1.10011001100\ldots \times 2^{-3}$$

Implicit bit

Parameters for typical floating-point systems

System	β	N	L	U
IEEE SP	2	24	-126	127
IEEE DP	2	53	-1,022	1,023
Cray	2	48	16,383	16,384
HP Calc	10	12	-499	499

UFL, OFL

EDU>> realmin
ans =
 2.2251e-308

- UFL = β^{L} = *smallest number represented:*
  - *IEEE DP* 2^{-1022}= 2.2 x 10^{-308}
- OFL = $\beta^{U+1}(1-\beta^{-N})$= *largest floating pt number:*
  - *IEEE DP:* $2^{1024}(1-2^{-53}) = 1.8\times10^{+308}$.

EDU>> realmax
ans =
 1.7977e+308

ε_{mach}

- **With rounding by chop:**
  $\varepsilon_{mach}= \beta^{1-N}=2^{-52} \approx 10^{-16}$ = *max possible* *<u>relative</u> error in representing a number*
- **With round to the nearest:**
  $\varepsilon_{mach}= \frac{1}{2}\beta^{1-N}=2^{-53} \approx 10^{-16}$

EDU>> eps
ans =
 2.2204e-016

Notice...

- that UFL and OFL represent absolute magnitudes.
- that UFL's can be often set to zero.
- that ε_{mach} represents relative precision. (rounding to nearest decreases ε_{mach} *by 1/2 compared to chopping.)*

Floating Point Arithmetic Errors

- Rounding
  - Addition of numbers of different magnitudes will result in the sum being represented with roundoff.
    (For a 6 digit system)
    192.403 + 0.635782 = 193.039
  - If the small number is small enough, the total won't change at all:
    192.403 + $1.5\text{x}10^{-5}$ = 192.403

Floating Point Arithmetic Errors

- Rounding
  - It makes a difference which way the series is summed (not commutative). *(sum1overn demo Matlab)*

$$\sum_{n=1}^{\text{nlim}} \frac{1}{n} = 1 + \tfrac{1}{2} + \tfrac{1}{3} + \cdots + \tfrac{1}{\text{nlim}}$$

$$= \tfrac{1}{\text{nlim}} + \cdots + \tfrac{1}{3} + \tfrac{1}{2} + 1$$

Cancellation

- Subtraction can result in a loss of precision <u>even if all numbers are representable.</u> (this can be a very serious problem.)

  1.55456 - 1.55435 = 0.00021= $2.1\text{x}10^{-4}$
  (even with 6 digit representation, our result has <u>only 2 digits of precision.)</u>

demo showing order of subtraction (canc-err)

$$\mathbf{a} = \left(1\tfrac{1}{3} \quad 2 \quad 3 \quad 4 \quad \cdots\right)$$
$$\mathbf{b} = \left(-1 \quad -2 \quad -3 \quad -4 \quad \cdots\right)$$

Floating Point Arithmetic Errors

- Rounding
  - Algorithms matter! Compare $(x-1)^6$ with the expanded polynomial.

$$f(x) = (x-1)^6 = x^6 - 6x^5 + 15x^4 - 20x^3 + 15x^2 - 6x + 1$$

What are the roots of this equation?

That is, for what *x*-values does *f(x)* = 0?

Floating Point Arithmetic Errors

- Rounding
  - Algorithms matter! Compare $(x-1)^6$ with the expanded polynomial.

$$f(x) = (x-1)^6 = x^6 - 6x^5 + 15x^4 - 20x^3 + 15x^2 - 6x + 1$$

Matlab Zoomdemo.m, zoomdemocanc.m

Each subplot examines a region closer to the roots at x=1. Notice the difference between the two algorithms.

Group discussion: why is this happening?

Floating Point Arithmetic Errors

- Rounding
  - Algorithms matter! Compare $(x-1)^6$ with the expanded polynomial.

$$f(x) = (x-1)^6 = x^6 - 6x^5 + 15x^4 - 20x^3 + 15x^2 - 6x + 1$$

Matlab Zoomderivatives.m, zoomderivativescanc.m

What about using the derivatives instead? What does the derivative do at a root?

Activity 2

Stirling Approx to *n!*

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n$$

- There is a series expansion for $n!$:

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \tfrac{1}{12n} + \tfrac{1}{288n^2} + \cdots\right)$$

Stirling Approximation to $n!$: $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$

Truncation Error Stirling Approx

- *Run Matlab Stirlingdemo.m*
- Note that relative error is large, but becomes smaller as n increases. This error is due to *truncation* of the series.
- Addition of the second term (1+1/12n…) reduces relative error by about two orders of magnitude.
- *Return to Stirlingdemo.m*

Ex 3, p 15 of text

- Abs and rel error of representing 1/5 in a 12-bit mantissa binary system.

$$\left(\frac{1}{5}\right)_{10} = 1.100110011001\ldots \times 2^{-3}$$

Taylor Approx for e^x

- The exponential function can be expressed in terms of the infinite series:

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!}$$

What about an approximation formed from *n* terms?

$$e^x = \sum_{k=0}^{n-1} \frac{x^k}{k!}$$

How to code the exp function

- For each *x*, compute a series of terms for the summation:

$$e^x = \sum_{k=0}^{n-1} \frac{x^k}{k!}$$

Inefficient to calculate $(k!)$ and (x^k) "from scratch" for each k.

How can we get the k-th term from the (k-1)term?

Matlab ExpTaylor

- Look at the code. For each *x*, compute a series of terms corresponding to limits on the summation:

$$e^x = \sum_{k=0}^{n-1} \frac{x^k}{k!}$$

demo *exptaylordemo.m*

Matlab ExpTaylor

- Precision generally increases with number of terms. *(Why does it decrease at first for some negative values of x?)*
- There is a limit beyond which it does not increase. *(Why?)*
- Accuracy depends on the value of the argument.

Exercise 3, p 11 of text

- How many terms are needed to estimate cosh(1/2) with error less than 10^{-8}?

$$\cosh(x) = \sum_{k=0}^{\infty} \frac{x^{2k}}{(2k)!} = 1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \ldots$$

$$\cosh(x) = \sum_{k=0}^{N-1} \frac{x^{2k}}{(2k)!} + \sum_{k=N}^{\infty} \frac{x^{2k}}{(2k)!}$$

Conditioning and Sensitivity

Condition number is characteristic of the problem and not the algorithm:

$$Cond = \frac{\left|\left[f(\hat{x}) - f(x)\right]/\,f(x)\right|}{\left|(\hat{x}-x)/x\right|} \quad \{\hat{x} \text{ is near } x\}$$

Large condition number indicates that solution is highly sensitive to small changes in input data.

Consider values of cos(x) for x close to zero and close to $\pi/2$

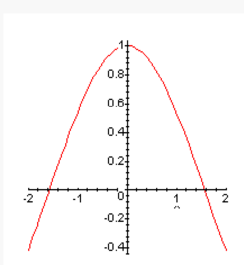
- *cos(1.5708) = -3.673205 x 10^{-6}*
  cos(1.5707) = 9.6326679 x 10^{-5}

$$\frac{(9.632x10^{-5} + 0.367x10^{-5})/0.367x10^{-5}}{(1.5708 - 1.5707)/1.5707} = \frac{27.2}{6.37x10^{-5}} = 4.3x10^{5}$$

- *cos(0.0000) = 1.000000000*
  cos(0.0001) = 0.999999995

$$\frac{(1.000000000 - 0.999999995)/0.999999995}{(0.0000 - 0.0001)/0.0001} = \frac{5x10^{-9}}{1.0} = 5x10^{-9}$$

- *Cos*(x) is conditioned much better at x=0 than at x = ± $\pi/2$



Following p.11

$$p(x) = (x-1)(x-2)\cdots(x-20) = 0$$

or

$$p(x) = x^{20} + a_{19}x^{19} + \cdots + a_1 x + a_0$$

Roots are known to be at x= 1, 2, 3, …, 20

Previous exercise revealed that 2nd formulation of algorithm can be unstable, because it is imprecise.

Assume stable, accurate algorithm

$$p(x) = x^{20} + a_{19}x^{19} + \cdots + a_1 x + a_0$$

A physical problem is to be modeled by this 20th order polynomial, with the coefficients to be fit to experimental data.

If a_{19} = -210, largest real root = 20, and all roots are real integers..

If a_{19} = -210 + 2^{-22} (-209.999999762), largest root is 20.85 and 10 of the roots are complex numbers!

In this case, the problem is ill-conditioned, even if the algorithm is stable.

Stability (or sensitivity) refers to algorithm

- A stable algorithm produces results that are relatively insensitive to perturbations made within the computation.
- Inaccuracy can arise from an unstable algorithm or from an ill-conditioned problem.
- Accuracy requires well conditioned problem **and** stable algorithm.

Quadratic Formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

What are some possible numerical problems with using this generally?

$$x = \frac{2c}{-b \mp \sqrt{b^2 - 4ac}}$$

Use one root from each formula.

$x_1 x_2 = c/a$

Error Metrics

supremum or L_∞

$$\|f - p\|_\infty = \max_{a \le x \le b} |f(x) - p(x)| \Rightarrow \max_{k=1\ldots N} |f(x_k) - p(x_k)|$$

L_1

$$\|f - p\|_1 = \int_a^b |f(x) - p(x)| dx \Rightarrow \sum_{k=0}^{N} |f(x_k) - p(x_k)|$$

L_2

$$\|f - p\|_2 = \sqrt{\int_a^b |f(x) - p(x)|^2 dx} \Rightarrow \sqrt{\sum_{k=0}^{N} |f(x_k) - p(x_k)|^2}$$

note similarity to vector norms

Nonlinear Equations and Root Finding

Chapter 2 of Turner

CSS455 Winter 2012

How would you find?

- A solution to the equation

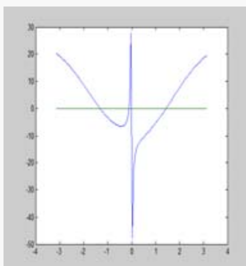
$$x^2 - \frac{1}{x} = 10 * \cos(x)$$

•Discuss with partner

•*see demo testfn.m*

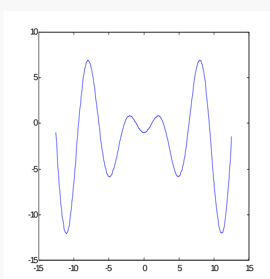
Plot it first!

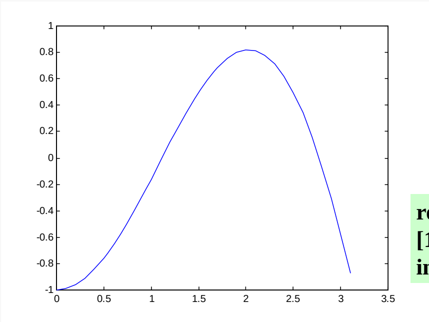
- Trial Equation
  f(x) = x^2 – (1/x) - 10*cos(x) =0
- Plot f(x) to learn of its general behavior.



Plot it first!

- Another Trial Equation
  xsin(x) = 1
- or
  f(x) = xsin(x) - 1 =0
- Plot f(x) to learn of its general behavior.





roots lie in [1,1.5] and in [2.5,3.5]

Bisection method

- Find an interval containing a root
- f(x) changes sign in the interval
- Reduce the interval until its size satisfies the convergence criterion for the root
- Following program is similar to one on pp 24-5 of Turner
- Function xsin(x)-1 is in the demfun1 m-file routine.
- Bisection method on this function is in *demobisect.m*

```
%demobisect.m
clear
% search between a and b
a = input ('enter lower limit')
b = input ('enter upper limit')
tol = 1.E-6;
while ((b-a)>tol)
   m = (a+b)/2;
   if (sign(demfun1(a))==sign(demfun1(m)))
      a = m;
   else
      b = m;
   end
   fprintf('\na= %f b= %f f(a)=%E
f(b)=%E',a,b,demfun1(a),demfun1(b))
end
```

Obvious refinements:
1. Only one function evaluation per iteration.
2. Lower bound on *tol* to make sure it is not set beneath machine accuracy.
3. Use *abs(a-b)*

Run *demobisect.m* with limits of 2.5 and 3.0

a= 2.500000 b= 3.000000 f(a)=4.961804E-001 f(b)=-5.766400E-001
a= 2.750000 b= 3.000000 f(a)=4.956773E-002 f(b)=-5.766400E-001
a= 2.750000 b= 2.875000 f(a)=4.956773E-002 f(b)=-2.425928E-001
a= 2.750000 b= 2.812500 f(a)=4.956773E-002 f(b)=-9.104357E-002
a= 2.750000 b= 2.781250 f(a)=4.956773E-002 f(b)=-1.934543E-002
a= 2.765625 b= 2.781250 f(a)=1.546218E-002 f(b)=-1.934543E-002
a= 2.765625 b= 2.773438 f(a)=1.546218E-002 f(b)=-1.854219E-003
a= 2.769531 b= 2.773438 f(a)=6.825878E-003 f(b)=-1.854219E-003
a= 2.771484 b= 2.773438 f(a)=2.491298E-003 f(b)=-1.854219E-003
a= 2.772461 b= 2.773438 f(a)=3.199059E-004 f(b)=-1.854219E-003
a= 2.772461 b= 2.772949 f(a)=3.199059E-004 f(b)=-7.668150E-004
a= 2.772461 b= 2.772705 f(a)=3.199059E-004 f(b)=-2.233692E-004
a= 2.772583 b= 2.772705 f(a)=4.828971E-005 f(b)=-2.233692E-004

a= 2.772583 b= 2.772644 f(a)=4.828971E-005 f(b)=-8.753439E-005
a= 2.772583 b= 2.772614 f(a)=4.828971E-005 f(b)=-1.962101E-005
a= 2.772598 b= 2.772614 f(a)=1.433469E-005 f(b)=-1.962101E-005
a= 2.772598 b= 2.772606 f(a)=1.433469E-005 f(b)=-2.643078E-006
a= 2.772602 b= 2.772606 f(a)=5.845825E-006 f(b)=-2.643078E-006
a= 2.772604 b= 2.772606 f(a)=1.601379E-006 f(b)=-2.643078E-006
a= 2.772604 b= 2.772605 f(a)=1.601379E-006 f(b)=-5.208484E-007»

•This process is linearly convergent. It takes the same number of iterations to add *n* bits of precision regardless of the position within the sequence. *Why?*

•Requires only the value of the function

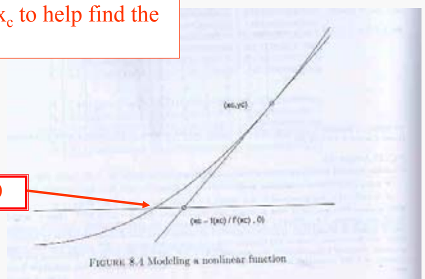
•Does not make use of magnitudes.

$$3x^3 - 5x^2 - 4x + 4 = 0$$

- Apply bisection method in [0,1] (#1, p 27)
- Plot it first! (*plotfn.m*)
- Use *demobisect2* to solve.

Newton's Method

•Can we use the knowledge of the slope at x_c to help find the root?



Root at f(x) = 0

FIGURE 8.4 Modeling a nonlinear function

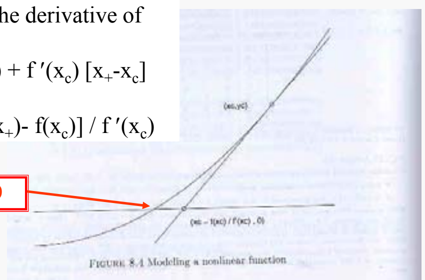
Newton's Method

•The slope of the straight line is given by the derivative of f(x) at x_c

•$f(x_+) = f(x_c) + f'(x_c)\,[x_+ - x_c]$

or

$x_+ = x_c + [f(x_+) - f(x_c)] / f'(x_c)$



Root at f(x) = 0

FIGURE 8.4 Modeling a nonlinear function

Newton's Method

•Alternatively, view as a Taylor's expansion about x_c

$$f(x_+) = f(x_c) + (x_+ - x_c) f'(x_c) + \frac{(x_+ - x_c)^2}{2} f''(x_c) + \ldots$$

$$f(x_+) \approx f(x_c) + (x_+ - x_c) f'(x_c)$$

$$\frac{f(x_+) - f(x_c)}{f'(x_c)} \approx (x_+ - x_c)$$

$$x_+ = x_c + \frac{f(x_+) - f(x_c)}{f'(x_c)}$$

Newton's Method

$f(x_+)=0$ at a root
$x_+ = x_c - f(x_c) / f'(x_c)$
In our case
$f(x) = x \sin(x) - 1$
$f'(x) = \sin(x) + x\cos(x)$

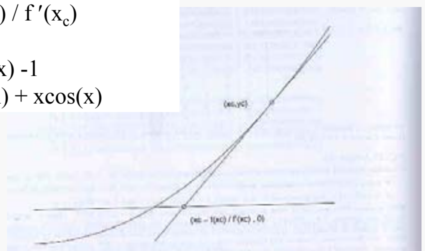


FIGURE 8.4 Modeling a nonlinear function

Newton's Method

Iterant: $x_{n+1} = x_n - f(x_n) / f'(x_n)$

- Start at x = 3.5
- Converges in only five iterations
- quadratic convergence: # of digits is doubled at each iteration.

```
xcurr= 2.886023  xprev = 3.500000
xcurr= 2.779536  xprev = 2.886023
xcurr= 2.772635  xprev = 2.779536
xcurr= 2.772605  xprev = 2.772635
xcurr= 2.772605  xprev = 2.772605»
```

Newton's Method

- Starting at x = 2.6, it converges to the same root

```
xcurr =
   2.6000
xcurr= 2.798728  xprev = 2.600000
xcurr= 2.773026  xprev = 2.798728
xcurr= 2.772605  xprev = 2.773026
xcurr= 2.772605  xprev = 2.772605»
```

- starting at x = 2 it converges to a very different root

```
xcurr= -8.630584  xprev = 2.000000
xcurr= -9.596971  xprev = -8.630584
xcurr= -9.322270  xprev = -9.596971
xcurr= -9.317247  xprev = -9.322270
xcurr= -9.317243  xprev = -9.317247
xcurr= -9.317243  xprev = -9.317243»
```

Newton's Method

- starting at x = 1.11, it converges to the nearby root.

•Convergence: Quadratic, but not guaranteed.

•Requires derivative.

- **xcurr =**
- **1.1100**
- **xcurr= 1.114156 xprev = 1.110000**
- **xcurr= 1.114157 xprev = 1.114156»**

$3x^3 - 5x^2 - 4x + 4 = 0$

- Apply Newton method near x = 0.7 (#1 on p. 41). All you need is the function (above) and the derivative. *demonewton2.m*

Systems of Nonlinear Equations

systems of nonlinear equations

- Consider Example 12 on p. 46
  - Two equations - two unknowns
  - Behavior depends strongly on exact detail of the equations
- Apply some of the same methods applied to the scalar nonlinear case:
  - e.g. Newton's method

Make plausible

- System of two equations (f,g) in two variables (x,y) similar to that of previous example

- Two Equations in two unknowns:

$$f_1(x_1, x_2) = 4x_1^2 + x_2^2 - 4 = 0$$
$$f_2(x_1, x_2) = x_1^2 x_2^3 - 1 = 0$$

In vector notation: **f**(**x**) =**0**

Newton's Method

- As in the scalar case, convergence will be faster if it is started close to root.
- In the scalar case, the derivative of the function as well as the function itself was required at each iteration.
- In the multidimensional case, the Jacobian of the function plays this role. (nontrivial evaluation.)

write in vector notation

- **x** is the vector of x_1 and x_2
- **f** is the vector of f_1 and f_2 evaluated with **x**
- $\mathbf{J_f}$ is the Jacobian of **f**

$$\mathbf{f}(\mathbf{x}) = \begin{bmatrix} 4x_1^2 + x_2^2 - 4 \\ x_1^2 x_2^3 - 1 \end{bmatrix} = 0$$

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

J is a square matrix, each element of which is a partial derivative of one of the equations in the set.

Partial derivatives: differentiate the function *f* with respect to one of its variables, treating the others as if they were constants.

$$f(x_1, x_2) = 4x_1^2 + x_2^2 - 4 = 0$$

$$\left(\partial f / \partial x_1\right) = 8x_1$$

$$\left(\partial f / \partial x_2\right) = 2x_2$$

$$\mathbf{J}(\mathbf{x})_{\mathbf{ij}} = \frac{\partial f_i(\mathbf{x})}{\partial x_j}$$

$$\mathbf{J} = \begin{pmatrix} \partial f_1 / \partial x_1 & \partial f_1 / \partial x_2 \\ \partial f_2 / \partial x_1 & \partial f_2 / \partial x_2 \end{pmatrix} = \begin{pmatrix} 8x_1 & 2x_2 \\ 2x_1 x_2^3 & 3x_1^2 x_2^2 \end{pmatrix}$$

What is the definition of a matrix inverse?

$\mathbf{x_{k+1}} = \mathbf{x_k} - \mathbf{J^{-1}f}$

(compare to scalar case)

$x_{n+1} = x_n - f(x_n) / f'(x_n)$

- In practice, we would avoid the inversion of **J**. But, here it will be done.
- The equations are written out explicitly on p.45-46 for the 2x2 case. *You need now!*
- The iterative formula then becomes $\mathbf{x_{k+1}} = \mathbf{x_k} + \mathbf{s_k}$, with
- $\mathbf{s_k} = -\mathbf{J^{-1}f}$
- the two components of **s** *are given by h and k in text)* $\mathbf{s} = \begin{pmatrix} h \\ k \end{pmatrix}$

See demosysnewton.m

2 x 2 system by Newton

$(x,y) \Leftrightarrow (x_1, x_2)$

```
enter x  .4
enter y   1.8
   4.000000000000000e-001   1.800000000000000e+000

   4.045808966861598e-001   1.829261425167858e+000

   4.041495644206274e-001   1.829386038547648e+000

   4.041494570206883e-001   1.829385925812154e+000

   4.041494570206443e-001   1.829385925812177e+000

EDU>>
```

Check convergence

Activity 3

Fixed Point Iteration (Activity)

- Solve the equation to yield the form $x = g(x)$
- $x^{[n+1]} = g(x^{[n]})$ (start with guess and iterate)
- $F(x) = x\sin(x) - 1 = 0$, can be solved to yield:
- $x^{[n+1]} = 1/\sin(x^{[n]})$, and iterate starting from initial guess $x^{[0]}$
- Convergence depends upon nature of curve in vicinity of root.
- Will not converge to root near $x = 2.77$ $|g'(x)| > 1$ *(see demoiter.m)*

x1= 7.086167 x0= 3.000000
x1= 1.389988 x0= 7.086167
x1= 1.016571 x0= 1.389988
x1= 1.176044 x0= 1.016571
x1= 1.083316 x0= 1.176044
x1= 1.131842 x0= 1.083316
x1= 1.104733 x0= 1.131842
x1= 1.119390 x0= 1.104733
x1= 1.111316 x0= 1.119390
x1= 1.115719 x0= 1.111316
x1= 1.113304 x0= 1.115719
x1= 1.114625 x0= 1.113304
x1= 1.113901 x0= 1.114625
x1= 1.114297 x0= 1.113901

x1= 1.114081 x0= 1.114297

x1= 1.114199 x0= 1.114081
x1= 1.114134 x0= 1.114199
x1= 1.114170 x0= 1.114134
x1= 1.114150 x0= 1.114170
x1= 1.114161 x0= 1.114150
x1= 1.114155 x0= 1.114161
x1= 1.114158 x0= 1.114155
x1= 1.114157 x0= 1.114158

x1= 1.114157 x0= 1.114157

- In this case, the algorithm converges to the root near x = 1.114, where $|g'(x)| < 1$
- Linearly convergent in this case.
- Convergence depends strongly on form of iterant, which is not unique.

$$f(x) = x^2 - x - 2 = 0$$
$$x = g(x) = x^2 - 2$$
$$x = g(x) = \sqrt{x+2}$$
$$x = 1 + 2/x$$
$$x = g(x) = \frac{x^2+2}{2x-1}$$

The first one diverges for most starting guesses.

The other three converge at greatly differing rates.

$$3x^3 - 5x^2 - 4x + 4 = 0$$

Apply fixed point iteration starting with $x_0 = 0.7$ Use the two iterants on p. 34.

$$x = \frac{5}{3} + \frac{4}{3x} - \frac{4}{3x^2}$$ Finds root = 2

$$x = 1 + \frac{3x^3 - 5x^2}{4}$$ Finds root = 2/3

Square Root Example

- A form of fixed point iteration.

$$x^2 = N$$
$$x^{[n]} = N/x^{[n-1]}$$

$$x^2 = N$$
$$2x^2 = x^2 + N$$
$$x^2 = (1/2)(x^2 + N)$$
$$x = (1/2)(x + N/x)$$
$$x^{[n]} = (1/2)\left(x^{[n-1]} + N/x^{[n-1]}\right)$$

The last iterant in the blue box is used after the procedure is scaled to require the square root of a number between 0.25 and 1.

Square root Iteration

Any number A can be written in the form

$$A = m \times 4^n, \text{ where}$$
$$n \text{ is integer and } \tfrac{1}{4} \le m \le 1$$

Then the square root is given by: $\sqrt{A} = \sqrt{m} \times 2^n$

The general square root problem reduces to finding the square root of a number between 1/4 and 1.

Square Root Example

- A form of fixed point iteration.

$$x^2 = N$$
$$x^{[n]} = N / x^{[n-1]}$$

Convergence Rate?

$$x^2 = N$$
$$2x^2 = x^2 + N$$
$$x^2 = (1/2)(x^2 + N)$$
$$x = (1/2)\left(x + N/x\right)$$
$$x^{[n]} = (1/2)\left(x^{[n-1]} + N/x^{[n-1]}\right)$$

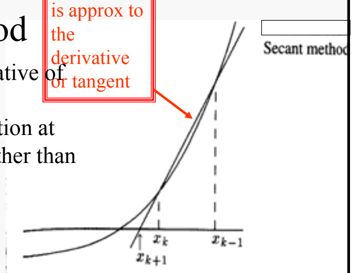
```
x1= 1.141299 x0= 0.500000
x1= 0.961125 x0= 1.141299
x1= 0.944237 x0= 0.961125
x1= 0.944086 x0= 0.944237
x1= 0.944086 x0= 0.944086
m= 0.891299 sqrt(m)= 0.944086
EDU>>
```

On page 36: this is also the Newton iterant.

Secant Method

- Does not require derivative of function.
- Requires value of function at two previous iterates rather than one.
- Only one new function evaluation per iteration.

Secant line is approx to the derivative or tangent



$$x_{k+1} = x_k - f(x_k)\frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})}$$

```
xk=2.5
fxk = demfun1(xk);
xkp1=4
while (abs(xkp1-xk)>tol)
   xkm1=xk;
   fxkm1=fxk;
   xk=xkp1;
   fxk=demfun1(xk);
   xkp1=xk-fxk*(xk-xkm1)/(fxk-fxkm1);
   fprintf('\nxkp1= %f  xk =
  %f',xkp1,xk)
end
```

•**Only one function evaluation per iteration.**

Secant Method
f(x) = x sin(x) -1

- Starting at points x = 2.5 and 3.5, converges in 6 iterations.
- convergence is still somewhat superlinear, although slower than Newton's method

```
xkp1= 2.682157  xk = 3.500000
xkp1= 2.746235  xk = 2.682157
xkp1= 2.774299  xk = 2.746235
xkp1= 2.772575  xk = 2.774299
xkp1= 2.772605  xk = 2.772575
xkp1= 2.772605  xk = 2.772605
```

Secant Method

- Starting at points x = 1.0 and 2.0 , converges in 4 iterations.
- convergence is still somewhat superlinear, although slower than Newton's method

```
xkp1= 1.162240  xk = 1.000000
xkp1= 1.114254  xk = 1.162240
xkp1= 1.114157  xk = 1.114254
xkp1= 1.114157  xk = 1.114157
```

$3x^3 - 5x^2 - 4x + 4 = 0$

Apply Secant method near x = 0.7 (#1 on p. 44)

Matlab *fzero* function.

- *fzero* utilizes a hybrid method, starting with bisection, switching to secant and to parabolic interpolation as appropriate.
- options = optimset('Display','iter','TolX',tol)

  z = fzero(@demfun1,x,options)
  - x scalar: searches for root near initial x
  - **x** vector: guarantees a root between x(1) and x(2) if *funname* has different signs at the two x-values.
  - *TolX* regulates convergence criterion
  - *"iter"* produces diagnostic output

fzero Output

```
pause
%use matlab library function fzerop
disp(' ')
disp ('library routine fzero')
x = [2.5 3.5]
trace=1
%z = fzero('d
options = opt
z = fzero(@de
```

```
library routine fzero
with initial guess interval
x =

   2.5000    3.5000

 Func evals      x            f(x)          Procedure
    1           2.5        0.49618        initial
    2           3.5       -2.22774        initial
    3       2.68216       0.189383        interpolation
    4       2.78579     -0.0295897        interpolation
    5       2.77179     0.00182104        interpolation
    6        2.7726    1.53111e-005       interpolation
    7        2.7726   -1.53621e-010       interpolation

 root found at x =    2.7726 »
```

Matrix Computations

CSS455 Winter 2011

Block Structure of Matrices

- Overall matrix can be viewed as constructed of rows and columns of smaller matrices or blocks.
- Care must be taken to preserve correct dimensions.
- Matlab will generally check dimensions

Block Structure

```
» A11 = fix(10*rand(3,3)

A11 =

     1     7     1
     6     4     4
     6     5     7

» A12 = fix(10*rand(2,3)

A12 =

     8     2     2
     2     8     8
```

```
» x = [A11 A12]
???  All matrices on a row in the
 same number of rows.

» A12 = fix(10*rand(3,2))

A12 =

     9     0
     2     0
     2     6

» x = [A11 A12]

x =

     1     7     1     9     0
     6     4     4     2     0
     6     5     7     2     6
```

Block Structure

```
>> A21 = fix(10*rand(

A21 =

     1     1
     8     1

>> A22 = fix(10*rand(

A22 =

     9     3     3
     4     3     3
```

```
>> row2 = [A21 A22]

row2 =

     1     1     9     3     3
     8     1     4     3     3

>> row1 = [A11 A12]

row1 =

     1     7     1     9     0
     6     4     4     2     0
     6     5     7     2     6
```

Block Structure

```
>> row2 = [A21 A2

row2 =

     1     1
     8     1

>> row1 = [A11 A1

row1 =

     1     7     1     9     0
     6     4     4     2     0
     6     5     7     2     6
```

```
>> A = [row1;row2]

A =

     1     7     1     9     0
     6     4     4     2     0
     6     5     7     2     6
     1     1     9     3     3
     8     1     4     3     3
```

Block Structure

```
>> row2 = [A21 A

row2 =

     1     1
     8     1

>> row1 = [A11 A

row1 =

     1     7
     6     4     4     2     0
     6     5     7     2     6
```

```
>> A = [A11 A12; A21 A22]

A =

     1     7     1     9     0
     6     4     4     2     0
     6     5     7     2     6
     1     1     9     3     3
     8     1     4     3     3
```

y = Ax
(Matrix-Vector Product)

- Each element of the product vector **y** is the result of an inner product (dot product) between a row of **A** (a row vector) and the column vector **x**

$$y_i = \sum_{k=1}^{n} a_{ik} x_k$$

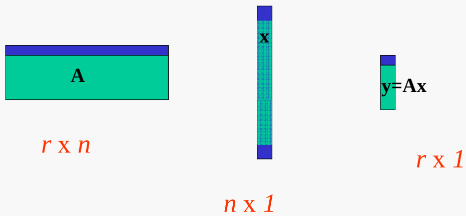
Work Group Project $\quad y_i = \sum_{k=1}^{n} a_{ik} x_k$

- Given the matrix-vector product
  y = Ax
  where **y** is an ($r \times 1$) column vector,
  x is an ($n \times 1$) and **A** is an ($r \times n$) matrix, write pseudo code for a suitable algorithm.
- Estimate in terms of r and n the number of floating point operations (multiplies, additions and subtractions) in this algorithm.

With Partner

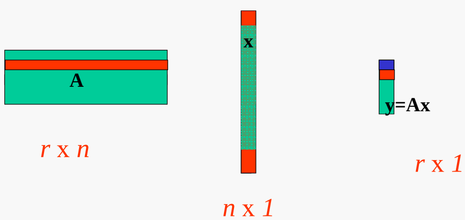
- Part I of Activity 4 - 5 minutes
- Describe your algorithm at board

The element $\mathbf{y}_1$ is formed by the scalar product of row-1 of **A** with column **x**.



$$y_1 = a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = \sum_{i=1}^{n} a_{1i}x_i$$

The element $\mathbf{y}_2$ is formed by the scalar product of row-2 of **A** with column **x**.



$$y_2 = a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = \sum_{i=1}^{n} a_{2i}x_i$$

Consider the elements of **y**

$$\boxed{\begin{matrix} a_{11}x_1 + a_{12}x_2 + \cdots a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + \cdots a_{2n}x_n \\ \ldots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots a_{mn}x_n \end{matrix}} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix}$$

Consider the elements of **y**

$$\begin{matrix} \mathbf{a_{11}x_1 + a_{12}x_2 + \cdots a_{1n}x_n} \\ a_{21}x_1 + a_{22}x_2 + \cdots a_{2n}x_n \\ \cdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots a_{mn}x_n \end{matrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix}$$

Row Ordered Algorithm

```
%compute the Ax product
y = zeros(m,1);
for i = 1:m
   y(i) = 0;
   for j = 1:n
      y(i) = y(i) + A(i,j)*x(j);
   end
end
%Repeat with  vector notation
y2 = zeros(m,1);
for i =1:m
   y2(i) = A(i,:)*x;
end
```

Matlab uses inner product in last loop.

Consider the elements of **y**

$$\begin{matrix} a_{11}x_1 + a_{12}x_2 + \cdots a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + \cdots a_{2n}x_n \\ \cdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots a_{mn}x_n \end{matrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix}$$

Consider the elements of **y**

$$
\begin{aligned}
&a_{11}x_1 + a_{12}x_2 + \cdots a_{1n}x_n \\
&a_{21}x_1 + a_{22}x_2 + \cdots a_{2n}x_n \\
&\cdots \\
&a_{m1}x_1 + a_{m2}x_2 + \cdots a_{mn}x_n
\end{aligned}
$$

Notice that there is a natural column oriented structure as well as the row structure. The code could be organized by column.

Column Oriented Approach - reverse the loop structure

```
% Use column oriented algorithm
y4 = zeros(m,1);
for j =1:n
   for i = 1:m
      y4(i) = y4(i) + A(i,j)*x(j);
   end
end
%Repeat with vector notation
y5 = zeros(m,1);
for j = 1:n
   y5 = y5 + A(:,j)*x(j);
end
```

In the inner loop x(j) is essentially a scalar constant

This can be thought of as "scalar a times vector x plus vector y" or "*axpy*". If in single precision, called "saxpy".

who cares?

```
%Repeat with matrix notation
y3 = A*x;
```

- In matlab, the compact notation makes this choice less than obvious:
- The results and the number of floating point operations are the same in the column and row oriented approaches.
- How could the choice be important?

Order of Data Storage

- If matrix A is stored row-wise:

$$a_{11} \quad a_{12} \quad \cdots \quad a_{1n} \quad a_{21} \quad a_{22} \quad \cdots \quad a_{2n} \quad a_{31} \cdots$$

•**The dot product approach might be efficient.**

•**If the data is stored column-wise, the *saxpy* approach might be more efficient:**

$$a_{11} \quad a_{21} \quad \cdots \quad a_{m1} \quad a_{12} \quad a_{22} \quad \cdots \quad a_{m2} \quad a_{13} \cdots$$

With Partner

- Part II of Acivity 4
- Go to board with it

Matrix vector product with upper triagular matrix

Row oriented (dot product)

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} \\ 0 & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} \\ 0 & 0 & a_{33} & a_{34} & a_{35} & a_{36} \\ 0 & 0 & 0 & a_{44} & a_{45} & a_{46} \\ & & & & a_{55} & a_{56} \\ & & & & 0 & a_{66} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{pmatrix}$$

$$y_1 = a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = \sum_{i=1}^{n} a_{1i}x_i$$

$$y_2 = a_{22}x_2 + a_{23}x_3 + \ldots + a_{2n}x_n = \sum_{i=2}^{n} a_{2i}x_i$$

$$y_j = a_{jj}x_j + a_{j,j+1}x_{j+1} + \ldots + a_{jn}x_n = \sum_{i=j}^{n} a_{ji}x_i$$

$$y_n = a_{nn}x_n$$

a(j,j:n)****x***(j:n) for each j (where * is dot product in Matlab)

Upper triangular - check code

```
% Set up banded matrix Ab
Ab = zeros(m,n);
Ab = triu(A,0);
yb1 = Ab*x;
yb2 = zeros(m,1);
for i= 1:m
   yb2(i)=Ab(i,i:n)*x(i:n);
end
[yb1 yb2]
```

```
ans =

    3.7595    3.7595
    2.7720    2.7720
    2.1410    2.1410
    1.6763    1.6763
    1.3094    1.3094
    1.0073    1.0073
    0.7511    0.7511
    0.5289    0.5289
    0.3331    0.3331
    0.1581    0.1581
```

This seems to give the correct answer.

Matrix vector product with upper triangular matrix

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} \\ 0 & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} \\ 0 & 0 & a_{33} & a_{34} & a_{35} & a_{36} \\ 0 & 0 & 0 & a_{44} & a_{45} & a_{46} \\ 0 & 0 & 0 & 0 & a_{55} & a_{56} \\ 0 & 0 & 0 & 0 & 0 & a_{66} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{pmatrix}$$

Column oriented

$y_1 = a_{11}x_1$
Col 1

$y_1 = y_1 + a_{12}x_2$
$y_2 = y_2 + a_{22}x_2$
Col 2

$y_1 = y_1 + a_{13}x_3$
$y_2 = y_2 + a_{23}x_3$
$y_3 = a_{33}x_3$
Col 3

$y_1 = y_1 + a_{16}x_6$
$y_2 = y_2 + a_{26}x_6$
$y_3 = y_3 + a_{36}x_6$
$y_4 = y_4 + a_{46}x_6$
$y_5 = y_5 + a_{56}x_6$
$y_6 = a_{66}x_6$
Col 6

y(1:j)=**y**(1:j) + ***a***(1:j,j)***x**(j)
for each j

Upper triangular - check code

```
%column oriented upper tri
yb3 = zeros(m,1);
for j = 1:n
   yb3(1:j)=Ab(1:j,j)*x(j) + yb3(1:j);
end
[yb1 yb2 yb3]
```

```
ans =

    3.7595    3.7595    3.7595
    2.7720    2.7720    2.7720
    2.1410    2.1410    2.1410
    1.6763    1.6763    1.6763
    1.3094    1.3094    1.3094
    1.0073    1.0073    1.0073
    0.7511    0.7511    0.7511
    0.5289    0.5289    0.5289
    0.3331    0.3331    0.3331
    0.1581    0.1581    0.1581
```

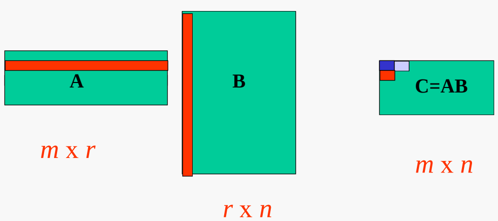
This seems to give the correct answer.

Matrix-Matrix products **C= AB**

$$C_{kj} = \sum_{i=1}^{r} A_{ki} B_{ij}$$

- Each element of product is an inner (dot) product between a row of A and one column vector.
- Each column of the product is a matrix vector product between **A** and one column of **B**.
- The entire **C** matrix is just a collection of matrix-vector products, and will have the same range of algorithms.

The element C_{21} is formed by the scalar product of row-2 of **A** with column-1 of **B**.



$$c_{21} = a_{21}b_{11} + a_{22}b_{21} + \ldots + a_{2n}b_{n1} = \sum_{i=1}^{r} a_{2i}b_{i1}$$

With Partner

- Activity 4, Part III
- Results

Work Group Project $C_{kj} = \sum_{i=1}^{r} A_{ki} B_{ij}$

- Given the matrix-matrix product
  C=AB
  where **A** is an $(m \times r)$ matrix, **B** is an $(r \times n)$ matrix and **C** is an $(m \times n)$ matrix write pseudo code for a suitable algorithm.
- Estimate in terms of m, n, and r the number of floating point operations (multiplies, additions and subtractions) in this algorithm.

C = AB in full triple loop notation

```
for k = 1:m
   for j = 1:n
      for i = 1:r
         C(k,j) = C(k,j)+ A(k,i)*B(i,j);
      end
   end
end
```

This is formally an n^3 operation.

The inner loop is a vector inner product.

Run the inner loop like an inner vector product.

```
%Run the inner loop like a dot product
C2 = zeros(m,n);
for k = 1:m
   for j = 1:n
         C2(k,j) = C2(k,j)+ A(k,:)*B(:,j);
   end
end
```

Reorder the loop structure and run like a *saxpy* operation.

```
for i = 1:r
   for j = 1:n
      for k = 1:m
         C(k,j) = C(k,j)+ A(k,i)*B(i,j);
      end
   end
end
C
C1
%Run the inner loop like a saxpyt
C2 = zeros(m,n);
for i = 1:r
   for j = 1:n
         C2(:,j) = C2(:,j)+ A(:,i)*B(i,j);
   end
end
```

Reorder the loop structure to make it look like a set of matrix-vector products. (*matlab operator)*

```
for j = 1:n
   for i = 1:r
      for k = 1:m
         C(k,j) = C(k,j)+ A(k,i)*B(i,j);
      end
   end
end

%Run the inner loop like a saxpyt
C2 = zeros(m,n);
for j = 1:n
            C2(:,j) = C2(:,j)+ A*B(:,j);
         end
C
```

Outer Product

- Column vector times a row vector is an outer product: (m x 1)(1 x n) = (m x n).
- The outer product produces a matrix.
- Reorder the loops to present the matrix product as a sum of outer products.

Reorder loops

```
%Reorder loops to look like outer prods
C4 = zeros(m,n);
for i = 1:r
   for j = 1:n
      for k = 1:m
         C4(k,j) = C4(k,j)+ A(k,i)*B(i,j);
      end
   end
end
```

- *k*-loop looks like column vector times scalar (saxpy)
- *jk* loops look like column vector times a row vector (outer product)
- *ijk* loops look like sum of outer products, each of which is (m x n).

Outer Product Formulation

```
%Reorder loops to look like outer prods
C4 = zeros(m,n);
for i = 1:r
   for j = 1:n
      for k = 1:m
         C4(k,j) = C4(k,j)+ A(k,i)*B(i,j);
      end
   end
end
%Write as sum of outer products
C5 = zeros(m,n);
for i = 1:r
            C5 = C5+ A(:,i)*B(i,:);
end
```

Windows Matlab timings

n	Dot	Saxpy	MatVec	Outer	Direct
10	0.0180	0.0003	0.0003	0.0022	0.0001
50	0.0056	0.0022	0.0002	0.0008	0.0001
100	0.0237	0.0103	0.0012	0.0052	0.0005
200	0.1166	0.0557	0.0066	0.0369	0.0039
400	0.8740	0.3439	0.0896	1.2878	0.0320
800	8.0158	2.5688	1.1946	12.6034	0.2466

>>

MatVec and Direct clearly fastest for all lengths. Both of them use matlab operator for most time consuming steps.

These are elapsed times are from tic/toc in wall-clock seconds.

Linux Matlab (#17)

n	Dot	Saxpy	MatVec	Outer	Direct
10	0.0342	0.0020	0.0026	0.0017	0.0000
50	0.0123	0.0047	0.0004	0.0012	0.0005
100	0.0553	0.0219	0.0019	0.0156	0.0003
200	0.2563	0.1084	0.0134	0.1606	0.0050
400	1.4821	0.7356	0.2074	0.9561	0.0157
800	13.4897	4.9986	1.7941	7.3177	0.1123

>>

Some times are slower, others are faster..

These are elapsed times in seconds.

Comparison

n	Dot	Saxpy	MatVec	Outer	Direct
10	0.0180	0.0003	0.0003	0.0022	0.0001
50	0.0056	0.0022	0.0002	0.0008	0.0001
100	0.0237	0.0103	0.0012	0.0052	0.0005
200	0.1166	0.0557	0.0066	0.0369	0.0039
400	0.8740	0.3439	0.0896	1.2878	0.0320
800	8.0158	2.5688	1.1946	12.6034	0.2466

>>

n	Dot	Saxpy	MatVec	Outer	Direct
10	0.0342	0.0020	0.0026	0.0017	0.0000
50	0.0123	0.0047	0.0004	0.0012	0.0005
100	0.0553	0.0219	0.0019	0.0156	0.0003
200	0.2563	0.1084	0.0134	0.1606	0.0050
400	1.4821	0.7356	0.2074	0.9561	0.0157
800	13.4897	4.9986	1.7941	7.3177	0.1123

>>

Activity 4 – Part IV