Conjugate Gradient Methods

Richard Fears and Jason Gregersen

Purpose

To test efficiency of different conjugate gradient methods on a variety of test problems.

Introduction

The Conjugate gradient method is a optimization technique that is very similar to a line search. We are trying to minimize some function $\phi(x)$ giving some starting position. The general line search process says to choose a downhill direction and take some step. We then choose another direction and another step. The specifics of the Conjugate Gradient method are, how do I form a new direction and how big is my step. Written as the skeleton of an algorithm we could say that if I have some point x_k that I found using the direction p_k, then I need to choose a new direction p_{k+1} and a step size α such that $x_{k+1} = x_1 + \alpha\, p_{k+1}$with the sequence of $x_{k's}$ converging to the minimum value. For the method of Conjugate Gradients, to choose our next step direction, we are going to take a linear combination of the gradient at our current point and our previous direction i.e.: $p_{k+1} = -\nabla f(x_1) + \beta\, p_k$ where β is chosen such that p_{k+1} and p_k are conjugate ($p_{k+1}\, A\, p_k = 0$). With this approach there are multiple choices for choosing β. Deciding which of these options is best on a specific problem, or in general, is the main purpose of this paper. The options we are going to choose between are ([4] pg 2):

the Fletcher-Reeves update	$B^{Fr}_{k+1} = \frac{\nabla f_{k+1}{}^{T} \nabla f_{k+1}}{\nabla f_k{}^{T} \nabla f_k}$,
the Polak-Ribiere update	$B^{Pr}_{k+1} = \frac{\nabla f_{k+1}{}^{T} (\nabla f_{k+1} - \nabla f_k)}{\lVert \nabla f_k \rVert^2}$
the Hestenes-Stiefel update	$B^{Hs}_{k+1} = \frac{\nabla f_{k+1}{}^{T} (\nabla f_{k+1} - \nabla f_k)}{(\nabla f_{k+1} - \nabla f_k)^{T}\, p_k}$
the Dai and Yuan update	$B^{Dy}_{k+1} = \frac{\lVert \nabla f_{k+1} \rVert^2}{(\nabla f_{k+1} - \nabla f_k)^{T}\, p_k}$
the Hager and Zhang update	$B^{num}_{k+1} = \left(\hat{y}_k - 2\, p_k \frac{\lVert \hat{y}_k \rVert^2}{\hat{y}_k{}^{T}\, p_k}\right)^{T} \frac{\nabla f_{k+1}}{\hat{y}_k{}^{T}\, p_k}$ with $\hat{y}_k = \nabla f_{k+1} - \nabla f_k$
the "Conjugate Descent" update	$B^{Cd}_{k+1} = \frac{\nabla f_{k+1}{}^{T} \nabla f_{k+1}}{(\nabla f_{k+1} - \nabla f_k)^{T}\, p_k}$,
the Lui and Storey update	$B^{Ls}_{k+1} = \frac{\nabla f_{k+1}{}^{T} (\nabla f_{k+1} - \nabla f_k)}{-(\nabla f_k)^{T}\, p_k}$
the "Conjugate Descent" update	$B^{Cd}_{k+1} = \frac{\nabla f_{k+1}{}^{T} \nabla f_{k+1}}{(\nabla f_{k+1} - \nabla f_k)^{T}\, p_k}$,

Testing Procedures

To compare the above update methods we will be comparing the number of iterations and the wallclock times to achieve a solution . Since we will also be using a variety of problems, we will get a chance to see the robustness of the algorithms as well.

Project Outline

1. We will code the Nonlinear Conjugate Gradient method as well as the various β_k updates mentioned above.
2. We will then compare the performance of the various algorithms on the following test problems.
   a. A high dimensional linear problem
   b. 2-D Non-linear problems (easy, H, Siam, Rosenbrock)
   c. Higher dimensional non-linear problems.

Code

We coded a version of the Conjugate Gradient code in which we were able to input the specific β_k being used. Once the new direction in the algorithm was computed we then chose to use Mathemtica's "Nminimize" command with an accuracy and precision goal of 4 to find the necessary step length α. In our Code we have also chosen to reset our value for β_k back to steepest descent after 20 iterations. This should account for the loss of conjugacy in directions that develops when using the algorithm for nonlinear functions.

Bk Updates

Conjugate Gradient Code

Miscellaneous Code

Testing Code

Functions

Initializations

2-D Testing

Linear

We first chose to run our code on a 30 dimensional linear problem. We are using a tolerance of 10^{-6} to end our iteration and a randomly generated inital condition. The command "linfun[]" seen below creates a SPD matrix "A" via code from [4]. It also generates a random "b" vector and then defines a function "f" as the quadratic form $f(x) = \frac{1}{2} x^T A x - x^T b$. In the process of testing the various β_k updates we encountered overflow trouble with several of the updates. Thus the results we are going to use to compare schemes will be based on the Pr, Fr, Dy, and Cd schemes only. As shown in the table below Pr and Fr were slightly less efficient, but they all were effective.

```
{A, b} = linfun[30];
```

```
Testing2[f, GoodSchemes]
```

scheme	time	loops	error
Pr	13.884	42	9.33964×10^{-7}
Fr	14.836	43	6.3385×10^{-7}
Dy	10.14	29	4.40477×10^{-7}
Cd	10.249	29	5.85574×10^{-7}

Since we are using a positive-definite matrix A, the minimum we find will be a global minimum. Also at the minimum the gradient will be equal to zero and since A is symmetric $\nabla f = A x - b = 0$, thus the solution to the minimization problem should be a solution to the system $A x - b = 0$. The results below verify this.

```
A.names[[4]][[5]] - b (*the 4 indicates that we are using Cd to test the results*)
```

$\{-8.8512 \times 10^{-8}, 9.63253 \times 10^{-8}, -1.55002 \times 10^{-7}, -4.36338 \times 10^{-8}, 3.53444 \times 10^{-8}, 7.71574 \times 10^{-8}, -4.25727 \times 10^{-9}, 7.51399 \times 10^{-8}, -1.47887 \times 10^{-7}, -5.89653 \times 10^{-8}, -7.99192 \times 10^{-8}, 1.2991 \times 10^{-7}, -4.82795 \times 10^{-8}, 1.99504 \times 10^{-7}, -9.83062 \times 10^{-8}, -1.95302 \times 10^{-8}, -1.21177 \times 10^{-7}, 1.5526 \times 10^{-7}, -1.08906 \times 10^{-7}, 1.37317 \times 10^{-7}, -9.72595 \times 10^{-11}, -4.50088 \times 10^{-8}, 1.08857 \times 10^{-7}, 4.47352 \times 10^{-8}, 6.99597 \times 10^{-8}, -2.29337 \times 10^{-7}, 9.33896 \times 10^{-8}, 5.54837 \times 10^{-8}, -1.61615 \times 10^{-7}, -7.35251 \times 10^{-8}\}$

Non-Linear Testing

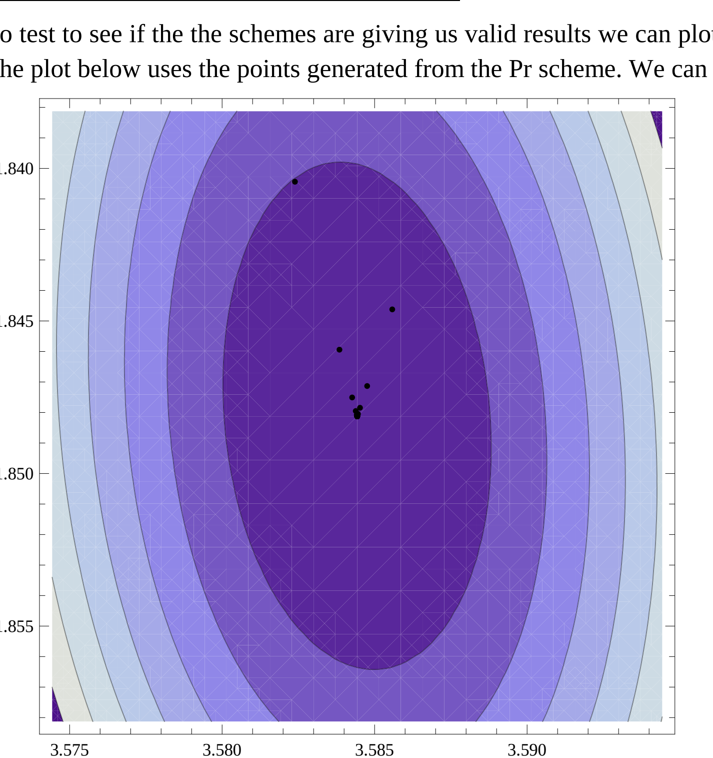
Next we want to test the schemes on some 2-D nonlinear functions, but first we need to make sure that that the CG code works on non-linear functions. Setting up the dimensions of our problem and the initial conditions we will run our code on the 2-D Himmelbleau function. The results below show that Fr and Cd are the most efficient. Pr is the least efficient and we expect that this is due to it getting jammed at some point and not working properly again until the restart at the 20th iteration.

```
dimension = 2;
y0 = RandomReal[10, dimension];
```

```
Testing2[H, {Pr, Fr, Dy, Cd}] // Quiet
```

scheme	time	loops	error
Pr	3.947	45	4.51919×10^{-10}
Fr	1.045	11	2.76126×10^{-10}
Dy	2.84	31	3.13287×10^{-10}
Cd	1.045	11	2.84322×10^{-10}

To test to see if the the schemes are giving us valid results we can plot the points generated during one of the schemes. The plot below uses the points generated from the Pr scheme. We can see that they are converging to a minimum value.



Now confident that the Code is running properly we can compare the performance of the different β_k schemes on an easy funtion as well as the Siam and Rosenbrock functions. Here our "easy" function is `easy[{x_, y_}] := x^2 + (y - 1)^2 - x` . On this function the solution was so easy that there is no comparison between the schemes. The fact that there is zero for the number of loops indicates that the solution was found in the first step without even running the main loop of the code.

```
Testing2[easy, GoodSchemes] // Quiet
```

scheme	time	loops	error
Pr	0.078	0	3.12348×10^{-7}
Fr	0.032	0	3.12348×10^{-7}
Dy	0.046	0	3.12348×10^{-7}
Cd	0.032	0	3.12348×10^{-7}

The results on the Siam function indicate once again that Pr seems to be less efficient (althought not the worst in this case).

```
Testing2[Siam, GoodSchemes] // Quiet
```

scheme	time	loops	error
Pr	8.097	31	1.23415×10^{-7}
Fr	5.6	10	2.38015×10^{-7}
Dy	6.724	20	6.208×10^{-7}
Cd	8.392	40	7.40466×10^{-7}

Te results on the Rosenbrock function indicate that this problem is difficult and all the schemes failed. Here you can also see that we have limited the number of iterations of the algorithm to 100. We believe that using a line search with more restrictions on the stepsize i.e. stronger wolfe conditions and other restrictions to ensure that the combination of β_k and α always result in a descent direction[4]

```
Testing2[Ros, GoodSchemes] // Quiet
```

scheme	time	loops	error
Pr	156.671	100	Overflow[]
Fr	168.248	100	Overflow[]
Dy	11.481	100	1.26008×10^{11}
Cd	9.532	100	0.71909

Higher Dimensional Problems

Next we will test the different schemes on nonlinear functions of higher dimensions. To generate non-trivial problems we are going to put a non-linear function in the first two dimensions of a larger linear problem. Then use a rotation matrix to mix up the dimensions. These steps are all done by "probgen". The input is the nonlinear function that goes in the first two dimensions, and the total dimension of the problem. The output is the SPD matrix and vector used to generate the linear dimensions, and the rotation matrix used to mix everything up.

```
dim = 4;
{B, c, Q} = probgen[Siam, dim];
y0 = ConstantArray[0, dim];
```

After generating the problem we solve it using the Dy update.

```
{sol, err, i} = ConjGrad[y0, bigrot, Tol, Dy];
```

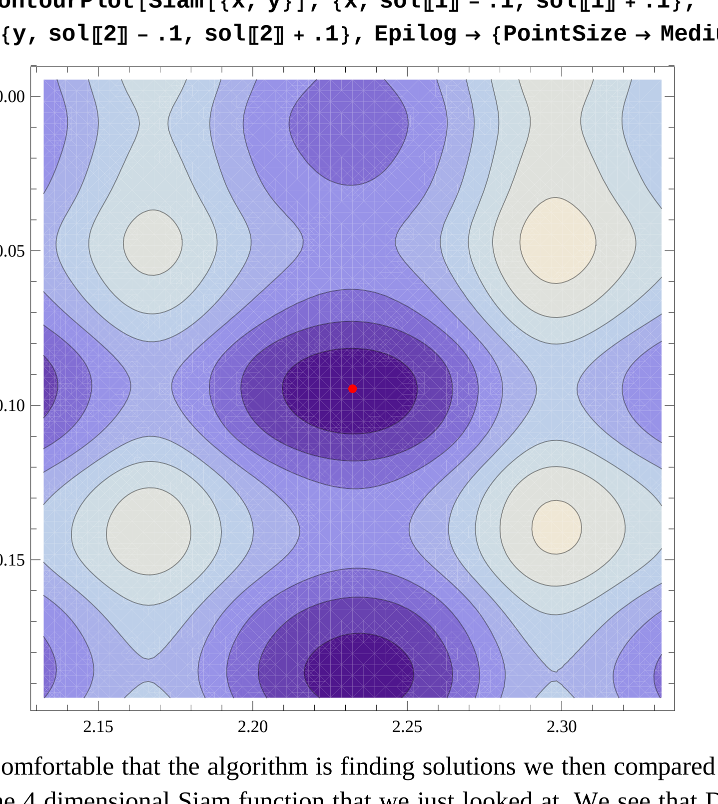
To test the result we need to un-rotate our solution and then check to see if the solution behaves as expected. Thus after the unrotating, we calculate the residual of the linear system and then plot the solution for the non-linear system, with the results shown below. Notice that although the residual is still relatively high, the plot of the solution appears to be right on. This is a result of the large gradient values in the siam function. The conjugate gradient tends to work on the hard stuff first, thus the focus was on dealing with the nonlinear portion.

```
sol = Transpose[Q].sol;
```

```
B.sol[[3 ;; dim]] - c
```

```
{-0.00192868, -0.000460561}
```

```
ContourPlot[Siam[{x, y}], {x, sol[[1]] - .1, sol[[1]] + .1},
 {y, sol[[2]] - .1, sol[[2]] + .1}, Epilog → {PointSize → Medium, Red, Point[sol[[1 ;; 2]]]}]
```



Comfortable that the algorithm is finding solutions we then compared the different updates. The first result is for the the 4 dimensional Siam function that we just looked at. We see that Dy did the best on this function and as we dis-cussed the althought the error might look high , the plot results were more comforting.

```
Testing2[bigrot, GoodSchemes] // Quiet
```

scheme	time	loops	error
Pr	11.747	100	0.771865
Fr	11.81	100	2.11306
Dy	12.417	100	0.00242763
Cd	18.003	100	40.7109

Given the underwhelming results of the previous example we decided to try to use an easier function but at higher dimensions. We took our "easy" function and tested it at 7, 10 and 20 dimensions. the results all suggest that Dy and Cd were much quicker and took signifigantly fewer function evaluations.

```
TestingBig[easy, 7, GoodSchemes]
```

scheme	time	loops	error
Pr	7.956	54	8.70054×10^{-7}
Fr	7.161	51	9.46829×10^{-7}
Dy	0.92	6	5.12765×10^{-7}
Cd	0.936	6	5.28924×10^{-7}

```
TestingBig[easy, 10, GoodSchemes]
```

scheme	time	loops	error
Pr	9.11	33	8.08468×10^{-7}
Fr	9.095	33	8.23361×10^{-7}
Dy	3.011	11	7.01618×10^{-7}
Cd	3.011	11	7.19439×10^{-7}

```
TestingBig[easy, 20, GoodSchemes]
```

scheme	time	loops	error
Pr	99.685	40	9.71007×10^{-7}
Fr	101.245	41	7.18119×10^{-7}
Dy	40.903	16	9.73111×10^{-7}
Cd	40.389	16	8.38868×10^{-7}

Next we tried to test the different updates on the Himmelbleau function at four dimensions. The results below showed that none of the updates yielded good results even after increasing the maximum number of iterations to 200. It does however look like while Dy and Cd appear to be divergent, the Pr and Fr schemes to be making some kind of progress.

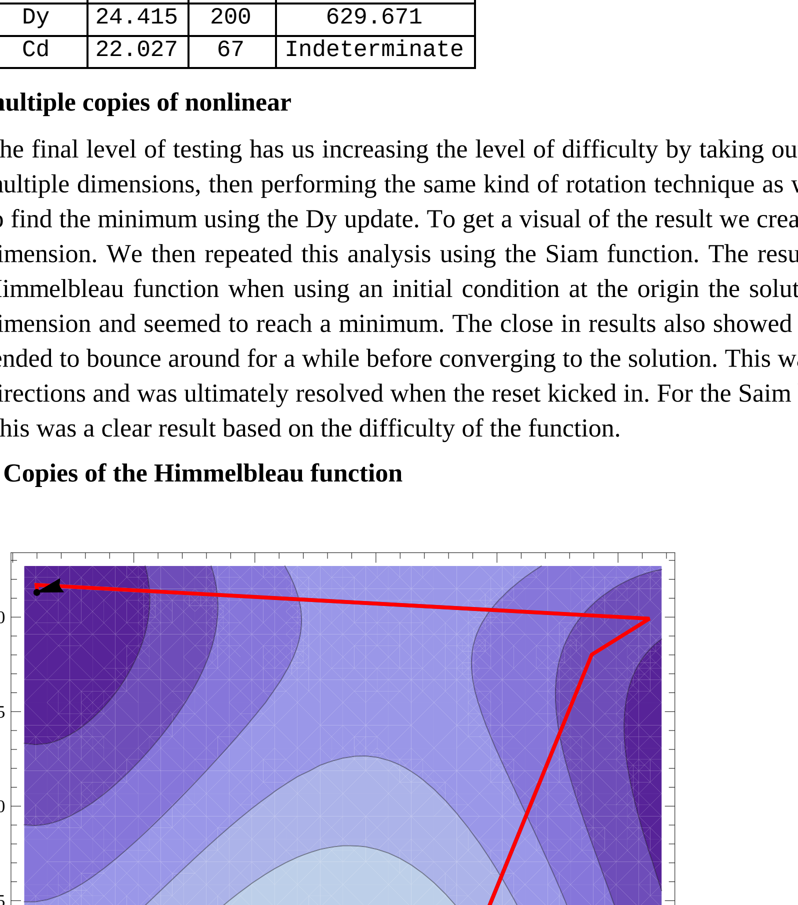
```
TestingBig[H, 4, GoodSchemes]
```

scheme	time	loops	error
Pr	21.341	200	0.00302546
Fr	20.529	200	0.00578591
Dy	24.415	200	629.671
Cd	22.027	67	Indeterminate

multiple copies of nonlinear

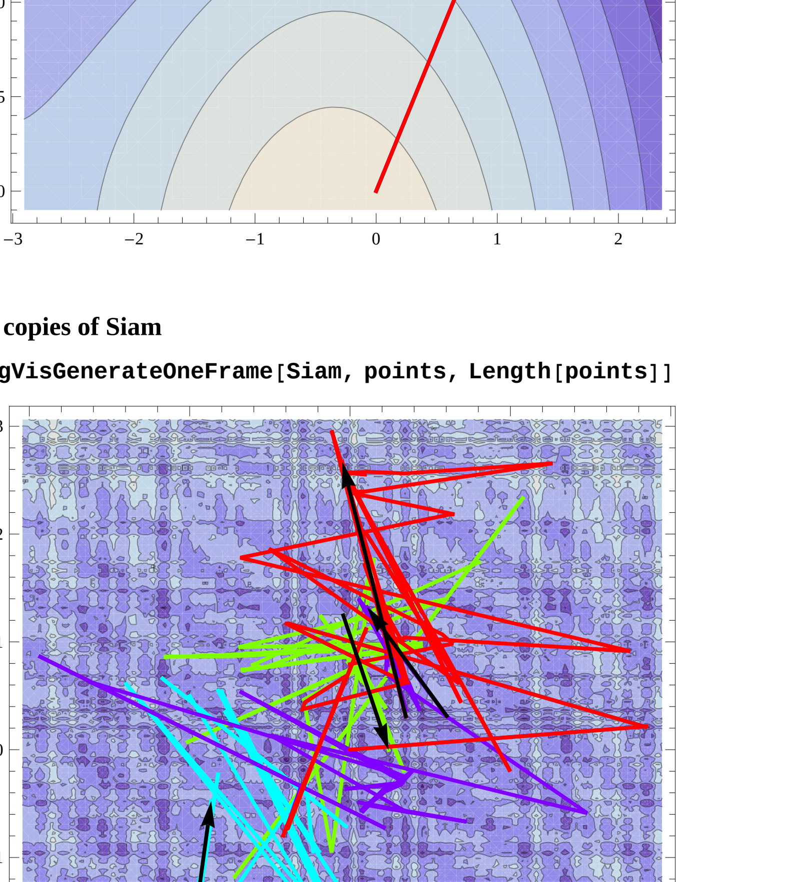
The final level of testing has us increasing the level of difficulty by taking our nonlinear function and copying it into multiple dimensions, then performing the same kind of rotation technique as was previously used. We then attempted to find the minimum using the Dy update. To get a visual of the result we created a plot of the solution for each pair of dimension. We then repeated this analysis using the Siam function. The results shown below indicate that for the Himmelbleau function when using an initial condition at the origin the solutions followed the same path in each dimension and seemed to reach a minimum. The close in results also showed thtat upon nearing the valley the points tended to bounce around for a while before converging to the solution. This was a result of the loss of conjugacy in our directions and was ultimately resolved when the reset kicked in. For the Saim however, the results were more sporadic. This was a clear result based on the difficulty of the function.

4 Copies of the Himmelbleau function



4 copies of Siam

```
cgVisGenerateOneFrame[Siam, points, Length[points]]
```



Conclusion

To summarize, we can say that there exist several different options for choosing the parameter β_k in the conjugate gradient method. The efficiency and stability of the different updates seems to vary on different problems. In general Dy seemed to be very efficient, but was outperformed by Fr and Pr on the 4 copy version of the Himmelbleau function. This variation of efficiency in the schemes is what has lead to the creation of hybrid schemes that switch between schemes to gain efficiency. As an example a scheme may run combinations of Pr and Fr or Dy and Hs. These choices tend to pair updates with strong convergence properties with those with may not convergen in general but if they do tend to be more efficient. For more information on Hybrid method see Hagar and Zhang[4, page6].

Reference

[1] Nocedal and Wright (2006). NumericalOptimization 2nd ed. Springer

[2] http://reference.wolfram.com/mathematica/tutorial/UnconstrainedOptimizationConjugateGradientMethods.html

[3] Struthers (2011). LinearConjugateGradient. http://www.mathlab.mtu.edu/~struther/Courses/5630_11/LinearConjugateGradient.nb

[4] Hagar and Zhang (2006), A survey of Nonlinear Conjugate Gradient Methods

[5] Jonathan Richard Shewchuk, An Introduction to the Conjugate Gradient Method Without the Agonizing Pain